

# **The Situated Software Architect**

A thesis submitted to  
the School of Information Technology  
of Monash University  
in fulfilment of the requirements for  
Doctorate of Philosophy (Computing)

By  
Paul Raymond Taylor  
December 2007

Monash University  
School of Information Technology

This thesis has not been submitted for the award of any degree or diploma in any other tertiary institution. No other person or person's work has been used without due acknowledgment. The body of this thesis (Chapters One to Ten inclusive) is less than one hundred thousand words.

Paul Raymond Taylor

31<sup>st</sup> December 2007

## **Acknowledgments**

The author would like to acknowledge the following people for their invaluable assistance in this research: Associate Professor Christine Mingins for her ability to open doors and inspire her students, Professor Richard Mitchell (Inferdata, formerly of University of Brighton, UK) for his willingness to lend his intellect to my enquiries, the twenty four software architects who engaged with this project for their willingness to participate and their professionalism, the Department of Software Development (and its successors) within the School of Information Technology at Monash University for the leadership it demonstrated to Australian academia and industry over a fifteen year period, and my wife, Heather, who encouraged and tolerated this mid-career dalliance into the wonderful world of ideas.

# **The Situated Software Architect**

**School of Information Technology,**

**Monash University**

Paul R. Taylor, 2007

Supervisor: Associate Professor Christine Mingins

## **Abstract**

This thesis is an examination of the practice of software design amongst professional Australian software architects. It is based on a qualitative analysis of the data arising from transcripts of in-depth interviews with twenty-four software architects and two qualitative case studies of software-architectural design within large Australian organisations. The thesis argues that the practice of software design can only be understood in a holistic sense via a pluralist perspective, and that a constructivist philosophical approach is necessary to account for the practice of software design in a meaningful way. This argument discredits rationalism and by implication modernism as inappropriate philosophical bases for the universal observation, description and comprehension of the practice of software design. The motivation for pursuing this line of investigation is twofold. Firstly, the research seeks to explicate rich and descriptive accounts of the practice of software design in Australian business and industry contexts via an interpretivist research paradigm. Secondly, it attempts to explain the apparently *a*-rational behaviours of professional software architects in order to separate the practitioner from a sense of inadequacy or failure that follows from the universal imposition of rationalistic design methods and paradigms. The thesis pursues these goals by attempting to fit a situated model of design to the accounts of the participants. Structuring the research process and the rich narratives that emerge from the qualitative analysis is achieved using four philosophical perspectives—rationalism, pragmatism, criticalism and radicalism. The research delivers original findings on how practicing software architects negotiate requirements in the light of known solutions, how they are bound by perspective and paradigm, and the mechanics of their personal design processes. Through these descriptions of activity, the architect is revealed as an arranger of context and situation so as to constitute an environment in which to design tacitly. The thesis concludes that a situated perspective is necessary to understand the practice of software design.

## Table of Contents

<b>Chapter 1:</b>	<b>Introduction</b>	<b>1</b>
1.1	Orientation	1
1.2	Purpose	2
1.3	Problem Statement	3
1.3.1	The role of <i>a</i> -rational forces in design outcomes	3
1.3.2	Apportionment of responsibility	4
1.3.3	Recognition of the significance of situation and situatedness	5
1.4	Research Aim	6
1.5	Scope	8
1.5.1	Relationship to software engineering	8
1.5.2	Centrality of the designer	9
1.5.3	The designer's skills profile	9
1.5.4	Types of architecture	9
1.5.5	Relationship between architecting and methodology	10
1.5.6	Relationship between architecting and coding	10
1.5.7	Relationship to Agile methods	11
1.6	Hypothesis	11
1.7	Overview of the study	12
1.7.1	Background	13
1.7.2	Research design	13
1.7.3	Field work and analysis	14
1.7.4	Findings, discussion and conclusions	14
1.8	Conclusion	14
<b>Chapter 2:</b>	<b>Four Perspectives on Software Architecture</b>	<b>16</b>
2.1	Introduction	16
2.2	Handles on Design	16
2.2.1	Four definitive sociological paradigms	17
2.2.2	About definitions and theory	21
2.3	Rationalism	21
2.3.1	Theory and practice	22
2.3.2	Software as a design medium	23
2.3.3	The Failure of the Master-plan	25
2.3.4	Alternatives to the Master-plan	28
2.4	Pragmatism	30
2.4.1	Situated Action	31
2.4.2	Design, Context and Culture	34
2.4.3	Design and History	36
2.4.4	Pragmatism versus bounded rationality	38

<b>2.5</b>	<b>The Situated Software Architect</b>	<b>39</b>
2.5.1	The Rational Software Architect	39
2.5.2	The Pragmatic Software Architect	41
2.5.3	The Critical Software Architect	43
2.5.4	The Radical Software Architect	44
<b>2.6</b>	<b>Conclusion</b>	<b>45</b>
<b>Chapter 3:</b>	<b>Design Theory and Software Design</b>	<b>47</b>
<b>3.1</b>	<b>Introduction</b>	<b>47</b>
<b>3.2</b>	<b>Philosophical Foundations of Design Theory</b>	<b>48</b>
3.2.1	Aristotelian and Platonic Concepts	49
3.2.2	Separation of Mind and Body	53
3.2.3	The Design of Existence	54
3.2.4	The Constructivist Alternative	56
3.2.5	Romantic Constructions of Design	58
<b>3.3</b>	<b>Models of Design</b>	<b>61</b>
3.3.1	Categories of Design Models	62
3.3.2	Vernacular design models	65
3.3.3	Evolutionary Design Models	70
3.3.4	Technology Maturity Models	71
<b>3.4</b>	<b>Conclusion</b>	<b>74</b>
<b>Chapter 4:</b>	<b>Situated and Ethnographic Accounts of Design</b>	<b>75</b>
<b>4.1</b>	<b>Introduction</b>	<b>75</b>
<b>4.2</b>	<b>Situated Cognition and Action</b>	<b>75</b>
4.2.1	Situated cognition	77
4.2.2	Situated action	79
4.2.3	Investigations of situatedness	81
<b>4.3</b>	<b>Design in Software Engineering</b>	<b>82</b>
4.3.1	Waterfall	82
4.3.2	Post-waterfall	83
4.3.3	All-at-once	85
4.3.4	The engineering metaphor	86
<b>4.4</b>	<b>Socio-cultural Models of Design</b>	<b>87</b>
4.4.1	Ethnographic design research	88
4.4.2	Engineering design research	92
4.4.3	Software design research	94
<b>4.5</b>	<b>Conclusion</b>	<b>97</b>
<b>Chapter 5:</b>	<b>Research Design</b>	<b>99</b>
<b>5.1</b>	<b>Introduction</b>	<b>99</b>
<b>5.2</b>	<b>Applicable Research Paradigms</b>	<b>100</b>
5.2.1	Applicable research types	101

5.2.2	Interpretivism	101
<b>5.3</b>	<b>A Method for Researching Situated Software Design Practice</b>	<b>103</b>
5.3.1	Defining the Acceptable Participant	104
5.3.2	Research Methods and Techniques	104
5.3.3	Sampling	105
5.3.4	A framework for eliciting descriptions of situated software design	106
<b>5.4</b>	<b>Assessment of the Research Method</b>	<b>107</b>
5.4.1	Researcher's Relationship to Participants	107
5.4.2	The Nature of Expert Recall	108
5.4.3	Generalisability	110
5.4.4	Ethics	110
5.4.5	Relevance	111
5.4.6	Rigour	112
5.4.7	The Role of Hermeneutics	112
<b>5.5</b>	<b>Conclusion</b>	<b>113</b>
<b>Chapter 6:</b>	<b>A Grounded Theory Model of Software Design Practice</b>	<b>115</b>
<b>6.1</b>	<b>Introduction</b>	<b>115</b>
6.1.1	On the use of pseudonyms	115
6.1.2	On the qualitative analysis	116
<b>6.2</b>	<b>What is 'Software Architecture'?</b>	<b>116</b>
6.2.1	Context and equilibrium	118
6.2.2	Interpreting requirements and goals	118
6.2.3	Purpose	120
6.2.4	Return on investment	121
6.2.5	The role of creativity	122
6.2.6	Planning tensions	124
6.2.7	Structure and shape	126
<b>6.3</b>	<b>What is 'Software Design'?</b>	<b>128</b>
6.3.1	Design as 'resolution of forces'	129
6.3.2	Emergence	130
<b>6.4</b>	<b>The 'Software Architect' Role</b>	<b>131</b>
6.4.1	Architect as salesperson	131
6.4.2	Impact of team capability	132
6.4.3	Illusion of progress	133
6.4.4	Influence of prevailing culture	134
6.4.5	Career investment and subversion	134
<b>6.5</b>	<b>Methodology</b>	<b>135</b>
6.5.1	Use, misuse and dissatisfaction	135
6.5.2	Risk mitigation and transfer	137
6.5.3	Technology churn invalidates method detail	137
6.5.4	Relationship between experience and method	138
<b>6.6</b>	<b>The design act</b>	<b>140</b>
6.6.1	Problem space considerations	142
6.6.2	Solution-based constraints	143
6.6.3	Candidate solutions	144

6.6.4	Complexity vs simplicity	146
6.6.5	Ontology	147
6.6.6	Conceptual view	149
6.6.7	Static view	149
6.6.8	Dynamic view	151
6.6.9	Historical view	152
6.6.10	Bias, perspective and perspective-shifting	152
6.6.11	Abstraction	154
6.6.12	Abstraction discovery	158
6.6.13	Generators	160
6.6.14	Crystallisation	162
6.6.15	Archetypes	163
6.6.16	Personal patterns	167
6.6.17	Intuitive leap	169
6.6.18	Breakdowns	170
6.6.19	Aesthetic	172
6.6.20	Habitation and aesthetic	173
6.6.21	Application of aesthetic	174
<b>6.7</b>	<b>Conclusion</b>	<b>176</b>
6.7.1	Definitions and Context	176
6.7.2	Design act	177
<b>Chapter 7:</b>	<b>Case Studies in Situated Software Design</b>	<b>181</b>
<b>7.1</b>	<b>Introduction</b>	<b>181</b>
<b>7.2</b>	<b>Case 1: The Naissance of the Decision Tree</b>	<b>182</b>
7.2.1	Actors and Roles	182
7.2.2	The Business Context	183
7.2.3	Design episodes 1 and 2: First attempts	185
7.2.4	Design episode 3: A data-oriented alternative	187
7.2.5	Design episode 4: A rule-oriented solution	189
7.2.6	Designer's reflections on the design process	192
7.2.7	Relationship to Grounded Theory	193
<b>7.3</b>	<b>Case 2: Perspective-bias in a Telecommunications Architecture</b>	<b>195</b>
7.3.1	Actors and Roles	195
7.3.2	The Business Context	196
7.3.3	Design episode 1: Pursuit of perfection	196
7.3.4	Design episode 2: Collapse and re-conceptualisation	198
7.3.5	Designer's reflections on the design process	200
7.3.6	Relationship to Grounded Theory	201
<b>7.4</b>	<b>Conclusion</b>	<b>202</b>
7.4.1	'Design episode' structures collaboration	202
7.4.2	Collaboration serves evaluation	203
<b>Chapter 8:</b>	<b>Findings</b>	<b>205</b>
<b>8.1</b>	<b>Introduction</b>	<b>205</b>
<b>8.2</b>	<b>Findings</b>	<b>205</b>



8.2.1	Architect in context	206
8.2.2	Architect as professional	207
8.2.3	Architect as negotiator	208
8.2.4	Architect as collaborator	211
8.2.5	Architect's use of methodology	212
8.2.6	Architect as abstractionist	213
8.2.7	Architect's memory	218
8.2.8	The 'design act'	221
<b>8.3</b>	<b>Conclusion</b>	<b>226</b>
<b>Chapter 9:</b>	<b>Discussion</b>	<b>228</b>
<b>9.1</b>	<b>Introduction</b>	<b>228</b>
<b>9.2</b>	<b>Rational themes</b>	<b>228</b>
9.2.1	Rationalism and subjectivity	229
9.2.2	Rationalism and negotiation	230
9.2.3	Rationalism and method	231
9.2.4	Plans as predictors of the design trajectory	234
9.2.5	Rationalism versus emergence—the design episode	237
9.2.6	Does rationalism have a role in explaining software design practice?	239
<b>9.3</b>	<b>Pragmatic themes</b>	<b>239</b>
9.3.1	Pragmatism and the 'social contract'	240
9.3.2	Pragmatism and paradigm	241
9.3.3	Pragmatic knowledge structures	242
9.3.4	Pragmatism and the design act	242
9.3.5	Pragmatism and method	244
9.3.6	Aesthetic as reflective practice	246
9.3.7	Does pragmatism have a role in explaining software design practice?	247
<b>9.4</b>	<b>Critical themes</b>	<b>248</b>
9.4.1	Critical analysis and the design engagement	248
9.4.2	Design as a means of resolving inequities	250
9.4.3	Criticalism and method	251
9.4.4	Does criticalism have a role in explaining software design practice?	252
<b>9.5</b>	<b>Radical themes</b>	<b>252</b>
9.5.1	Radicalism and conceptualisation	253
9.5.2	Does radicalism have a role in explaining software design practice?	254
<b>9.6</b>	<b>Conclusion</b>	<b>255</b>
<b>Chapter 10:</b>	<b>Conclusions</b>	<b>256</b>
<b>10.1</b>	<b>Introduction</b>	<b>256</b>
<b>10.2</b>	<b>Response to the hypothesis</b>	<b>257</b>
10.2.1	Revisiting definitions	257
10.2.2	Evaluation of findings against Chapter Five's design framework	258
10.2.3	Conclusions	261
<b>10.3</b>	<b>Generated hypotheses</b>	<b>262</b>
10.3.1	Perspective-shifting	262

10.3.2 The episodic nature of the design act	263
10.3.3 Methodological support for criticalism and radicalism	263
<b>10.4 Reflections on research practice</b>	<b>264</b>
<b>10.5 Contribution</b>	<b>267</b>
10.5.1 Originality	267
10.5.2 Contribution to design theory	267
10.5.3 Contribution to software engineering	269
<b>10.6 Conclusion</b>	<b>271</b>
 <b>Appendix A: Glossary</b>	 <b>274</b>
 <b>Appendix B: Ethics Clearance</b>	 <b>276</b>
 <b>Appendix C: Interview Pack</b>	 <b>281</b>
 <b>Appendix D: Participant Profile</b>	 <b>290</b>
 <b>Appendix E: Example of Topic Analysis</b>	 <b>301</b>
 <b>Appendix F: Topic Maps</b>	 <b>311</b>
 <b>Appendix G: Project Website</b>	 <b>317</b>
 <b>Appendix H: Author's Publications</b>	 <b>337</b>
 <b>Appendix I: Bibliography</b>	 <b>339</b>
 <b>Appendix J: Index</b>	 <b>357</b>

## Figures

Figure 1: A map of the thesis.....	13
Figure 2: Four sociological paradigms for organisational analysis.....	18
Figure 3: Relationship between philosophy, theory and models, with examples.....	50
Figure 4: Comparison between the scientific analysis process (left) and the design process (right) (Figs 1.1 and 1.2 in (Budgen 1994)).....	52
Figure 5: Three-dimensional Vitruvian design map for a stone axe, reproduced from Figure 4 in (Mayall 1979).....	62
Figure 6: ‘Vernacular design’ (Fig 72, p. 58); ‘the empirical exchange’ (Fig 71, p. 58); ‘direct patronage’ (Fig 70, p. 57) (Walker and Cross, 1976).....	66
Figure 7: Spiral model of knowledge phases and transitions (Nonaka and Takeuchi 1995).....	68
Figure 8: The ‘mature’ design process—the ‘rational design network’, from Fig 70 in (Walker and Cross, 1976).....	72
Figure 9: The four design modes compared, from Fig 73 in (Walker and Cross, 1976).....	73
Figure 10: Lyytinen’s systems development process framework (Lyytinen, 1987).....	89
Figure 11: A framework for the management of meaning in design (Markus and Bjorn-Andersen, 1987).....	90
Figure 12: A social action model of the organisational information system design process (Fig 2 in (Gasson, 1999)).....	92
Figure 13: The architect’s design context as described by the architects.....	177
Figure 14: The ‘design act’ as described by the architects.....	180
Figure 15: Original <i>Product</i> design (first design episode) as presented for peer review. ...	185
Figure 16: <i>Product</i> model redrawn (second design episode). ....	186
Figure 17: <i>Product</i> design—data-oriented version (third design episode).....	188
Figure 18: <i>Product</i> design—rule-based version (design episode four).....	191
Figure 19: Conceptual sketch of <i>Le Corbusier’s</i> exchange-centric model.....	198
Figure 20: Conceptual sketch of <i>Mackintosh’s</i> alternative model. ....	199
Figure 21: Simon’s Generate/Test cycle (Simon 1985) and a model of the participant’s reported ‘design episode’ phenomenon. ....	223
Figure 22: Relationship between design engagement, phase, iteration, and episode.....	224
Figure 23: Frequency of participant pseudonym occurrences in this thesis’ qualitative analysis chapter.....	265
Figure 24: Participant’s professional roles.....	292
Figure 25: Participant’s years in roles.....	292
Figure 26: Participant’s number of object-oriented architectures or systems.....	293
Figure 27: Participant’s largest architecture or system (classes).....	293
Figure 28: Participant’s longest time with one architecture or system. ....	294
Figure 29: Participant’s architecture and design responsibilities held. ....	295

Figure 30: Participant's points of engagement with the development process. ....	295
Figure 31: Participant's use of methodology. ....	296
Figure 32: Participant's use of object-oriented languages. ....	297
Figure 33: Participant's use of persistence frameworks and products. ....	297
Figure 34: Participant's use of object-oriented class libraries. ....	298
Figure 35: Participant's use of object-oriented distribution technologies. ....	299
Figure 36: Participant's use of CASE products. ....	299
Figure 37: Topic map for 'software architecture'. ....	312
Figure 38: Topic map for 'software design'. ....	313
Figure 39: Topic map for 'the role of the software architect'. ....	314
Figure 40: Topic map for 'methodology'. ....	315
Figure 41: Topic map for 'the design act'. ....	316

## Tables

Table 1: Comparison between rational and constructivist philosophies.....	20
Table 2: Comparison between rational and constructivist (situated) design. ....	61
Table 3: Characteristics of the design act—a framework for structuring interviews. ....	107
Table 4: Interview metrics. ....	291

# Chapter 1: Introduction

There is no such thing as genuine knowledge and fruitful understanding except as the offspring of doing. (Dewey 1916, p. 275)

## 1.1 Orientation

This thesis investigates the place of design practice in the art, craft and science of software development. The particular practice investigated is that of software architecture design, and the means by which it is investigated is through its agents, professional software architects. This investigation results in a set of findings based on grounded theory that yield rich and informative views of the expert software architect's practice. The resultant theory has implications for software design methodology, design process, and the management of design.

The central theme throughout this thesis is design—the creative act of synthesising solutions and artefacts amidst conflicting requirements and multi-dimensional constraints. Design is the central activity in all forms of making. Our ability to design makes us unique as a species. We design from within our consciousness, and our consciousness of our surroundings (and our ability to change them) frames design in the human experience. Almost every conceivable aspect of modernity is shaped by, or depends upon design. Even so, no unifying theory exists to explain the range of human cognition, behaviours and skills called upon by design. As a result, tensions in design practice remain unaddressed in many ubiquitous design domains.

The construction of an understanding of design in a particular context necessitates the close examination of rationality, objectivity, cognition, being, action and practice. In the case of the design of software architecture, theory comes from three overlapping sources—philosophy, the theoretical discipline of design science, and the applied discipline of

software engineering. The research is therefore interdisciplinary and heterogeneous in the theory it draws from, the methods it uses and the practices it surveys. The body of the research—a qualitative analysis of interview and case study data collected from expert software architects—puts form to how software designers approach design *in situ*. The analytical themes that emerge from these accounts concern how the design of software architecture is practiced, perceived and valued by the architects themselves. The resulting findings and models reveal forces in the design context and behaviours amongst the designers that have until recently been undocumented, and explain what otherwise might be interpreted as unpredictable, illogical or irrational design outcomes. Further interpretation of the findings yields theories of designer practice that account for situational forces in the design context. The informed view of software design practice these findings collectively afford has implications for both theory and practice.

## 1.2 Purpose

The primary purpose of this research is to be descriptive of an area of design practice that has received scant attention from researchers to date—the perceptions, motivations, experiences and values of highly experienced, practicing software architects. The research focuses on the effect of situation on design practice, as distinct from examining design expertise in isolation of the design context. The reasons for this inattention may have to do with problems of accessibility, definition and description. This kind of research is dependent upon identifying a suitably homogeneous group of designers with a sufficiently long and varied history in software architecture design. Such designers are difficult to recruit and access due to a range of problems that include identification, selection, screening, availability and privacy concerns. In Australia, Frampton (2005) accessed professional architects in IBM Australia’s architecture group. Robertson (2003) recruited twenty-six information architects, mostly in Sydney, for research on design and decision-making (Robertson 2004; Robertson and Hewlett 2004). Recognising a shortage of world class data modellers locally, Simsion (2005) supplemented a small cohort of Australian professionals with recruits sourced from international Data Administration and Management Association (DAMA) workshops.

There are also problems of definition. The terms ‘software architecture’ and ‘software architect’ are not intrinsically definitive and this ambiguity feeds the immaturity of the discipline. Shaw and Garlan (1996) describe software architecture as being traditionally based on a ‘substantial folklore’ of system design with little consistency or precision.

Other authors confirm informality of practice (Glass 1999; McBreen 2002). In the light of such claims, this thesis aims to determine to what degree such statements are true, and if they are true, to expose and express with clarity the reasons why the practice has been, is, and may remain folkloric.

The reward for overcoming these obstacles is the elicitation of a generalised and rich picture of a class of highly skilled, highly valued designers. The resulting descriptions serve to illuminate how the work of software designers aligns with (or differs from) those working in different design domains. Within the discipline of software methods, such a description is valuable in understanding the forces bearing upon software methods in contemporary design and development contexts—forces that have begun to marginalise some aspects of traditional software engineering methodology in favour of stripped-down, expeditious variants (Cockburn 2002; Palmer and Felsing 2002).

### 1.3 Problem Statement

Most research is motivated by perception of a problem, such as a gap or inadequacy in the incumbent paradigm or theory, or a discrepancy between theory and practice. The problem that motivates this research has its origins in the author's professional practice (and observation of other software designer's practice) in business and industry settings over fifteen years. It is important that the nature of these perceived problems are clearly stated so that the reader can make an assessment of the author's motives in pursuing this research and of the objectivity evident in the thesis' analysis.

#### 1.3.1 The role of *a*-rational forces in design outcomes

Three observations summarise the impasse. Firstly, the author has repeatedly observed that the design of software architecture by professional software engineers and architects (or the quality of software architecture) is driven more by an individual designer or design leader than by individual or collective adherence to externalised methods, processes or frameworks. This observation appears on face value to contradict the rational foundation of software design methodology which, over the past thirty years, has maintained that design method can be comprehensively separated from those who enact it. This observation raises questions about the existence of an alternate paradigm, and this thesis introduces philosophical and design-theoretic points of view on what an alternative design paradigm might be. If rationalism has indeed failed to embed comprehensively in the minds and practices of the professional designer class, the reasons are likely to be



multifarious and in need of research effort to be stated with clarity.

Secondly, in many cases (in the author’s experience) software design outcomes are shaped as much by factors and forces in the immediate design situation as they are by objective constraints (such as problem requirements, platform or product constraints). This second observation suggests that ‘human’ factors have as much weight in the design of quality software architecture as do methodological or technological ones, even though the software engineering discipline has repeatedly attempted to automate or mechanise design. Again, just what these factors are is one of a number of potent questions addressed by this research.

Thirdly, the author’s experience suggests that the actual act of designing quality software architecture or components does not exclusively rely on the designer following a publicly visible process of decomposition, transformation or synthesis. Rather, it is equivalently influenced by the interactions of designers and stakeholders in a given situation. This is true for design at all levels of abstraction from product to architecture, component to class. These observations imply that ‘soft’ or subjective factors such as personalities and capabilities are significant factors in design success, and that attempts to codify universal design methods from conceptual design down to detailed design may only ever be capable of explaining *some* of the designer’s actions and motivations.

These three observations of practical software design outcomes are collectively grouped under the moniker ‘*a*-rational forces’—that is, forces that shape design outcomes that are *not* rational and are not primarily rooted in or related to rationalism. The term *a*-rational is definitely not intended to imply irrationality! As the thesis progresses, it will become clear that this overarching term can describe actions that appear to have logical and even causal justifications within discourses, situations, domains of understanding and communities of practice.

### 1.3.2 Apportionment of responsibility

The prospect of explaining the *a*-rational behaviours of practicing software architects hints at the main motivation for this work. In the history of software engineering, rationalism has been revered as good and anything else *not* good. Software project disasters are frequently attributed to the failure of people, not technology, and often to the failed project’s inability to execute a rational development process or method. In the nineteen eighties, Norman (1988) and the early protagonists of human-centred design attacked the assumption that a user’s confusion and disorientation when faced with an everyday office

machine such as a photocopier was the user's (and not the machine designer's) fault. In similar fashion, a case can be made that users are not always to blame when software design methods appear irrelevant to the designer or the design task at hand. This argument is a specialisation of a larger one represented by the broad movement away from the use of plans and abstract representations of the world as predictors or instigators of action. Proponents of this view hold that plans are 'naive and dangerous fantasies of control and order' that impose 'inappropriate, expensive and disruptive interventions in all sectors of working life' (Johnston 1999, p. 146). Those who work with or under these systems, or who are involved in attempting to implement them, often end up being dubious of their worth. But in many cases, any form of argument against systems based on conservative or rational planning regimes is characterised as 'user resistance' or even recalcitrance, rather than the articulation of real flaws in the underlying plan-based approach. The philosophical basis for such claims, and the alternative models they offer, are reviewed in detail in this thesis. The argument that software design is not purely an objective and rational process opens a void that must be filled—if every part of it is not rational, then what are the *a*-rational parts? This question echoes the primary motivation for this thesis.

This research explains and justifies the behaviour of software architects in terms other than those forced on most observers by the incumbent rational perspective. An alternative perspective will allow the impartial and open-minded observer to draw new interpretations of a design history, outcome or scenario. Put simply, it will excuse software architects from judgement on the basis of their *a*-rational behaviour. It may, in some cases, allow redefinition of what a rational assessment would otherwise define as failure. In summary, the primary underlying motivation for this research is to emancipate the designer from the shackles of the incumbent rationalistic software design paradigm—without attempting a revolutionary paradigmatic overthrow or even diluting its value. The philosophical and theoretical implications of this motivating goal are dealt with in the next three chapters.

### 1.3.3 Recognition of the significance of situation and situatedness

'Situation' is a critical concept in this work. Its incorporation as a central research theme is supported by the general theories of 'situatedness' (situated cognition and action) which are explored in Chapters Two and Four. The decision to centralise situation in this work allows the researcher to take a constructivist philosophical stance and to adopt a holistic, encompassing approach to the subjects. This orientation positions the work in opposition to rationalism, which opens many challenges at each phase of the research. The reward, however, is the assembly of a situated model of software design practice and relief from the

tension that can arise from improperly fitting rational models.

This tension is not unique to software design. The tension between objective methodology and creative design lies at the core of all design, and various design disciplines can be observed dealing with it in different ways. For example, architecture prefers to emancipate its designers from construction and material constraints, preferring them to adopt the role of free-spirited artist. At the other extreme, engineering necessarily locks its designers closely to material constraints and in so doing excludes creativity. The fact that software design (even in the abstractness of software architecture) encompasses both extremes has led to uncertainty of role and identity for the software architect and designer. In situations where this uncertainty surfaces other forces may be freed to dictate design outcomes. This study seeks to extricate these forces in an unambiguous and useful fashion, thereby explaining why a purely rational view of design is not universally applicable in the software domain.

## 1.4 Research Aim

The research aim is as follows:

*This research aims to explain how experienced software architects and designers understand, reflect on, and describe the ways that they draw upon rational methods, past experience, contextual factors and other inputs to design enduring object or component-based software architectures in industry and business contexts.*

Exploding this statement into its phrases allows the underlying motivations to be described.

*This research aims to explain how experienced software architects and designers* —the subject of the research is the practitioner, rather than the software architecture itself, or the team. This choice is justified in Chapter Five. The participants will be selected to have solid architecture, leadership, design and development experience in object-oriented and component based software technologies. Participants will need to show they are experienced in designing within problem and solution domains that present levels and dimensions of complexity that necessitate architectural solutions and non-trivial software architectures, and that they have maintained and evolved their designs over time.

*...understand, reflect on, and describe the ways* —going to the designers rather than to their designs means that the study’s raw data will be the designer’s accounts and perceptions of their design experiences. This is markedly different data to that which would be derived

from analytical studies of their software designs. The difference is intentional and follows from this study's focus on the practitioner's self-reported experience of designing *in situ*. Further consideration of research design issues (such as matters of interpretation) can be found in Chapter Five.

*... that they draw upon rational methods*, —the study will gather data on how software architects use rational methods and techniques. This will reinforce previous work on how programmers and teams use methodologies, such as (Carroll 2000; Parnas and Clements 1986; Roberts Jr. et al. 1998; Walz et al. 1993) and many others.

*...past experience*, —designers draw heavily upon their knowledge of past solutions, and the compliance of software architects in this practice is best evidenced by their embracing of software design patterns over the last decade. It is likely that software architects will report consciously and subconsciously drawing upon architectural and design patterns when designing. This research will explain why and how.

*...contextual factors*, —as posited in the discussion of motivations, this study attempts to understand some of the ways that contextual factors shape software architectures, in what ways context drives 'situated' software design, how it differs from a rational design perspective, and how rational and situated modes of designer behaviour can alternate or coexist.

*...and other inputs* —there may be other inputs to designing that emerge from the interactions with the cohort of architects. These will be analysed as they emerge.

*...to design* —design and development can span diverse functional and structural regions within a software system. User interface design, performance design, business process and solution design are all legitimate types of system and software design, each with rich bodies of knowledge and necessitating experience and skill in practice. This study focuses on the kind of design that creates, deploys and evolves the object-oriented system or product core—the system architectures, frameworks, and components that constitute the backbone of complex business and industrial software systems. This includes conceptual design as might be performed in object-oriented analysis or object-oriented domain modelling, as well as code-level design.

*...enduring object or component-based software architectures* —the software development contexts of interest are those that have a large commitment to, and investment in object technology.

*...in industry and business contexts* —there are dozens of contexts in which object technologies are used. The aims of this investigation will be best served by involving

architects who have experience over the lifetime of software systems or artefacts, and where attention is given to the artefact's non-functional attributes such as performance, maintainability and extensibility. Industry and business are more likely to provide the environments in which this kind of expertise is both fostered and demanded. The scope of vertical domains is left relatively open but can still be used to exclude, for example, embedded systems, artificial intelligence systems, and simulation systems, to name a few.

## 1.5 Scope

The discussion of the research aim partially defines the scope of this research, however, the exploratory nature of the problem being tackled necessitates some additional clarification of scope.

### 1.5.1 Relationship to software engineering

The study's relationship with software engineering requires clarification. Most of the time, software design and development is grounded in software engineering theory and anchored to trusted software engineering principles. This study does not dispute the value, applicability or suitability of established modelling, analysis, decomposition, design, coding, or testing theory or techniques. Recognising these as foundations of practice, this study focuses on why and how the architect uses these theories and techniques in practice. The dogma evident in rationalism (and to some degree in software engineering) that this study challenges is that which proposes methods and processes as controlling or dictating the design of software architecture. The distinction becomes clearer with an analogy.

The traditional craftsman designs and builds continuously and indistinguishably, using tools at hand and in response to his vision of the finished artefact and the characteristics of the particular materials being used. No externalised process or flow-chart guides his actions, neither is he instructed or forced to use his tools in a prescribed manner. Being entirely self-directing in response to vision and situation, he plots his own path, responding to his evolving goals and to cues from his immediate environment. Although his actions flow from tacit know-how, it is simplistic to think that the form of this knowledge is rigid, structured, or able to be expressed independently of situation. It is the equivalent designer's behaviours in the software fabric that this study seeks to explicate, analyse and describe. Software engineering contributes tools, technologies and overarching methods but it does not furnish the software architect with a prescriptive process or template for the act of designing. The goals and motivations of software designers in the act of design

remain as uncharted, private, tacit and notionally unpredictable as they do for designers who work in most other fabrics and disciplines. The relationship between the act of design and methodology is addressed further as another point of scope.

### 1.5.2 Centrality of the designer

Other design disciplines focus as much on the designer as on the designs. This can be seen most prominently in architecture, where both the architect and their buildings are analysed and criticised, and if judged to be suitably exemplary, both designer and the design (through successive re-interpretations by architectural historians) contribute to the theoretical discourse of architectural design. Some architects contribute without ever having had their designs built (Thackara 1986). It is not the intention of this research to promote the erection of platforms from which software architects can assume the vainglorious personas of their built-world architect cousins. Rather, there is a case to be made that extending our understanding of software designers will complement our understanding of their designs. To pursue this goal, this research deliberately focuses on the expert software designer, and uses the picture that results as a platform to interpret software design situations and outcomes.

### 1.5.3 The designer's skills profile

The subject of this research is individual designer's accounts of how they design object-oriented software architectures. Prime candidate technologies include C++, Java and .NET. The design of software architectures in non object-oriented and non component-oriented technologies, such as COBOL or Microsoft's Sequel (an SQL-based business language) are excluded. Object-based (rather than object-oriented) technologies which support classes, instances (objects) and encapsulation but not inheritance, polymorphism and dynamic binding are also excluded from the project's scope, because it is through the use of dynamic binding that many widely acknowledged architectural mechanisms are enabled (Booch 1994). For example, all of Gamma et. al's (1995) design patterns rely upon dynamic binding to provide architectural flexibility and extensibility. Object-based technologies do not provide sufficient structuring mechanisms to support the accumulation of experience considered relevant to this study.

### 1.5.4 Types of architecture

Also excluded is the design of non-software architectures, despite the argument that certain kinds of architectures are closely related to object-oriented software architectures.

Examples of excluded architecture types include system architectures (which arrange hosts, network services, gateways, routers, firewalls and proxy servers into a topology), business process architectures (despite their expression in formal process modelling notations such as BPMN (bpmn.org 2007) and business architectures, which model business units, resource flows and dependencies. Solution architecture, which typically amalgamates all of the above into a business-focussed roadmap, is also excluded, as is product architecture, which resolves the relationships between a current software product line, new technologies and new requirements into a roadmap of features and releases. Constraining the research scope to object-oriented software architecture manages the study's scope, ensures that the participating architects are talking about the same things, allows generalisations to be drawn, and preserves the relevance of the research's findings to the discipline of object-oriented software engineering.

#### 1.5.5 Relationship between architecting and methodology

Variants of the question 'how do software designers design' have motivated some prior research. A good example is the question of how software developers use methods. Work in the period immediately following wide-scale adoption of structured methods revealed that experienced designers did not follow methods slavishly—rather, they 'cherry-picked' techniques, approaches, tools and heuristic knowledge, modifying and combining them, based upon their experience. This finding has now been verified in many research contexts (Carroll 2000; Dekleva 1992; Dietrich et al. 1997; Fitzgerald 1997; Jayaratna 1994; Khushalani et al. 1994; Parnas and Clements 1986). This thesis is not primarily about how software architects use methods, although their reflections on methods, archetypes and patterns will emerge. Rather, it delivers a holistic description of the context in which they perform design, where methods are one of many available resources. This thesis' argument assumes from the outset that software architects do not design by *following* methods—rather, they control and set their own path using resources at hand for the situation they find themselves in. Methodologies such as the Rational Unified Process (Kruchten 2000) or its contemporaries or predecessors inform and resource but do not drive or dictate the designer's path. It is the common themes in the designer's accounts of how they 'control and set their own path' that this thesis seeks to describe.

#### 1.5.6 Relationship between architecting and coding

Past research on software design practice has predominantly focussed on how programmers use methods to assist them in making code-level design decisions. Little research has been

done that focuses on architects as distinct from programmers, perhaps for some of the reasons claimed earlier. The issue of whether the research findings on programmers equally apply to software architects may depend on how much programming is inherent in architecting. This topic will be explored with the participants, but the assumption will be made that most architects of object-oriented systems will be proficient coders able to move flexibly between architecture and code-level design tasks. The research on programmers will therefore be considered applicable to architects. Programmers who have not had substantial object-oriented architecting and design experience will not be selected for participation.

### 1.5.7 Relationship to Agile methods

During the period in which this research was conducted (1999-2005) Agile methods emerged. Agile methods (AgileAlliance 2005; Cockburn 2002; Highsmith 2002)—particularly eXtreme Programming (Beck 2000)—claim to address some of the problems of inflexibility and unresponsiveness of traditional software methods. Agile methods recommend a highly iterative, discovery-based ‘emergent’ approach to software design and development. It is unclear whether the recent availability of Agile methods has substantially changed the practice of software design. For a start, it is reasonable to expect that the accumulation of experience of published methods would take a number of years of continuous use. Also, the degree to which Agile approaches to design differ from those of established methods (such as prototyping (Agresti 1986), Rapid Application Development (Martin 1991) and iterative object-oriented design (Booch 1994)) is difficult to determine with any accuracy. The participant’s degree of familiarity with, and commitment to Agile methods will be probed in the field-work phase of this study. Participants with experience of using Agile methods will provide a source of data on the effectiveness of the methods which should go some way to understanding the significance of these questions and the nature of the impact that Agile methods have had on practitioners.

## 1.6 Hypothesis

In parallel with pursuing the research aim stated earlier, this thesis also posits an argument, and it is that *software design can only be meaningfully understood when viewed as situated action*. The argument doubles as a hypothesis for descriptive research of this type in that it takes an early position on what the research will discover. When the research has satisfied the aim (by being descriptive of expert practice) it will also have built evidence for or against the



argument (that the design of software architecture is essentially situated). This statement of the hypothesis (the thesis' central argument) represents something of a forward reference, since the hypothesis arises from the review of background theory and literature (Chapters Two to Four). However, as the hypothesis is a prime motivation of the work, it is brought forward and stated here. It remains then for the argument to be either put with satisfactory evidence or to be modified or further qualified in the light of the evidence from the research's findings. The hypothesis is tested (the argument is evaluated) in the Conclusion (Chapter Ten).

It is important to notice that the hypothesis is not intended to represent an *either-or* choice—this thesis does *not* set out to *replace* a rational model of software design with a situated one. Rather, it argues for a plurality of models of design and sets about depicting the situated model in more detail than has previously been attempted. As a result, it is hoped that the reader will appreciate the respective merits of the situated and rational models, where each is applicable and relevant, and how they relate to each other. The application of this research is therefore to improve understanding of the somewhat mystical act of architectural software design by an appreciation of these two designer's modes of action.

## 1.7 Overview of the study

The graphical map of this thesis (Figure 1) uses a pyramid to convey progression from the research aim at the pinnacle, through the successive layers of the background chapters, the research design, case study and results chapters, to the discussion and conclusion chapters. The shaded clouds identify epistemological domains from which existing theory is drawn. The unshaded clouds represent exploratory research activities. The thesis chapters are shown with dependencies back to the activity or phase that motivates the content. The following narrative summarises the purpose of each phase and highlights how they collectively lead from research aim to research conclusion.

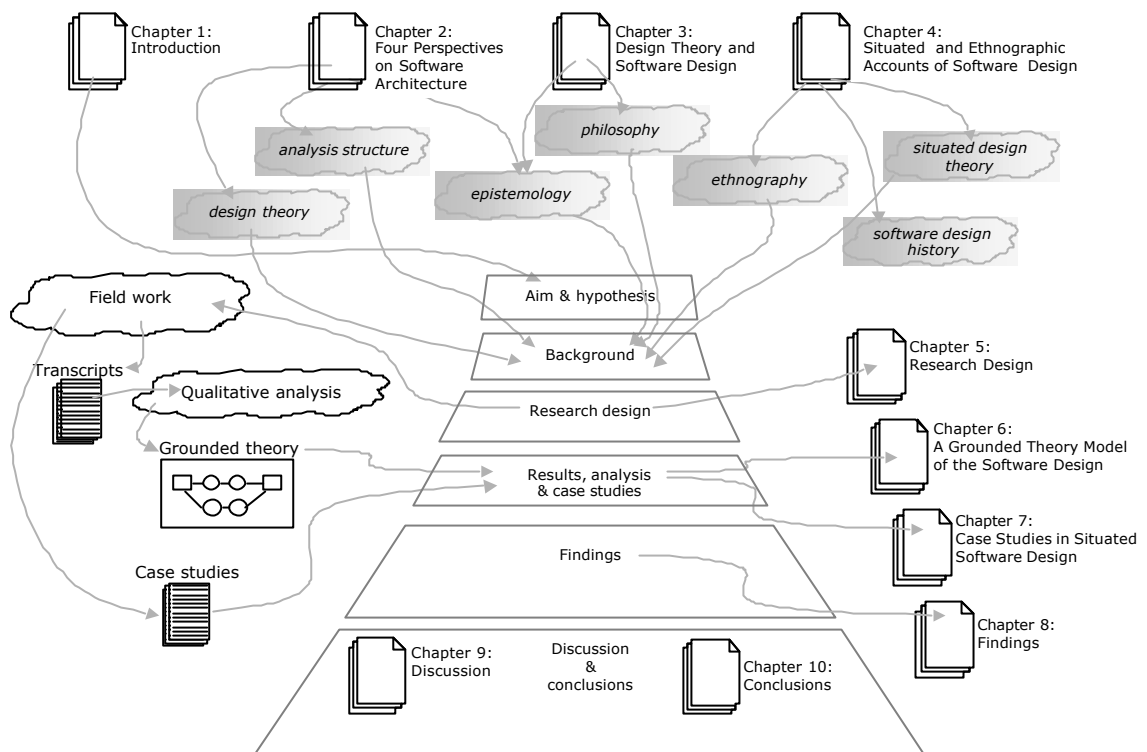


Figure 1: A map of the thesis.

### 1.7.1 Background

As stated, the thesis draws on three broad disciplines—design theory, philosophy and software engineering. Design theory yields themes of planning, vernacularism, structured versus organic design processes, generators, design control and others. Introduced in Chapters Two to Four, these themes are reinforced with historical accounts in non-software domains such as built-world architecture, town planning and cognition. The thesis moves forward by drawing together the findings from these diverse design disciplines and domains, applying them to the design of software architecture, and using them to set a framework for how the effects of situation can be assessed in accounts of design practice. The first half of the thesis (Chapters Two to Four) invites the reader to open up some familiar and some possibly unfamiliar areas, in the interests of recognising common themes. The second half of the thesis (Chapters Five to Ten) tests these themes for their applicability and relevance to the practice of software architecture design.

### 1.7.2 Research design

The source of data about how software architects perceive design and designing is the architects themselves. The options for the selection of a research paradigm and the design

of the research's methodology are dealt with in Chapter Five. This research uses structured interviews with software architects and qualitative analysis of the interview texts as its means of collecting and analysing data respectively. Chapter Five explains the choices of research paradigm, defines a research method, and deals with methodological issues of consistency, rigour and reliability.

### 1.7.3 Field work and analysis

Structured interviews with expert software architects contribute accounts, narratives, heuristic knowledge, personal concepts and reflections on design practice. As the interviews proceed, qualitative analysis of the emergent themes is performed to identify significant themes and to steer the conduct of the interviews. After completion of the interviews a qualitative analysis of the data delivers grounded theory assertions that identify drivers, forces and motivations in the context of software architecture practice and the designer's personal accounts of design practice (Chapter Six). Two accounts of designing are assembled into a chapter of case studies (Chapter Seven).

### 1.7.4 Findings, discussion and conclusions

The grounded theory assertions and the case study results are brought together as a set of findings and collectively tied back to the study's aim (Chapter Eight). The findings are then related back to theory in the discussion (Chapter Nine) which allows the formation of additional conclusions. Some additional questions about design practice arising from Chapter Two ('Four perspectives on software architecture') are also addressed in the discussion. Chapter Ten draws together the research's conclusions, including a response to the hypothesis, some generated hypotheses, responses to some of the key themes of the research, and a statement of the originality and significance of the findings.

## 1.8 Conclusion

Rationalism is the incumbent software design paradigm and provides the underlying philosophical base for software engineering (Truex et al. 2000). Software engineering furnishes software architects with paradigms, models, tools, techniques and methods but does not adequately motivate or explain the creative and personal activity of the design of software architecture. The act of design—the point at which a software architect puts a design where none previously existed—remains one of personal creativity and appears to be influenced as much by situation as by objective constraints and requirements.

Professionalisation of the software architect's role over more than three decades has reinforced the expectation that software design professionals should rely upon methods and strive for repeatability. However, research has consistently found that designers and developers do not use methods but prefer to hand-craft their personal design processes using an apparently *ad hoc* collection of tools and techniques. Few detailed descriptions exist that explain why and how expert software architects approach design in this way.

Rationalists would attribute the failure of software engineering to seriously influence the practice of software architecture design to the practitioner's disinterest or inability to understand, use, internalise and pass on formal or prescriptive design methods in their practice. However, professional software architects continue to show little appetite for submitting to formal methods or prescriptive processes. This tension is not new—it has existed for several generations of software developers and appears to be a significant motivation of the Agile methods movement.

The rational paradigm will continue to provide a foundation for software construction but it does not support or adequately explain the personal act of designing software. Alternative models are needed. This study explores designer's accounts of practice in order to describe such a model. This is not an attempt to replace existing software methods such as the Rational Unified Process (Kruchten 2000) or Agile methods (AgileAlliance 2005; Cockburn 2002) but instead supplements these methodologies with a socio-cultural layer, allowing more holistic interpretations of design practice and design outcomes.

The research will be exploratory and descriptive (rather than causal or explanatory) in that it will deliver a 'rich picture' of a practice of software design from an unconventional philosophical perspective. The project's aim—to describe and account for the practice of the design of software architecture—will be satisfied by this emergent theory, case study evidence, and rigorous comparison with the literature.

## Chapter 2: Four Perspectives on Software Architecture

The only principle that does not inhibit progress is: *anything goes* (Feyerabend 1993, p. 14).

### 2.1 Introduction

Design, design theory, and the observation of design practice are complex and amorphous domains, and to describe expert design practice, some organisation of both theory and practice is required. Clegg's (1994) organisational framework posits four theories (rational, pragmatic, radical and critical) of the purpose and meaning of technology design in society. This chapter explains these, and goes on to explore the meaning of each of these theories for software design practice. These perspectives are consistently used throughout the remainder of the thesis as a structuring device, particularly in regard to structuring the cross-disciplinary thematic literature review (Chapters Three and Four).

### 2.2 Handles on Design

At this early point in a thesis it is customary to state formal definitions of key terms. But a formal definition of design, or even software design, might pose more questions that it would resolve. Design is a fundamental human behaviour. Design is 'one of the most elusive yet fascinating topics in the software field', Glass (1999, p. 104) suggests—elusive because it defies rational objectification, and fascinating because it holds the key to the success of most software projects. Dym (1995) chooses a metaphor to illustrate his conception of design—his 'design onion' identifies four layers—experiential (design is about learning by doing); mathematical (it only achieves valid mathematical status when framed in mathematical terms); cognitive (it is about the functioning of the human mind in

creative activity) and social (it is about process activities, especially in team settings). These layers each represent distinct bodies of knowledge, and the difficulty in stating the foundations of design explains one of the recurring problems faced by theorists who attempt to unify design methods or express design in universal frameworks. The layers make ‘strange bed-fellows’, Glass writes (p. 104). While depicting design as a mixer of theories and practices from divergent disciplines is illustrative, Dym’s metaphorical onion, like many metaphors, suffers from over-generalisation.

Other metaphors feature prominently as a mechanism for typifying design. For example, bricolage (Louridas 1999) is construction, using whatever comes to hand, by a process of trial and error, assembling, tinkering or playing. Jencks’ (1973) case for purposeful *ad hoc* design makes similar claims about the situational serendipity that appears to drive the design process at times. The bricolage metaphor illustrates how the designer approaches the design task—whereas the engineer seeks direction from the immutable laws of the universe, the bricoleur seeks direction from his collection. Where the engineer creates the means for the completion of his work, the bricoleur redefines the means that he already has. While the engineer plans, the bricoleur assembles. The bricoleur’s results are always unpredictable and may diverge from the original intentions, but the argument that this redefinition does not happen for the engineer engaged in a planned form of design may at times be equally difficult to make. Metaphors such as these provide stimulating input to the ongoing discourse of what design is. Louridas’ claim on ‘bricolage’ as a universal design metaphor ultimately highlights the author’s postmodern perspective and points to the need for pluralism in viewpoints of design.

### 2.2.1 Four definitive sociological paradigms

The inadequacies of metaphors and the absence of universal, useful definitions makes a framework like Clegg’s (1994) model of organisational behaviour applicable to systems and project-based design. The original source is Burrell and Morgan (1979) who classified industrial sociologies by drawing two orthogonal dimensions, one concerned with the nature of society (ontology) and the other with the nature of our knowledge of the world (epistemology). The ontological scale ranges from concern for order and stability to concern for conflict and change. The epistemological scale ranges from the objective to the subjective. Hirschheim and Klein (1989) took this elementary framework and decorated it with analyst roles (Figure 2), illustrating the software designer nicely.

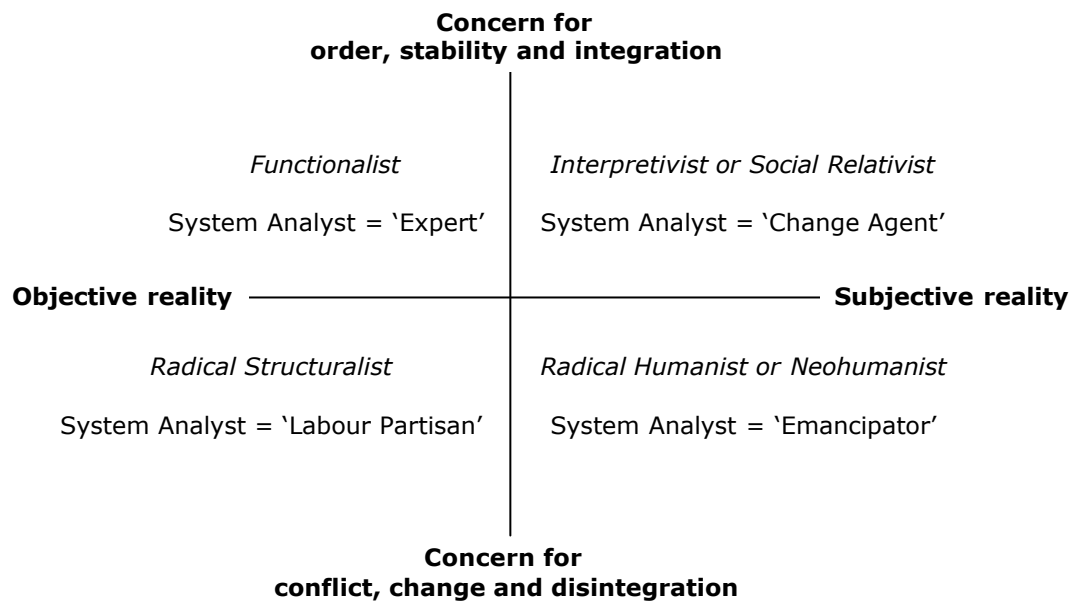


Figure 2: Four sociological paradigms for organisational analysis.

Starting at the top left quadrant, Hirschheim and Klein define systems analysts (or researchers or practitioners) concerned with regulation, continuity and order as functionalists who work within rationalistic methods to transform organisations toward an objective truth. The majority of system and software design methods fall within this category. At the top right quadrant, the interpretivist analyst works as a change agent, facilitating people to discover and own their needs and their environment. Checkland's (1981; 1990) Soft Systems Methodology fits this quadrant. In the bottom left quadrant, the radical structuralist analyst accepts the objective nature of the world but attempts to effect radical change, inevitably encountering conflicting issues of labour versus capital. In the bottom right quadrant, the radical humanist analyst is also concerned with change, but emphasises the subjective nature of reality—they typically attempt to use information technology as a tool to emancipate its users.

Coyne's (1995) four-part categorisation of information technology in society is substantially similar. Coyne uses Gallagher's categories of philosophical thought as the basis of four views of information technology systems in society. These views—conservative, pragmatic, critical and radical—provide us with useful ways of thinking about design (Gallagher 1991). Coyne's first view—that of data, information and knowledge as commodities that can be stored, made to flow through channels, and magnified over time, constitutes a *conservative* characterisation of information technology, in that it suggests that a commodity must be created, apportioned and conserved. This view

provides impetus for the creation of information technology systems that provide for faster and more ubiquitous transmission, more efficient storage, better access, and smarter ways of generating more information. In this conservative conception, design means forming and extending the infrastructure subject to the constraints of conserving value. People feature in this view as actors in the activities of provision and conservation. Coyne illustrates this category by associating well-known computer scientists. Herbert Simon (1985), for example, (and others of the artificial intelligence community who seek to reify human knowledge directly in machine form, particularly in the symbolist school) are conservative, he claims. In the software methods domain, methods with an engineering orientation such as the Rational Unified Process (Kruchten 2000) may be similarly branded.

A second characterisation views the importance of information technology as its ability to become a tool, or an extension of a person's reach or capability. In Heidegger's (1962) conceptualisation of 'being', a tool is an inert object until its actor begins to use it, at which point it becomes a part of the actor and he a part of it. Because the tool metaphor places the person at the centre and focuses attention on the engagement between human and technology, Coyne calls this the *pragmatic* perspective. Coyne argues that theoretical views of computer systems design are giving way to the pragmatic in many ways. Some confirmation of this may be found in the emergence of Agile methods (Cockburn 2002).

A third characterisation of information technology hinges on the notion that its importance derives from its role in society's political and social web of control. Information technology serves as a medium of global communication that cannot be regulated, is prone to abuse, and promotes ways of thinking that some in society would wish to oppose. Coyne terms this characterisation as *critical*. From the *critical* perspective, design means unbalancing and re-balancing structures of social and political control, subject to the net forces in the particular context. A critical perspective has value in understanding the non-technological dimensions of systems development where economic, political and sociological drivers frequently outweigh technological ones. In the software methods domain, the self-regulating open-source communities and developer collectives (such as Raymond's 'bazaar' (1999)) are arguably critical, in that they invert conservative notions of ownership and control.

A fourth characterisation of information technology in society focuses on the difficulty of defining a solution that addresses needs in a complex, dynamic, evolving environment. In ultimately self-deprecating fashion, this position holds that any ideas that we develop



about the centrality of information technology turn out to demonstrate the opposite, and any sweeping change that we may ascribe to the introduction of information technology into our world has already been usurped by something pre-existing. Coyne names this characterisation *radical*—it takes what purports to be a progressive position and demonstrates the orthodoxy in this position. Design from this perspective involves ‘deconstructing’ prevailing attitudes and canonical positions to resolve new constructions of ‘being’—our perceptions of how we exist in relation to others, and to the objects around us. Coyne names Marshall McLuhan (1964; 1967) as a radical who was prepared to argue along the lines of media subverting messages (McLuhan’s contributions to design theory are summarised in the next chapter). In the software methods domain, Gabriel’s (2002) early attempts at method deconstruction based on Feyerabend’s theory of counter-intuition are radical, because they deliberately subvert orthodox methods in the interests of enlightenment.

These four views provide useful vantage points from which software design practice may be examined. They are general enough to apply broadly and they build upon a philosophical base. They also account for rational (conservative) design behaviours on par with pragmatic, critical and radical ones—a characteristic that many method assessment frameworks cannot claim.

<b>Paradigm Characteristic</b>	<b>Rational</b>	<b>Constructivist</b>
Cognition	Follows a ‘data processing’ metaphor—cognition is evidenced by the deterministic processing of known inputs to produce outputs.	‘Emergent’—cognition is an emergent property of the interaction between an individual and the environment.
Behaviour	Behaviour is largely predetermined by plans and the deterministic application of known skills and resources to problems.	Behaviour is largely in response to contextual situations, observations, events and interactions.
Epistemology	Knowledge is embodied in process and procedure. Observations are generalised to theories that converge toward universal laws. Independent of context or situation.	Knowledge is embodied in people’s conscious and sub-conscious, and results from their continuous experience of interacting with others and the environment.
Learning	Programmed, as a result of the periodic evaluation of plans and processes.	Continuous, as a result of interacting with each other and the environment.

Table 1: Comparison between rational and constructivist philosophies.

### 2.2.2 About definitions and theory

The philosophical frameworks of Table 1 hint at why the quest for a ‘handle on design’ (a definition) will continue to be elusive. Coyne and others suggest that no such definitions are possible until the observer’s philosophical stance has been declared, and there appears to be merit in being able to move between different philosophical perspectives. To understand how the observer’s philosophical stance changes the definition of design, and to build towards a research question, the implications of Coyne’s four perspectives on software design are explored commencing with rationalism.

## 2.3 Rationalism

The history of software engineering reveals repeated attempts to express software design as a set of rational theories. Rationalistic approaches have undoubtedly been highly successful in delivering the mechanisms of software development and production that today’s global software industry relies upon. Rationalism falls within Coyne’s *conservative* perspective, in which design is an intervention in an otherwise predictable world intended to convert a less desirable situation into a more desirable one. It assumes that the designer can declare needs and intentions and relate these to objects in a meaningful way, and that technological artefacts conserve the meaning and intent of their designers and creators over time, despite changing circumstances. In conservative design, new artefacts are the products of individuals and teams who modify, combine, synthesise or reproduce pre-existing objects. Designers (rather than constructors or users) control what they produce, and this control finds its expression in methodology. Methods are typically sequences of techniques, steps, activities and phases that prescribe generic design paths that are assumed to work in the majority of cases. Methods, like theories, gain acceptance with use over time. Conservative design relies upon certain universal principles and these are assumed to underlie all successful design.

Rationalism is not a single philosophical position (Meredith 2002). Foucault describes it as a ‘discursive practice’ we are all caught up in (Coyne 1995, p. 18) which permeates our understanding and colours our perceptions of technology and design. Rationalism promotes reason, logic and theory above the stuff of fabrication and instantiation, and it challenges pragmatism in several ways. Firstly, rationalism’s distinction between theory and practice imposes an ordering in which theory leads practice. As a result, the practical is not of primary concern, and knowledge that arises from practice is therefore degraded or unjustified. Rationalism supports the view that technology is a product of theoretical

enquiry and is subservient to it. In more extreme cases, rationalism works against participative practice by promoting a hierarchical view of knowledge that drives a wedge between the designer and users of the designed artefact.

Rationalism asserts that the physical presence of a technology is less important than what it contains or what it accomplishes—technology is a medium, but the content it conveys is what matters. Computer technologies (like minds and texts) are ‘knowledge containers’, and communication technologies (like language) are ‘knowledge conveyors’. The technological medium is less important than its message. This emphasis downplays the situation in which knowledge is created and used. In an organisation, for example, the effectiveness of a technology is a function of its efficiency and reliability in conveying knowledge, rather than a function of its ability to permeate the organisation’s web of interactions and change people’s practice or the organisation’s culture. Rationalism denies the formative and generative power of technology in its context of use.

### 2.3.1 Theory and practice

Questions of the appropriateness of rationalistic approaches to a domain of knowledge such as software design invite reflection on ways of thinking about knowledge in that domain. In many domains of scientific knowledge, theory is distinct from practice, researchers can consciously determine on which side of the line their work contributes, and questions about the relationship between theory and practice are relevant. In the software domain, however, the meaning of ‘theory’ is less clear. If posed to a group of software architects, the question ‘what are the theories of design within software engineering?’ might be expected to bring forth a grab-bag of methods, techniques and heuristic knowledge. This theory-practice ambiguity is not without precedent and is regarded by some to extend beyond the social to the physical and natural sciences. Feyerabend (1993) takes the view that much of scientific discovery is opportunistic and that all theory is incomplete or based upon generalisations that are unsupportable in all cases. Another view (Coyne 1995) is that although we attribute great technological feats (such as the Apollo moon missions) to science, it is not clear whether this attribution follows for rational reasons or because of the privileged position we bestow upon theory over practice. Philosophers such as Dewey (1916) regard theory and practice as inseparable, just as thinking and doing are inseparable—‘only by wrestling with the conditions of the problem at first hand, seeking and finding his own way out, does he think’, Dewey (p. 160) surmises.

In the scientific tradition, knowledge about natural phenomena and physical structures is

thought to be largely independent of its creators, or the setting in which the creation occurred. This ensures that the observations made and implications drawn are, and remain, independent of the observer. As long as the phenomena under study are stable, and all those involved adhere to a common framework of interpretation, this ideal can be maintained. While this holds for many design artefacts in traditional engineering, Keil-Slawik (1992) suggests that it does not hold true for the development of software. Traditional engineering focuses primarily on material structures and their physical effects, whereas in software engineering we are mainly concerned with ‘symbolic structures and their cognitive effects’ (p. 172). Two other reasons account for this difference—the dynamic nature of the relationship between the artefact and its context of use, and the uniqueness (on various levels) of the contexts of development and use. As Parnas (1985) pointed out in his treatise on the immaturity of software development practice, software in general lacks the degree of repetitiveness that is so characteristic of materials or artefacts in other engineering disciplines. Thus theory of software engineering is compromised in a classical sense by the nature of the software fabric.

### 2.3.2 Software as a design medium

Discussions of design process repeatability inevitably raise questions of the appropriateness of the engineering metaphor. When Lammers (1986) asked a number of eminent computer programmers the question ‘Is computer science a Science?’ she got various answers, including ‘I don’t know what truth computer science is trying to learn’ (p. 55), ‘I definitely consider computer (programming) to be an art’ (p. 201), and ‘I call (computer science) a craft because is certainly isn’t a science yet’ (p. 216). These commentators are not the only ones to borrow terminology of a bygone era. Goguen (1992) thinks that software engineering is presently ‘more like a medieval craft’ than a modern engineering discipline (p. 200). Shaw and Garlan (1996) describe the design and development of software architecture as being traditionally based on a substantial folklore of system design, with little consistency or precision. While many observers regard engineering as an attainable but distant goal for the discipline of programming, some openly regard it as a poorly chosen metaphor (Borenstein 1991), and pursue more radical metaphors such as craftsmanship (McBreen 2002) and theatre (Laurel 1993). The perception of software architecture as folkloric practice may be the result of a rationalistic bias on the part of the observers as per Foucault’s ‘discursive practice’.

The need for continuity of design effort throughout the software artefact’s lifecycle also challenges a rationalistic conception of software design. At the time when software

theorists were converging on the waterfall model, design theorists in non-software domains had moved on towards models of design as discourse and as a continuous process. Mitchell (1988) perceived fractures in the alignment between traditional (engineering) design and manufacture and software design and development. Mitchell viewed design as ‘a continuous and non-instrumental thought process, a creative act in which everyone, designers and non-designers alike, may participate equally’. The designer’s role in the post-mechanical era, he argued, was to make the design process equally accessible to everyone. To realise this goal, Mitchell demanded involvement—design must ‘become a socially oriented process in which... we are all both spectators and actors’ (p. 214).

That software design is continuous during system development is a tenet of object-oriented lifecycle models (Booch 1994; Meyer 1988). The focus of this continuous design effort is the software architecture (Foote and Opdyke 1995), its components, collaborating classes, and the mechanisms within individual classes. The software medium itself affords far greater opportunity for redesign than most others—in software, there is nothing to stop a programmer from propagating a highly localised bug fix at the implementation level into the abstract classes that comprise the system’s architecture. Or—for that matter—from a domain object forward through presentation layer classes into the system’s user interface, or backward into the architecture’s persistence layer. In this kind of evolutionary software development, we can never be sure where production processes (such as coding and bug-fixing) end and architectural re-design begins. In fact, design can occur on every level of abstraction and scale concurrently, and even localised development can trigger redesign on an architectural scale (Foote 2000). Mitchell’s and Foote’s continuous design are fuelled by designer-initiated interaction with all parts of the system and its context. The enlightenment that followed from experience with object-oriented software architectures has translated into the partial disintegration of phase boundaries. The elongated feedback loops typical of waterfall lifecycle models have collapsed and software integration and release cycles have dropped from months to weeks.

While these characteristics of object technology allow a relaxing of rigidity in process models, they do not address other aspects of the rationalistic orientation in lifecycle models. An awareness of the limitations of traditional rationalistic methods continued to be raised by both computer scientists (Budgen 1995; Floyd 1992b; Winograd and Flores 1986) and information systems researchers (Galliers 1992; Hirschheim 2001; Redmond-Pyle 1996). Planning is one important aspect that has a long history in design theory.

### 2.3.3 The Failure of the Master-plan

One of the most enduring influences of rationalism on all forms of design concerns the area of planning. Master-planning constrains design into a distinct phase, and forces premature resolution of design options on the basis of knowledge of requirements and constraints at a given point in time. A brief digression into architectural history illustrates these principles. Two forms of architectural master-planning—the civic master-plan and the public housing estate—left unmistakable legacies that demand reinterpretation in every generation. Adherence to the principles of modernism amongst architects and planners began to falter with the mediocre results of the British New Towns policy, which culminated in 1972 with the inauguration of Milton Keynes (Frampton 1988). Although it is now widely recognised that in city planning, the boundary between the design of physical structures and the design of social systems dissolves almost completely (Simon 1985), the modernists took a supercilious attitude, neglecting the daily, practical needs of inhabitants. Le Corbusier and his contemporaries adopted a theory of zoning—by carefully re-ordering the city, by separating its functions, they thought it possible to solve a wide range of social ills (Myerson 1993). By the time the celebrated revolution in popular culture of the sixties arrived, architecture's infatuation with Le Corbusier's egocentric vision had been noticed more broadly. Journalists such as Jacobs (1964) reported on the discrepancy between the diversity and simple richness of life on Manhattan's multicultural street corners and the lack of life in New York's new Garden City developments. Architectural theorist Christopher Alexander crystallised these observations in a pattern-based design manifesto based on a philosophy of user-centricity (1979), a backlash against modernism (Grabow 1983). The master-planned cities of Milton Keynes and Levittown (Gans 1967; Venturi 1970) were the last attempts by architects to project utopian urban schemes on a grand scale.

Levittown became a notable case study for planning theorists. Levittown is what we would today call a housing estate on the outskirts of Pennsylvania. It is distinguished by the fact that it was one of the first such estates to be planned, built and sold by a single developer, Abraham Levitt and Sons, and as such, one of the first master-planned domestic environments. The Levitts devised a mass production scheme that allowed them to build inexpensive housing for the postwar flood of veterans and their families. Architects, planners and sociologists have since iconised Levittown as the first and archetypal American suburban estate, maligning it for its emphasis on self-containment, lack of community facility, social dis-integration and expressionless uniformity. Levittown,

initially a blueprint for the suburbanisation of middle class America, was upheld as primary evidence of the thesis that quality, diversity and richness of the experience of life is dictated by one's surroundings (Gans 1967).

Levittown left another imprint on design thinking at the end of the modernist era. The houses, each being relatively plain and indistinguishable as a result of the Levitt's drive for economy, began to sport various forms of symbolic décor. Occupiers built post-and-rail fences reminiscent of southern ranches, nailed old wagon wheels to feature walls, put coach lights either side of the door, and erected flagpoles. The architectural cognoscenti were delirious with ridicule—such kitsch appliqué further reinforced the general public's lack of taste and provided proof of the meaninglessness of such symbolic appliqué. To the modernists, decoration was crime—symbolism of any kind on buildings or artefacts represented backwards-looking self-indulgent romanticism, recalled a discredited period of design, and failed to embrace Internationalism or the functional purity of the new materials. Venturi (1970; 1977), one of the first postmodern theorists, reinterpreted these symbolic decor attachments. Far from being kitsch architectural detritus that the modernist movement had attempted to sweep away, these mass-produced do-it-yourself appliqués were legitimate symbols of working and middle-class aspirations that were entirely consistent with emerging postmodernism. Venturi considered Levittown to be 'almost alright' in the sense that it represented a kind of aesthetic-in-opposition to modernism's constant desire to drive bulldozers through socially coherent communities. The Levittown residents, landlocked by the constraints of a modernist master-plan, broke free by articulating their environment with architectural appliqué, Venturi claimed. More recently, Brand (1994) has documented the same human propensity to indulge in constant modification in his exploration of how buildings 'learn'.

Design and development of all kinds in isolation or without necessary interaction is at best sterile and at worst dangerous. With regard to the modernist housing estate genre, the Pruitt-Igoe housing project in St. Louis, Missouri, provides an oft-cited case study in late-modernist apartment design (Mitchell 1988). Upon completion, the project won design awards from the American Institute of Architects which assessed the project as being an exemplar for future low cost housing projects. But the high-rise design was soon considered uninhabitable by its residents—parents could not supervise children in the ground level playground, the facilities were inadequate, and the building's segregation inhibited natural social flows and broke the resident's established social relationships, allowing crime and vandalism to flourish. Just sixteen years after the initial flurry of

accolades, the complex was demolished with the inhabitant's blessing.

There were other famous failures. When Le Corbusier finally managed to realise his design for a 'unité jardin verticale', it was the communal facilities that remained unused. The utopia of the preconceived concrete and steel forms could not be filled with life, not only because of the architect's 'hopeless underestimation of the diversity, complexity and variability of modern aspects of life' but also because modern societies with their functional interdependencies 'go beyond the dimensions of living conditions, which could be gauged by the planner with his imagination' (Habermas 1997, p. 232). These episodes from architectural history serve to illustrate important reasons why master-planned design was discredited. It is not coincidental that the history of planning in software engineering lifecycle models—from waterfall to whirlpool—reveals a similar progression. Pruitt-Igoue cameos the destiny of design performed independently of consideration of the needs of inhabitants or users. A masterful idea on the architect's drawing board and acclaimed as a concept, it failed dismally in actuality because the designers were seduced by the prospect of realising a utopian town planning theory and ignored habitation. The Levittowners sought ways of articulating their environment because the master-planners had left it featureless and expressionless. The utopia of the British new towns faded as a result of its designer's failure to anticipate the social fabric of a community in advance. In general, a designer working alone at a point in time cannot anticipate the needs of a structure's inhabitants. Modernism's propensity to freeze all design considerations allowed the achievement of a form of structural and functional purity but eventually became its fatal flaw.

Modernism (and by implication rationalism) was condemned for other reasons. Thackara (1988) was concerned with the detachment of the design process from construction processes. Salingaros (2006) developed mathematical models of 'architectural temperature' (an abstraction of scale) that illustrate modernism's semiotic inconsistencies. Frampton (1988) criticised modernist themes in planning as lacking sensitivity and relevance. There are important lessons for all designers to be found in these episodes from architectural history. Communities have rich and complex social dimensions that cannot be planned *a priori*. People have an innate need to express themselves—to articulate their environments. To misunderstand these lessons is to risk repeating the mistakes of the modernists. These themes are equally evident in the history of software lifecycle process models. The 'rich and complex' dimension of community life justifies the pursuit of collaborative system design approaches, and software lifecycle models have incorporated



iteration as a result. Iteration serves to reduce feedback loops and increases development response times (Busby 1998) but does not address all of the concerns of those researching systems design in complex environments. Alternative planning models to modernism's master-planning are needed that acknowledge contextual factors and adequately allow for the rich and complex forces of situation.

#### 2.3.4 Alternatives to the Master-plan

From the failures of the modern era, design theorists moved on to autopoiesis, organic and evolutionary models, where design is thought of as emergent rather than being projected or prescriptive. This period evidenced a new awareness of the complexities of the social systems in which design is performed. There were three recurring themes. Firstly, modernism's strict separation of design from construction and habitation was perceived to have resulted in less habitable structures, as Pruitt-Igoe illustrated. Participative design approaches, as advocated by Mitchell (1988) and others, addressed this concern. Secondly, there was an increasing recognition of the cost of modernism's disregard for history and context. The link between architecture and social history became fashionable, and historic precincts in cities began to be reconceived as museums of economic, social and cultural evolution in what was to become a global urban renewal movement. The emergence of the postmodern movement embodied an attempt to reconnect people with symbolism and form in buildings and artefacts. Thirdly, modernism's grandiose attempt to design for all people for all time was widely perceived to have failed (Hubbard 1996). Town and master-planning swung away from enforcement of social and economic zones of control toward more integrated and inclusive theories and approaches.

Alternatives to master-planning reconceived the goals of design from product to process. Simon (1985) expressed these ideas when he observed that a paradoxical, but realistic view of design goals is that their function is to motivate activity, which in turn generates new goals. Each step of a project's implementation creates a new situation, and the new situation provides a starting point for fresh design activity. The notion of finality, Simon claimed, is inconsistent with our limited ability to foretell or determine the future—what *we* call final goals are in fact criteria for choosing the initial conditions that we will leave to our successors. Social planning has much in common with the processes of biological evolution, Simon claimed, in that it exhibits a form of myopia—we as designers in our time and generation prefer not to see too far. Looking a short distance ahead, we try to generate a future that is a little better than the present. In doing so, we create a new situation in which the process can then be repeated. In the theory of evolution, Simon concluded,

there are no theorems that direct this myopic hill climbing.

Alexander and other theorists of the seventies wrote with the voice of a generation of disaffected inhabitants of their predecessor's modernist monuments. His alternative to master-planning is succinctly put in 'A City is not a Tree' (1988) which opens with the clarification that 'the tree of my title is not green with leaves... it is an abstract structure' (p. 67). Alexander's tree is not the one that underlies hierarchical master-plans or any manifestation of functional decomposition—rather, it is a metaphor for bureaucratic control of the commercial, social and domestic functions of a city. 'A City is not a Tree' proposes an emergent behavioural paradigm for planning and design that attempts to replace established processes of decomposition (the kinds of process that produce a tree) with those of devolved evolution (alternate processes that produce a network) (Grabow 1983). Alexander's approach to complex, multi-dimensional planning problems commences with a cameo of habitation:

In Berkeley, at the corner of Hearst and Euclid, there is a drugstore, and outside the drugstore a traffic light. In the entrance to the drugstore there is a newsrack where the day's papers are displayed. When the light is red, people who are waiting to cross the street stand idly by the light; and since they have nothing to do, they look at the papers displayed on the newsrack which they can see from where they stand. Some of them just read the headlines, others actually buy a paper while they wait. This effect makes the newsrack and the traffic light interactive; the newsrack, the newspapers on it, the money going from people's pockets to the dime slot, the people who stop at the light and read papers, the traffic light, the electric impulses which make the lights change, and the sidewalk which the people stand on form a system — they all work together. (Alexander 1988, p. 68)

This vignette reveals static and dynamic elements. The static part is the physical arrangement of newsrack, lights and sidewalk—a structural pattern within the system which Alexander calls a 'unit' of a city. This particular unit is inhabited by a dynamic flow of people and objects (newspapers, coins, electrical impulses). A unit is bounded by the dynamic system it supports, and contains only those physical objects that are required to make the dynamic system work. Units can exist on a much larger scale than the one involving a street corner, such as the arrangements of facilities that support dynamic systems for shopping, eating, socialising, working and entertaining.

In natural cities, units and the dynamic systems they support overlap, Alexander claims. The news rack may be a part of an adjacent unit that supports the system of people waiting at a nearby bus stop. Alexander claims that this kind of overlap is emergent, rich, and complex—often too complex to be designed *a priori*, and certainly too difficult for

conventional town planning methods to anticipate. When units overlap, they form what Alexander calls a sub-lattice structure. When units are non-overlapping (disjoint), they form a tree. Alexander's network model of emergent design is behavioural rather than static or structural. Similar emphases (that drive the design of system structure based upon dynamic behaviours or scenarios) have found their way into object-oriented software design methods (Jacobson 1992; Taylor 2001d; Wirfs-Brock 1993). To avoid a tree of units in which the interactions between places, objects and people are artificially simplified or inappropriately constrained, network organisational structures are needed. Alexander's (1977) pattern language is the most concrete realisation of an alternate theory of design that defines apparently viable decentralised design processes that drive the evolution of system structures.

## **2.4 Pragmatism**

Under Coyne's (1995) pragmatic theme, meaning and intent are neither pre-existent nor absolute, and designing cannot be done without adopting the interpretive norms of one's community and making grounded judgements from within that context. Pragmatic design is more about projecting expectations than it is about addressing needs. Community replaces appointed roles, pre-existing artefacts become objects for reinterpretation, and the act of designing is *performed* more than it is planned. Situation dictates to the design act and design evaluation more than theories, universal principles or methods. Pragmatic design is an exploration in which needs are discovered as design progresses. Pragmatic designers are caught up in a world of artefacts and practices, and their history. The pragmatic designer works to explicate designs, releasing them from the situation. At all times the design and its justifications are visible and within reach. This contrasts with the conservative, heroic designer, whose individual designerly powers are more likely to be tacit, implicit or even closely guarded.

Pragmatism overthrows the Greek philosophers' theory-practice dualism. By preferring contemplation over action, abstraction over exemplar, mind over body, means over end, subject over object, they talked about science but did not practice it, Dewey (1958) claims. This duality began to dissolve during the scientific revolution of the sixteenth and seventeenth centuries as the rapidly developing natural sciences combined and interchanged ideas from both intellectual and technological realms. In this period, the production of tools served to illuminate and enlarge the significance of other objects and events, and the discovery of new theory became dependent on tools. Hickman (1992),

Dewey's biographer, wrote that 'theory became a tool of practice and practice a means to the production of new effects... theory had no longer to do with final certainty but instead, as working hypothesis, with the tentative and unresolved' (p. 99). Dewey regarded facts, ideas and concepts all as tools.

Because tools become effective only in use, pragmatism relies strongly on locality or situation, and tool use recalls the recurring theme of the relationship between planning and action. In artificial intelligence, researchers moved from models of *a priori* planning to contextual planning when the former proved to be too limiting (Clancey 1993). Suchman (1987) addressed the foundational ideas of this transition by advocating a recognition of action as being understandable and explainable only in its immediate context, coining the term 'situated action' to distinguish this view from the entrenched one of action following and conforming to a plan, description or specification. Consideration of the impact of situation previously had been confined to language (Searle 1969). Suchman's depiction of action is consistent with Feyerabend's (1993) conviction that scientific discovery—and in fact all action—emerges from amidst a 'maze of interactions' and that 'successful participation in a process of this kind is possible only for a ruthless opportunist who is not tied to any particular philosophy and who adopts whatever procedure seems to fit the occasion' (p. 10). Situated cognition also found expression in Lave's (1988) research on adult's everyday use of mathematics in familiar settings such as supermarkets. Lave was concerned that cognitive investigation of mind, body, activity or setting alone was an oversimplification, and was influential in expanding the focus of cognitive research from the experimenter's subject to the 'whole person in action, acting with the settings of that activity' (p. 17). Situated cognition subsequently influenced social theories of learning, finding expression in, for example, social theories of 'communities of practice' (Lave and Wenger 1991).

#### 2.4.1 Situated Action

Situated action is an important theoretical underpinning of Coyne's pragmatic perspective. Its definition is couched in pragmatic terms. Suchman claims 'the term *situated action*... underscores the view that every course of action depends in essential ways upon its material and social circumstances'. Rather than to attempt to abstract away action from its circumstances and represent it as a rational plan, the situated approach studies 'how people use their circumstances to achieve intelligent action' (p. 50). Rather than build a theory of action out of a theory of plans, the situated approach aims to investigate how people produce and find evidence for plans in the midst of their situated actions. Suchman's

ethnographic account of islander navigators illustrates the two extremities of action perspective. Alternative views of human intelligence and directed action can be typified by contrasting the European and Trukese approaches to navigation. (The Truk Islands are centrally located amidst the Carolines in the Federated States of Micronesia). The European navigator begins with a plan—a course—which he has charted according to certain universal principles. He carries out his voyage by relating his every move to that plan. His effort throughout the voyage is directed to remaining ‘on course’. If unexpected events occur, he returns to his plan, and designs and executes a corrective action. The Micronesian navigator begins with an objective rather than a plan. He sets off toward the objective and responds to conditions as they arise in an *ad hoc* fashion. He utilises information provided by the wind, the waves, the tide and current, the fauna, the stars (without actually having explicitly mapped them) and he steers accordingly. If asked, he can point to his objective at any moment but he cannot describe his course. Vast migrations of peoples in the eleventh and twelfth centuries over huge distances occurred, particularly amongst Pacific islanders, using just these methods of navigation.

The European navigator exemplifies the prevailing software engineering models of purposive action—models motivated by the perceived need for transparency and predictability. While the Micronesian navigator would be hard pressed to tell us how he actually steers his course, the comparable account for the European seems to be transparent, in the form of the plan that is assumed to guide his actions. While the Micronesian navigator’s actual course is contingent upon unique circumstances that cannot be anticipated in advance, the European navigator’s path is derived from universal principles of navigation and is essentially independent of the exigencies of his particular situation.

Suchman’s story was based on ethnographic work performed by Hutchins (1983) in which the reality and effectiveness of the navigational methods of the islanders were validated. The story illustrates the essence of Suchman’s model of situated action, and carries a number of important implications for planning, action and design. Firstly, different ways of acting are favoured differently amongst cultures—how to act purposefully is at least partially learned and is subject to cultural variation. European culture favours abstract, analytic thinking where the ideal is to reason from general principles to particular instances. The Micronesians, however, learn a cumulative collection of concrete, embodied responses and are guided by memory and experience accrued over years of actual voyages. Suchman argues that all activity is ‘fundamentally concrete and embodied’ (p. viii), that our

responses to situations (i.e. our actions) are cultural and conditioned, and that these attitudes deeply divide both our ways of thinking about acting and the actions themselves.

A second interpretation of the navigation story is that our actions are either *ad hoc* or planned depending on the nature of the activity or the actor's expertise. People have navigated in an *ad hoc* fashion in certain of the world's oceans for centuries, but the design and construction of a civil engineering project, for example, is an activity that must proceed with considerable investment in detailed planning. If individual actions do not align with this plan, then they must be (or must be made to be) inconsequential. It is common to associate planning with goal-directed activities and *ad hocism* with expressive, creative acts. Planning, then, equates with mature action as typified by engineering, whereas *ad hocism* equates with unconstrained acts of the creative free will such as artistic endeavours (Jencks and Silver 1973).

It can be concluded that both the Micronesians and the Europeans use instruments and work to plans—it is just that the instruments are of a different kind and the plans have different representations. All actions, according to Suchman, are situated, and although we like to think of ourselves as drawing on the European tradition of cognition and planning, in reality we all act like Micronesian navigators most of the time. The argument for situated action hinges on uncertainty—circumstances cannot always be anticipated ahead of time and are constantly changing, so we *must* act in a reactive fashion to some degree. Our actions as they are observed to play out are never planned in the strong sense in which we expect planning to lead action. In some forms of design, plans are best viewed as a weak resource for what is primarily a reactive and *ad hoc* series of decision-making episodes.

Suchman did not deny the usefulness of plans, but claimed that plans should be considered *a priori* improvisations and *post hoc* reconstructions of action (Bardram 1997)—in other words, that there is no strong cause and effect relationship between plans and action (Johnston 1999). It turns out that in many situations, plans are used for more than just forecasting action. It is often only when we are pressured to account for the rationality of our actions, given this 'European tradition' that we invoke the omniscience of a plan, or retrospectively and selectively reconstruct our situated actions so that they conform to an imagined or recovered plan. This behaviour—not unknown in software development projects—is referred to as *post hoc* rationalisation (Crellin et al. 1990; Parnas and Clements 1986). When stated in advance, plans are necessarily vague, in so far as they must accommodate the unforeseeable contingencies of particular outcomes and situations.

When retrospectively reconstructed, plans systematically filter out detail that characterises situated actions in favour of detail and actions that appear to align with the plan (Kotre 1995).

The navigation story also illuminates our assumption that representations of actions (such as plans) can in fact be the basis for an account of situated action at all. Culturally, acceptance of a causal relationship between planning and action is almost universal, so much so that to question it risks treading on belief rather than fact. This belief is rooted in scientific positivism (Weatherall 1979) and, of course, rationalistic thinking. Acceptance of the situated model of action necessitates careful consideration of where software ‘planning’—in particular software architecture, software abstraction and design—sits on the figurative line between the European and Micronesian traditions of planning and action.

#### 2.4.2 Design, Context and Culture

The pragmatic conception of design also elevates collaboration and culture as significant factors in design. To adopt a pragmatic stance in design is to engage with and interpret context. In systems design, this means setting down a design path based on discovery rather than *a priori* formulation or projection. Mechanisms of discovery include user involvement, iteration and continuous gross and fine-grained feedback at all levels of the software artefact’s lifecycle. Pragmatic design infers willingness on the part of the designer to defer certain structural and functional concerns. The designer’s openness to interpretation and the emergent design that results draws attention to the role of context and culture in the design process.

Cultural theorists characterise contemporary cultures in ethnographic terms. Marshall McLuhan’s (1964; 1967) ‘media philosophy’ earned him considerable kudos during the transition of social orders in the mid-1960s, and his influence is evident in the design of personal computer interfaces. Software technology expedited the information revolution by providing ubiquitous communications, an electronic office on every desktop. Instantaneous global electronic communication now enables a new culture in which people find new freedoms and encounter altered power structures. Design in McLuhan’s global village has changed its nature—not only do the village’s inhabitants demand and expect continuous incremental improvements in the products, systems and services they use, but they look to design for a new degree of involvement, a new kind of dialogue. Coyne’s argument that rationalism is giving way to pragmatism in design finds support in

McLuhan's predictions. To McLuhan, the printed page as a medium promotes a hierarchical and bureaucratic model of human knowledge in which those who control content exert power over others. In stark contrast, the information age with its global, real-time communication channels promote a liberal and empowering view of knowledge. This age promises the re-emergence of a 'tribal culture' amidst the collapse of the world's formerly separate bureaucracies into a 'global village', characterised by immediacy, involvement and a strong sense of social cohesion. In the electronic era, 'action and reaction occur almost at the same time' (McLuhan 1964, p. 4).

McLuhan is most famous for his mantra 'the medium is the message', later modified to 'the medium is the *massage*' (McLuhan and Fiore 1967). Ignoring the postmodern pun for a moment, the aphorism best captures his thesis that the historical points at which a new medium (such as print or electronic channels) is adopted corresponds with significant shifts in both culture and distribution of power. Media revolution takes place when a new medium sweeps away the previous one but the content does not substantially change, hence the medium *is* the message. The pun embedded in McLuhan's book title ironically twists the concept by suggesting that the medium pummels, kneads and manipulates its subjects, affecting them more than the content does. Since the emergence of electronic media, governments, communicators, advertisers and designers have consciously used and abused the undeniable effect of media on message for their own political, commercial and social purposes.

Of Coyne's classifications of information technology in society, McLuhan's concept of medium usurping message is radical, while his advocacy of community involvement and ownership clearly favours pragmatic over rationalistic design. McLuhan's early awareness of the emergence of the global village, the collective consciousness and its 'total social processes' (1964, p. 358) led him to modishly declare that 'Heidegger surf-boards along on the electronic wave as triumphantly as Descartes rode the mechanical wave' (1962, p. 248). This declaration of philosopher-of-choice recognised the magnitude of the shift occurring in world culture—Heidegger's concern for 'being' and his treatment of objects in a contemporary context made him attractive as a philosopher flag-bearer for the information age. Others concerned with software and systems would subsequently favour the philosophy of Heidegger for related reasons (Winograd and Flores 1986). The discourse shared by culture and design is mutually definitive. Culture contextualises and constrains design, and enables pragmatic design action. Design constitutes one of the primary forces capable of shaping and diverting culture when its artefacts are subsumed



into the experience of existence. As software mobilises from the computer on the desk to computers in the pocket and in the jacket sleeve, the speed with which it influences culture will increase exponentially.

### 2.4.3 Design and History

Because pragmatic design is interpretive, understanding the design of an artefact or system necessitates reconstructing or replaying its history. Feyerabend advocates a view of methodology that integrates history. Feyerabend's theory of counter-intuition advocates a pluralistic methodology in which a designer adopts many perspectives, constantly compares and contrasts ideas, and attempts to improve failed ideas rather than discarding them—all in the interests of leaving options open so that truth can be discovered rather than being prematurely shut off.

As McLuhan's prophecies of a global electronic village are fulfilled more accurately than even he could have imagined, a primary theme is that of control. Successive waves of technology reorient culture in ways that are profound but at times subtle, not to mention unpredictable. In a presentation in 1980 (in Melbourne) on the impact of microcomputers on society, Barry Jones claimed that 'the difficulty is that changes [to society as a result of the introduction of microcomputers] are likely to have made their impact before we can make an adequate assessment' (Jones 1980, p. 95). 'Today, Jones' prediction seems mundane because we now accept that technological impact on society is unpredictable and irreversible. Our best option for understanding is to reconstruct events as best we can, given our limited visibility. Plans cannot be more than snapshots that reveal intentions and projections of the designers at a point in time, whereas the history of an artefact or system and its use serves to explain the opportunism and arbitrariness of discovery and design.

Implications for theory and scientific method of the 'opportunism and arbitrariness of discovery and design' are espoused by Feyerabend, who claimed that 'science is an essentially anarchic enterprise, and that theoretical anarchism is more humanitarian and more likely to encourage progress than its law-and-order alternatives' (Feyerabend 1993, p. 9). Quoting Lenin, Feyerabend warned that history is always richer in content, more varied, more many-sided, more lively and more subtle than even the best methodologist can imagine. History is full of accidents and conjectures, curious juxtapositions of events, and it demonstrates to those who care to look 'the unpredictable character of the ultimate consequences of any given act or decision of men'. Are we really to believe, Feyerabend argues, that the 'naïve and simple-minded rules of which methodologists take as their guide

are capable of accounting for a such a maze of interactions?’ (p. 9).

Feyerabend’s references to ‘method’ refer both to the scientific method, and to the methodological principles that underpin domain-specific methods, such as software design methods. Feyerabend bases his deconstruction of rationalistic method partly on an examination of the history of science. Barely a single rule or principle is not violated at some time or other, he claims. Far from an embarrassment or a result of insufficient knowledge or inattention to detail, such violations turn out to be necessary fulcrums upon which new discoveries are made. It is when researchers and pragmatists decide not to follow scientific dogma or when they unwittingly break the canon that serendipities unfold. Given any rule, however fundamental or rational, Feyerabend claims, ‘there are always circumstances when it is advisable not only to ignore the rule, but to adopt its opposite’ (p. 14), citing the alternate wave/particle theories of light and other examples.

Feyerabend suggested a universal maxim of his own. It is clear, he claims, that the idea of a fixed method, or of a fixed theory of rationality, rests on too naïve a view of man and his social surroundings. When we look to the rich material provided by history, we must avoid the temptation to abstract and simplify it for purposes of condensing it into a tell-able size, or cherry-picking each scenario so as to fit a pre-existing rationalistic orientation. We must resist our learned instincts to reduce the complexity of reality into elements of canonical objectivity, or instances of familiar theory. If we honestly do this, Feyerabend implores, there is only one principle that can be defended under all circumstances, in all human creational activities, and in all stages of human development... ‘the only principle that does not inhibit progress is: *anything goes*’ (p. 14). Feyerabend’s anarchistic approach is not malicious—rather, he demonstrates (with examples) ‘how easy it is to lead people by the nose in a rational way’ and that ‘all methodologies, even the most obvious ones, have their limits’ (p. 17).

Feyerabend may be appealing for a weaker form of anarchy than his mantra might initially suggest. Firstly, he is not advocating ignorance of theory or conventional wisdom, because he suggests a strategy of ‘counter-induction’ in which he urges us to intelligently formulate hypotheses that are counter-inductive to established theories and established facts. This requires us to know the theories and facts in the first place. He advocates a pluralistic methodology, in which a scientist adopts many perspectives, compares his ideas with other ideas, and attempts to improve failed ideas rather than discarding them. He focuses us on contrasting rather than aligning our constructs with extant theory in order to generate alternatives, on destabilising stultifying assumptions, and on stimulating thought.

Feyerabend's program is the injection of stimulant into the lethargic veins of the scientific method. Nothing is ever settled and no view is ever cast in concrete. An idea, the theory it begat and its history must all be preserved together, because such preservation leaves the theory open to further development, and possible reinterpretation in new contexts.

The implications for design concern models of knowledge and the role and form of methods. For software design, Feyerabend's position on method leads to an injection of history into descriptions of what was designed, rather than some variant of *post hoc* rationalisation in which system and project outcomes are reinterpreted to fit preconceived ideas of rationalistic design. He recommends replacement of systematic accounts of knowledge with a historical account of each stage of knowledge, in which discoveries or developments are expressed as narratives. He promotes a re-conceptualisation of design methods so that they encourage unhindered generation of alternatives rather than technical adherence. Beyond this, he advocates the generation and exploration of alternatives that counter accepted wisdom (counter-induction), on the basis that 'there is no idea, however ancient and absurd, that is not capable of improving our knowledge' (p. 33). He advocates improving rather than discarding theories, ideas or designs that initially appear to be unstable or unsuitable in the interests of leaving options open. Feyerabend's message challenges software engineering's rationalistic orientation—most notably objectivism—and its reliance on method (Gabriel 2002).

#### 2.4.4 Pragmatism versus bounded rationality

Coyne's pragmatism appears to share some characteristics with Simon's (1983) notion of bounded rationality. Simon recognised that a rational decision-making process is in most cases unachievable and that the likelihood of making an 'optimal' choice is remote. Instead, what successful decision makers generally do is to select an option that, rather than being the best possible, is the best available. In choosing options that are simply *good enough*, decision makers act rationally within the bounds placed upon them by their environment and limited cognitive capacities, hence Simon's term 'bounded rationality'. The bounded-rational actor's decisions 'satisfice' (Simon's term) rather than optimise. The theory of bounded rationality gives up some of the elegant formal properties of the rational choice model to produce a theory of decision making that is far more descriptively accurate (Meredith 2002). For the practicing software architect, bounded rationality accounts for decision-making in the presence of incomplete data but it does not account for the replacement of a rationalistic and objective relationship with requirements, constraints, plans and models with a constructivist or situated one. Because it does not challenge the

underlying rationalistic philosophical stance, bounded rationality explains decision-making in the presence of noise and inaccuracies but does nothing to elaborate constructivist models of action.

## **2.5 The Situated Software Architect**

Feyerabend argues that pluralism must drive design methods and that action should challenge and extend an idea's history. Coyne contends that design (and action) must be informed by multiple (postmodern) perspectives. McLuhan reasons that design has no meaning outside of context, and that action, culture and medium are inseparable. Suchman argues that all action is essentially situated—of the moment—and conventional plans as controllers of action are meaningless. These assertions challenge incumbent rationalistic conceptions of design. In pragmatic terms, they pose a problem—the determination of how useful or otherwise these theories are in explaining how software design is practiced. To pursue this line of enquiry, we need a closer examination of software design practice.

Expert software architects face complex, dynamic and unique design problems that require them to work collaboratively, flexibly and skilfully. No single design model or framework is likely to be able to explain all of their approaches and behaviours comprehensively. Coyne's four design themes—conservative, pragmatic, critical and radical—provide useful platforms from which to construct descriptions of software design practice. For the remainder of this chapter, each of these perspectives is adopted with respect to the software architect. These descriptions are further illustrated with some rhetorical questions. These are not intended to be research questions, aims, goals or hypotheses—rather, they should be read as a mechanism for clarifying the scope of each of the four perspectives. Responses to each of these questions are made at the end of the thesis (Chapter Nine).

### **2.5.1 The Rational Software Architect**

Rationalism points to an orientation on the part of the designer. The rational software architect acknowledges an external, higher source of knowledge, and follows it, to the degree possible. Departures are regarded as flirting with risk. A rationalistic commitment is expressed when the architect treats a set of methods as canonical, carrying them forward and applying them to different problems in a range of contexts. Rationalism turns the architect toward external authorities for the receiving of design wisdom and new insights, and imposes limits on the degree to which personal experience can change the received

view. Rationalism casts the software architect as a highly trained and highly valuable technician, skilled in software implementation technologies, able to follow methods accurately and implicitly, and always conscious of the economic values of time and effort.

This characterisation raises important questions of how the rational technician-architect does in fact use methods to guide the design of software architecture. If the rational software architect is indeed a method-user rather than method-interpreter or method-author, the conflicts between a method's lore and the architect's personal experience need to be somehow resolved:

*How do software designers resolve tensions between the selection and application of methods and the use of pragmatic know-how, particularly in cases where these conflict?*

Particular resolutions that favour the architect's experience invoke interpretations that begin to move the architect away from strict rationalism. Interpretation of rational method and internalisation of same are, however, different things, but the architect may not be able to express the difference. Architects may express a range of views as to their perception of the value of method:

*What purposes do methods serve in the hands of experienced software architects?*

In the light of Suchman's model of situated action, and the various models of emergent design, the question of how experienced software architects have learned to most accurately estimate and plan or project their design effort is important:

*How do experienced software designers regard the use of architectural projections or plans?*

Estimating architectural design work is one side of the *a priori* planning issue—the other is the success or otherwise of producing encompassing software architectures ahead of their subsequent implementation:

*Do software architects believe that these plans should drive or shape their designs? In the cases where plans and final designs diverge, how do architects explain or rationalise these inconsistencies?*

During detailed design and development, experienced software architects must achieve an appropriate balance of planned and emergent structure. Emergent structural design may be achieved by facilitating collaborative design, by delegating to individuals, or by using a

mixture of both. The quality and coherence of the software architecture is largely dependent on how well this is done. The software architect's leadership style has the potential to most markedly impact the software architecture in this way:

*How is a balance between forward-engineered and emergent design and architecture reached?*

Seeking answers to these questions will illuminate to what degree the rational software architect survives and prospers in professional practice. But rationalism is not the only mode in which the professional software architect works, and the picture will continue to be incomplete without representations of commonplace pragmatic action.

### 2.5.2 The Pragmatic Software Architect

There are times when an expert software architect designs in a pragmatic way. The pragmatic software architect designs and constructs—at all times—in context and in collaboration with other designers and the wider community of stakeholders. To the pragmatic architect, theory and practice are imprecise distinctions that matter little. If theory is externalised, it is treated as following from practice. Reliance on plans as dictators of design effort is minimal. Methods and techniques are assessed in terms of their proven utility and community acceptance. The pragmatic software architect takes away grounded experiences and heuristic knowledge to inform future work in similar situations, whilst recognising that each new design context will necessitate new interpretations of familiar objects or techniques.

Pragmatic practitioners driven by internalised knowledge of (sometimes arcane) programming language, product or framework detail, tacit skills and time-proven solutions to pull them through project deadlines. Learning is done in master-apprentice fashion whilst working craftsman-like in software workshops rather than from theory or methods. Methods merge with fragments of work practices that 'seem to work', and design teams quickly grow their own localised practices and idiosyncratic cultures. The pragmatic designer is untrusting of structured methods or theories, and deliberately filters methodology or text-book knowledge for its immediate application in the current context. Some characteristics of this mode of designer behaviour need to be described and further elaborated:

*What characterises the pragmatic software architect?*

The elaboration of some of these characteristics will illuminate how situated design

actually occurs:

*In pragmatic, situated practice, how are designs created, or arrived at?*

There are a number of related questions that concern the understanding of the situated design act:

*What form do situatedness, opportunism and being open to contextual cues take in practice?*

Contextual and cultural factors play a role in the play of ideas and options that become software designs, and any attempt to abstract from these located scenarios will likely undo their internal consistency and relevance. It is therefore important to understand the influence of contextual factors in some form of design narrative (Wirfs-Brock 2006). Also of interest is the way that designs are transferred. Design transfer traditionally concerns the transfer of software artefacts and the transfer of the knowledge or capability that impelled the design's formation. The former will draw upon internalised software design documentation and transfer techniques such as architectural models, pseudo-code and code. The latter raises important design knowledge management issues:

*In pragmatic, situated practice, how is generalised design knowledge explicated and passed on?*

Pragmatic design unseats conventional notions of methodology as controller of design practice. A pragmatic method cannot strictly sequence activities, dictate practices based on preconditions, or impose universal laws on emerging designs. A degree of reorientation of thinking on methodology may be necessary in order for it to support this design mode:

*What does it mean for a methodology to be contextualised, to become implicit in a design—to become situated?*

Contextualisation of a method occurs in the hands of the practicing situated designer, who must learn to reflect on actions, before, amidst and after the action is completed. Individual and collective reflection can provide a source of feedback that drives method contextualisation at gross and fine levels, and can also provide invaluable feedback in the individual designer's personal learning processes. The practice of reflection can manifest in a number of ways:

*How do experienced software designers employ reflection when they design? Do they acknowledge reflection at all?*

These and related questions will help to characterise the pragmatic software architect. But

pragmatism is not the only mode in which architects work, and the picture will continue to be incomplete without some kind of critical perspective.

### 2.5.3 The Critical Software Architect

There are times when an expert software architect designs in a critical way. Experienced software architects will have worked in a number of organisations, projects, teams and business cultures, and will have experienced project failure. Such experiences may have developed a critical perspective of software development, in which methods and project structures are deconstructed to reveal underlying power structures. Critical theory asks ‘who is in control?’, and targets the pervasive structures that exist to preserve the *status quo*. Once revealed, the appropriate responses are perpetual scepticism, emancipation or revolution.

A critical assessment of a situation leads to a discernment of the distribution of power, to the revelation of power structures, the oppressors and the oppressed. Design is, by its nature, the process of altering the world, and the design of software and computer systems has the potential to re-distribute power both during the development period and after the system change has been commissioned. The ways in which expert software architects consider these issues in order to effect software acceptance, deployment and use are largely unknown, as are the ways in which they choose to deal with these forces in their designs and design outcomes to their stakeholders.

Although the software architect might not need to be the controller of all things technological, the architect should know where control lies. The critical view of methods and project structures as devices for assertion of power between competing factions inevitably emerges in any examination of software development organisations and teams:

*Is there evidence to support the assertion that software architecture and design can influence or redistribute the balance of power in a design situation?*

Methods are ostensibly adopted by organisations to improve the quality and timeliness of software design. They have also been adopted as a means of reducing development risk, and in a critical light, as a means of distributing risk:

*What purposes do methods serve in the context of organisations and the competing parties engaged in economic enterprise?*

A designer’s brief is often vague in regard to the responsibility for how the artefact or product is ultimately used. This is a different concern to that of useability, which deals



with how useable the software product will be in the hands of users. A designer's sense of responsibility for use concerns how the product may change its environment. For example, a designer may design a tool without concern for how it is used, or may go further to consider in what contexts the tool will be most used and how it might change these contexts from a social, cultural and humanitarian perspective. A software designer's lack of concern for how a product is used implies the rationalistic separation of object from subject, of tool from human user:

*Does the software architect perceive responsibility for how the product or system will be used, or for the implications of its use on people, regardless of their role or position?*

A related issue is to what degree software designers take responsibility for recognising and intervening in inequities, as are often found in environments of sweeping change on the back of technological change, social engineering or workplace reform:

*Do software architects regard themselves or their designs as resolvers of inequities?*

In some situations, software architects may be tempted to subvert authority in the interests of product completeness or quality. The ways that tensions between parties in software development contexts are resolved (or ignored) are revealing of the power structures. Similarly revealing is the degree to which a software designer is prepared to erect facades over the true costs and activities of software and system design:

*What compels software designers to apply post hoc rationalisation in their accounts of design work performed, or to mask investment in design or architectural quality from their managers?*

These and related questions will help to characterise the critical software architect. But criticism is not the only mode in which architects work and the picture will continue to be incomplete without assessment of radical behaviour.

#### 2.5.4 The Radical Software Architect

There are times when an expert software architect designs in a radical way. Adopting a radical position can lead to new, unanticipated concepts that invert entrenched or orthodox meanings, expectations, or intentions. The ways in which expert software architects acknowledge this kind of designerly behaviour are largely unknown and substantially undocumented. The radical software architect uses re-conceptualisation to illuminate

reality and truth in various situations. Radical philosophy attempts to ‘exorcise its own rhetoric, and that of its progenitors’ (Coyne 1995, p. 103). As a philosophical domain of thought, its incessant restlessness and its proclivity to dissolve its own foundations make it difficult to pin down, although the poststructuralist writings of Derrida and the technique of deconstruction have achieved some prominence. Deconstruction reverses, inverts and demolishes opposing positions that are revealed from analysis of the texts of any intellectual enquiry.

Coyne notes that one of the responses to deconstruction is to generate ‘subversive’ discourse that challenges the foundations even as they are being built. A software designer might engage this stance by consciously subverting design orthodoxy:

*What techniques do software architects adopt in order to question or evaluate the use (and reuse) of familiar (or popularly subscribed) designs or design methods?*

This penchant to generate alternatives, or to remain open to design alternatives that have been rejected in the past, is consistent with Feyerabend’s philosophy. Feyerabend does not argue for an ongoing *investment* in a range of (rejected) design alternatives, but for a meaningful discourse as to why selections are made:

*What processes do software designers use to argue for, or build a case for a particular design solution over others?*

*Do software architects always understand how and why a particular design was chosen? If they do, can they usefully act on such insight?*

This thesis proposes that these four modes of designerly behaviour collectively constitute the ways that expert software architects design, and that to analyse a designer’s actions or interpret a designer’s behaviour from only one of these four perspectives is flawed.

## 2.6 Conclusion

This chapter has argued for a plurality of philosophical views of software design practice. Coyne’s four categories of information technology in society are viable candidates. The *conservative* platform is incumbent, is underpinned by systems theory and the scientific method, but is discredited in postmodern philosophy. Rationalistic software design methods exemplify its application. Practitioner’s understanding of (and commitment to) rationalism is open to challenge. Coyne’s *pragmatic* perspective regards design as emergent and design activity as situated, but how the pragmatic practitioner approaches design is not

entirely clear. Coyne's *critical* perspective exposes equal and unequal power relationships and has the potential to illuminate why some design initiatives and projects fail, or why design efforts are compromised. The *radical* perspective portrays designer's propensity to rethink and validate intuitions inherent in their approaches. All four perspectives will be used to structure the discussion of the findings on software design practice (Chapter Nine).

Important insights from design theory and history provide the foundation for taking a pluralist stance. In the realm of physical architecture, modernism attempted to dictate the doctrine of functionalism throughout the western world's built environment (Hubbard 1996). The attempt was derailed by modernism's underestimation of the complexity of social systems and the basic needs of people to inhabit buildings built in human scale that are amenable to piecemeal modification and extension (Brand 1994). Re-conceptions of the use of plans and designs followed, and Suchman's (1987) view of all action as being ultimately situated represented a turning point in how the activity of conceptual work (such as design) was viewed.

Progress towards a situated model of software design has been strong in some dimensions and weak in others. Object-oriented methods have introduced rapid prototyping, iterative development of architecture, and encapsulation and polymorphism that constrain the evolutionary activity behind stable interfaces and allow detailed design decisions to be deferred. Tension continues to exist between those who regard software design as a rationalistic procedure and those who see it as an emergent, situated craft. Commentators outside of software's mainstream regard software as 'not as heavily automated, bureaucratized or streamlined as, say, automobile design and manufacture', and a craft that is 'analogous to writing and publishing' (Coyne 1999, p. 28).

This chapter has provided the necessary background to the study's research aim. The next chapter will go deeper into the key themes that sit in the intersection between design theory and software design.

## Chapter 3: Design Theory and Software Design

Medieval cathedrals were certainly much more complex structurally than the pyramids, yet there is still considerable evidence that the cathedrals evolved through a process of experimentation and trial and error not unlike that of the Egyptian megaliths... architects of churches erected only forty or fifty miles apart around Paris in the late twelfth century and early thirteenth centuries must have watched and been influenced 'almost from day to day' by each other's experiments. One builder's structural and aesthetic successes and failures were challenges and lessons to the others. (Petroski 1992, p. 55)

### 3.1 Introduction

This chapter examines selected themes from the epistemological domains of design theory and software design. Design theory, to the extent that it can be identified as epistemology in its own right, is the set of models, types and languages that inform architecture, industrial design and 'design science'. The term 'theory' gets broad use in these disciplines. Coyne (1995) claims that much of what authors like to call 'theory' is not more than systematic thinking or contemplation, and that what is more commonly meant by 'design theory' is in fact 'design studies' (p. 213). Nonetheless, the research literature that falls within this intersection is considerable and surveying it is a challenging exercise in cross-disciplinary review (Blum 1996; McPhee 1996; Taylor 2001b).

This chapter commences with design's philosophical foundations and an explanation of why understanding design inevitably rests on the object-subject duality. The remainder of the chapter surveys relevant models of design and locates these in a philosophical milieu. First, categories of design models are surveyed. Next, some models of design selected for their particular relevance to software are described, including vernacular and evolutionary

models. These reviews provide the philosophical basis needed to understand the models of cognition and design based on situated and hermeneutic perspectives surveyed in the next chapter (Chapter Four). Collectively, these chapters provide the basis for the design of a research methodology (Chapter Five).

### **3.2 Philosophical Foundations of Design Theory**

Design has philosophical roots in both ancient and modern history. Some of the factors that have most influenced contemporary design thinking have their origins in the modernisation of Europe over three centuries. These complex processes of change transformed a traditional world of peasants, craftsmen, clergy and landlords into an industrial society. The various (and sometimes conflicting) ideas are commonly collected into two major intellectual threads—the Enlightenment and Romanticism. The philosophers of the Enlightenment dreamed of a world ruled by reason and governed by science and technology. Their conception of the ‘project of modernity’ concerned democratisation, secularisation, and industrialisation, all three perceived as the foundations of a rational society. As the philosophy of choice of the rising bourgeoisie class, the Enlightenment found expression in the form of various European revolutions. From these, Romanticism emerged as a rejection of the authoritarian rule of the feudal systems and the rule of the democratised Enlightened societies. Both movements expressed society’s desire to change the world for the better, but their chosen means were opposites. Whereas the Enlightenment had a clear idea of the goals of modernisation, Romanticism was non-committal, vague and open-ended. Dahlbom (1992) observes that the differences can be traced to their respective conceptions of knowledge. The Enlightenment was a goal-directed, problem-solving, cognitive enterprise in search of objective truth. Romanticism was a process-oriented, inspired, expressive movement inviting participation in bold constructions of unconstrained utopias. Where the Enlightenment exhibits a strong sense of reality, Romanticism invites exploration of the possible. The Enlightenment makes maps of observed reality, whereas Romanticism regards such maps as locked to an instant in time in a world of ephemeral experience, preferring instead to pursue the discovery (through definition) of new worlds.

In each of these contrary philosophical threads, change has very different meanings. In Enlightened society, change occurs when the incumbent rational paradigm is either further enhanced or opened up by the discovery of new evidence, or replaced by an even more rational one. Change is understood as the lawful rearrangement of unchanging elements.

The Enlightenment idea of a man-made, constructed world is the idea of applying science to engineer a desired and designed nature, society and mankind. But in Romanticism, caution is abandoned and change is regarded as the very fuel of growth. By making the observation that our world is shaped by our experience of it, Romanticist philosophers set about changing the world by changing our experience of it. If nature is constituted by our conception, it becomes possible to construct new worlds, in spiritual if not material fabric. When Kantian philosophers tied the basis of construction to culture and the fundamental elements to societal values, the focus of Romanticism shifted from the nature of reality to the social processes by which multiple realities may be constructed. This gulf between the Enlightenment's positivism and Romanticism's hermeneutics (the study of the methodological principles of interpretation) has never been bridged. Referred to by C. P. Snow as 'the two cultures' (Snow and Collini 1993) it finds expression today in the ongoing debate on the gulf between the humanities and the sciences. In post-industrial design history, Enlightenment thinking pervaded modernism and suppressed Romanticism for a period, until the latter re-emerged in the form of postmodernism. In software development, this gulf lies barely beneath the surface of the perennial question 'Is computer programming art or science?'

### 3.2.1 Aristotelian and Platonic Concepts

Aristotle (384-322 B.C.) formulated a philosophy of absolute concepts and rules and argued that all of the world's knowledge consisted of combinations of these conceptual atoms. New knowledge forms through fresh combinations of existing concepts, and any concept, no matter how complex, must decompose into basic concepts. Aristotle and Plato (428-348 B.C.) devised classification, a foundation of almost all sciences.

Every classification scheme has a basis—the selected characteristics upon which the classification rests. The history of classification in the natural sciences reveals shifts in the basis of classification from observable characteristics to habitats, breeding habits, evolutionary progressions and now DNA sequences. These shifts suggest that the nature of the classification process is incremental, iterative and grounded in historical perspective. Because all classification systems must deal with ambiguity (an object will always be found that equally belongs in two or more categories), classifiers have typically ended up choosing the basis that yields a classification that best suits their particular purpose at the time. In software design, Booch (1994) draws upon Aristotelian classification to explain how, in practice, it is often difficult to give convincing categorical definitions to even the most mundane of objects—'the identification of classes and objects is the hardest part of

object-oriented design’, he writes (p. 133). Static inheritance structures reveal the limitations of Aristotle’s absolutism outside of constrained laboratory settings. Classification works fine if the problem domain can be mapped into a small, stable set of abstractions, but not all problem domains are amenable to such analysis. Using the domain of classification as an example, Figure 3 illustrates how philosophy, paradigm, model, method and technique are related.

	<b>Software</b>	<b>the Scientific Method</b>	<b>Interpretivist research</b>
<b>Philosophy</b>	Aristotleian concepts (all reality decomposes to atomic concepts)	Aristotleian rationalism (reality exists independently of the observer)	Platonic dialogue (dialogue reveals the exemplars that define reality)
<b>Paradigm</b>	Object-orientation	Positivism	Constructivism
<b>Theory/Model</b>	Classification	The Scientific Method	Interpretivist research design
<b>Method</b>	CRC	Repeatable experimental design	Qualitative analysis
<b>Technique</b>	Specialisation versus implementation inheritance	Statistical analysis	Textual analysis

Figure 3: Relationship between philosophy, theory and models, with examples.

Plato preferred to think of concepts as being primarily defined by our practices and actions rather than as absolute things, occupying a place of their own in the world of knowledge. Concepts exist only because we exist, and concepts allow us to share a vocabulary of shared meaning. Plato viewed knowledge acquisition as the process of coming to recognise one or more of these shared concepts. Plato developed a process of dialogue along the lines of a ‘twenty-questions’ game to support classical categorisation. Is it animal, mineral or vegetable? Does it have fur or feathers? Can it fly? Ideas become concepts when two questioning minds debate, the argument and counter argument serving to bring forth a more complete and complex understanding than previously existed. Established concepts then become exemplars to assist in learning. In defining new exemplars, knowledge is transferred through dialogue.

Concepts are familiar to system and software designers as the building blocks of abstraction. Using Aristotelian concepts, we formalise our knowledge by providing rules and criteria for determining when concepts apply. Using Platonic dialectic processes, we devise proto-typical examples that may then be compared to specific observed or

experienced phenomena to assess similarities and differences. Aristotelian definitions lead to clear distinctions. Concepts in software design also appear to be Platonic in nature, particularly those that relate to doing design. When designers talk of adapting prototypical solutions, such as the way software architects use design patterns (Coplien 1996) Platonic reasoning is played out. Theorising about design moves the reflective designer away from Platonic and toward Aristotelian conceptualisation. The two forms of thinking about design are interspersed in written works that relate theory to practice. In software design, this chasm of concepts cannot be avoided, because programming languages (by their support for classification amongst other characteristics) are blatantly Aristotelian in character. It is the mapping from action-oriented imitation to the rigid categories of software structure that accounts for some of the difficulty of making software systems fit imprecise human modes of work (Jackson 1995).

The philosophies of both Aristotle and Plato are, as Dahlbom and Mathiassen (1993) put it, ‘mechanistic’, in that they attempt to define reality and all of the kinds of systems that interact with it as mechanisms that obey universal laws. In this view, all forms of knowledge and perception sit under the umbrella of universal science and all investigation of knowledge and phenomena must be subject to positivism and the scientific method (Weatherall 1979). Some theorists believe this is the direction that design theory and research must be pushed, as evidenced by attempts to formalise a discipline of design science (Simon 1985; Warfield 1994). Scientists attack a problem by trying to discover a universal rule that governs the situation, whereas designers attack a problem by a proposing a solution and observing (either in real or simulated terms) its effect (Louridas 1999). Budgen (1994) compares and contrasts the two approaches (Figure 4), concluding that the design process differs in form and intent from the processes of science. Others argue that there can never be a unifying science of design because of the irreconcilability of viewpoints in design research (Sargent 1994). Still others have abandoned the positivist paradigm altogether (Coyne 1991).



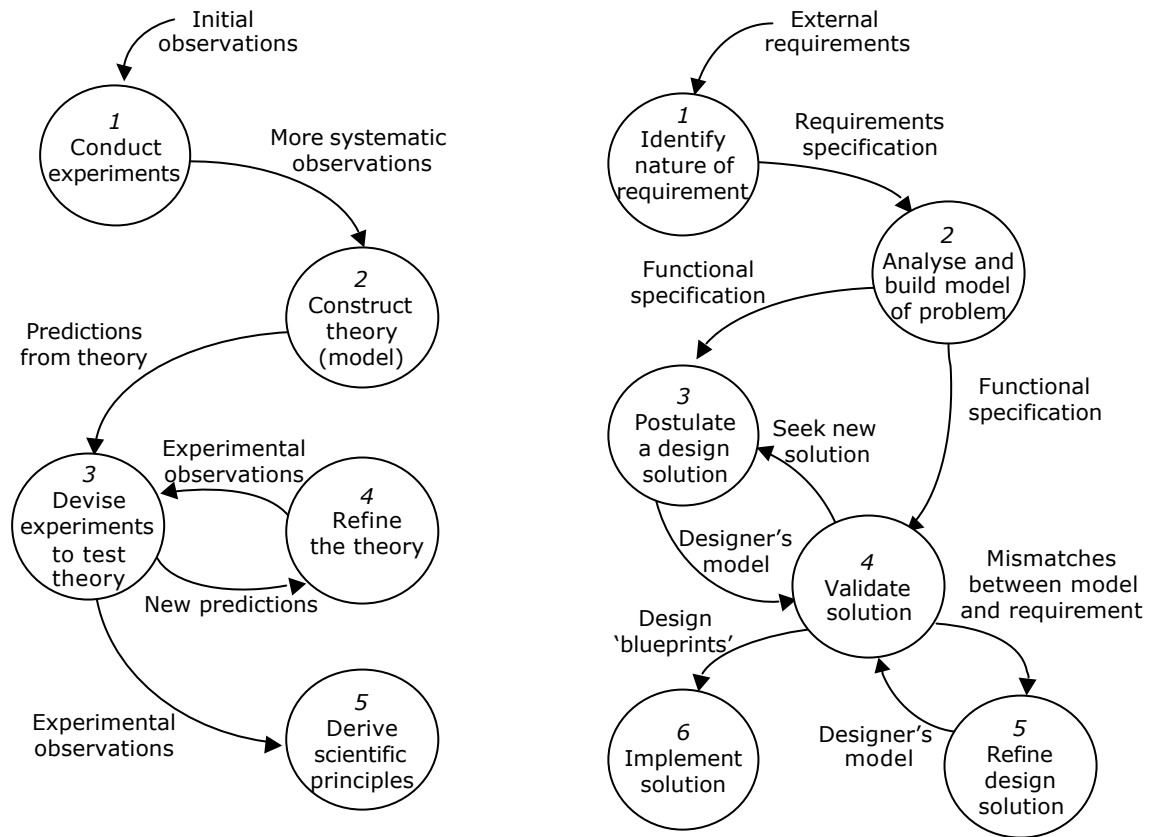


Figure 4: Comparison between the scientific analysis process (left) and the design process (right) (Figs 1.1 and 1.2 in (Budgen 1994)).

In software design, Enlightenment thinking finds expression in the rationalistic view of software development as a production process. This view has been criticised by Winograd and Flores (1986) and Floyd (1992b) amongst many others (Turner 1987). Floyd enumerates her objections to the production model as follows. Firstly, the traditional view of software development as industrial production assumes that there is a given reality 'out there', which we confront during the software development process. The essential task of the designer is to start from the problem defined in this reality and to find a correct solution to it. By analysing the facts of this reality, the analyst/designer obtains requirements for software systems. Software engineering is concerned with the production of software to stated requirements and is not primarily concerned with the separate dimension of use, she claims.

Viewing software development as production treats designers and developers as interchangeable resources in an industrial production process. The process should be independent of individuals, their interpretation, influence or style, and as a consequence, for any given problem, different developers should converge to the same results. Communication should be regulated via fixed interfaces and protocols to ensure the

division of labour on indeterminate bases. Subject to technical feasibility, any part of the production process (including design) can be automated. The developer's responsibility is the proper construction of the product in accordance with the requirements specification, and no more.

Floyd acknowledges that Enlightenment thinking such as this has been instrumental in bringing about impressive advances in programming methodology, and has allowed systems of scale and complexity to be built. But Floyd claims that it fails to offer any help in understanding the software development process actually going on in any given situation—processes relating to the emerging insights into the functionality, implementation and useability of programs and systems. The production view highlights important mechanical aspects of software development but obscures the view of development *as design*, or in Floyd's terms, as 'a specific type of insight-building process' (p. 93) that is geared to producing feasible and desirable results within a particular domain. The conception of software development as production is essentially a construction—an invention that satisfies one prevailing set of motivations at the expense of others.

### 3.2.2 Separation of Mind and Body

The break from Aristotelian and Platonic notions of absolute knowledge is attributed to Descartes (1596-1650) who shaped scientific method profoundly by separating the mind (soul substance) from the body (physical substance), thereby allowing one to be studied independently of the other (Hirschheim 2001). This allowed philosophers to distinguish between the way the world is and the way we perceive it. The mind is not just a passive machine for receiving information about the world—rather it shapes these impressions and adds interpretations:

Our ideas, rather than being representations of an external real world measured by their similarity to that world, are constructions measured by their internal coherence. (Dahlbom and Mathiassen 1993, p. 42)

Compared with Enlightenment philosophy, this was a romantic philosophy that distinguished between the world in itself and the world of phenomena experienced by every individual person, so that it became impossible to hold absolutes, to *know* something with absolute certainty. To accept Descartes' perspective means accepting that anything wrong with the world might actually be something wrong with the perceiver, and as a result, the early romantics tried to find answers to questions about the world *in human nature*. Later romantic philosophers transferred the origin of reality from the individual to culture,

claiming that our perceptions of the world are cultural in origin. In defining concepts and conceptual schemas as constructive creations, the romantics introduced the idea of multiple, culturally shaped realities. Rather than looking for objective truth, the romantics wanted to use different perspectives to open up diverse and meaningful perceptions of reality, in order to bring new meanings and deeper understanding. With the freedom to choose multiple perspectives, the seemingly straightforward question ‘what is this object’ becomes ‘what do you perceive it to be?’ (Bullock and Woodings 1983).

This kind of perspectivism orients us in Coyne’s (1995) ‘critical’ quadrant (Figure 2). Nietzsche (1844-1900) used perspectivism to argue that preferred metaphors, theoretical positions in science, and established interpretations tell more about the background and interests of their proponents than they do about the world. When an assertion such as ‘object-orientation is a superior paradigm for software design’ is put forward and vehemently argued for, perspectivism accounts for the proponent’s motivation to defend a position instead of a concern to understand the truth or otherwise of the claim. Rather than establishing evidence or truth, Nietzsche would seek to ascertain the controllers and the controlled, the power relationships represented by such a statement. Truex (2000) has done a similar kind of critical assessment of today’s rationalistic software methods with informative results. Perspectives play an important role in understanding the relationship between stakeholders and the designer in the design process, and how a designed artefact is perceived and used. The use of a sceptical analytical framework (ie. as per the ‘critical software architect’) can be a powerful aid in comprehending design situations.

### 3.2.3 The Design of Existence

Perspectivism replaces absolute truth with as many relativistic constructions of truth as there are distinct actors. In the period that followed, Jean-Paul Sartre (1905-1980) furthered the extension of relativism to existentialism—the thought that the world exists only because we want it to. By implication, things have no purpose other than that invested by us. Sartre described humankind as being entrapped in an existence without essence, and so he constructs ‘a system of projects directed toward the future’ to infuse existence with meaning (Bullock and Woodings 1983, p. 677). Existentialism is one of the few philosophical movements to explicitly account for the relationship between man and the designed or built world. Existentialism accounts for the driving force behind material progress. The absence of absolutes and universal meaning and our inability to rest comfortably with what we have achieved motivates us to design the world around us, as if the act of perpetual designing served to account for our identity.

If Sartre's work explained the separation of mind and body, human striving and the need for design to justify existence, Heidegger (1889-1976) further explained 'being' in terms of our relationship with the objects that surround us. Heidegger's work developed out of phenomenology, the work of his teacher Husserl (1859-1938). He used the term 'Dasein' to describe the separation of subject and object—the particular situation that humans find themselves arbitrarily 'thrown into'—a condition characterised by having consciousness but being surrounded by inert material objects (Steiner, 1979). Far from being an obtuse concept, 'thrownness' turns out to be a very ordinary and familiar one, known to us as the fluid and unpredictable interactions we have with objects in our environment. Winograd and Flores (1986) summarise Heidegger's philosophy, with implications for software and system design in four points. Firstly, our implicit beliefs and assumptions cannot all be made explicit—there is no neutral viewpoint from which everyone can see all things, and a truly objective understanding of most worldly phenomena is difficult and often impossible to achieve. This view aligns with Budgen's (1994) characterisation of software system design as fitting Rittel and Weber's (1984) 'wicked' class of problems. Secondly, practical understanding is more fundamental than theoretical understanding—the Western philosophical tradition, Winograd and Flores (1986) observe, is based on the assumption that a detached theoretical point of view is superior to the involved practical viewpoint. Heidegger reverses this, claiming that it is only when we interact with the world that we begin to fundamentally know it. Detached theorising can be informative but it can also obscure phenomena by isolating them. Thirdly, we prefer to relate to things directly, not via models—Heidegger rejected 'mental representations' in favour of 'concernful acting in the world'. Winograd and Flores' interpretation stresses 'concernful activity' (informed action) over 'detached contemplation' (observation only, with no direct experience of phenomena). Fourthly, meaning is fundamentally social, it emerges from interaction and not purely from individual action—the rationalistic view of cognition, Winograd and Flores propound, is individual-centred. For example, we teach meaning in language to individual learners, whereas linguistic meaning in language can only emerge in the context of social activity.

Heidegger also claimed that properties are emergent in use, which implies that an object itself outside of use has no properties. Winograd and Flores use an example of a hammer to illustrate. As a craftsman drives a nail with a hammer, the hammer does not exist in the consciousness of its holder. Rather, the craftsman uses it as an extension of his forearm, sensing the transfer of energy from hammer to nail, and the reaction of the nail and timber in the subtle bounce of the hammer on impact. But when the hammer slips or the blow

glances the nail its holder becomes (sometimes painfully) aware of its presence—its ‘hammeriness’ emerges in momentary failure. The hammer’s properties are defined by its current context of use and the action of the moment. Heidegger used the term ‘readiness-to-hand’ to describe this melding of actor and object, and the term ‘breakdown’ to describe the jolt back to awareness when the object momentarily misbehaves. As observers, we may talk about the hammer and reflect on its properties, but for the craftsman engaged in the thrownness of driving home a nail, there is no hammer there at all. The fact that an observer (as opposed to the actor holding the hammer) can never know this illustrates the experiential nature of ‘being’. Achieving this kind of complete transparency in a designed artefact, and anticipating and minimising the impacts of its ‘breaking down’ behaviours, define important goals for any designer of artefacts for use.

#### 3.2.4 The Constructivist Alternative

To acknowledge interpretation in system and software design opens the door to the influence of constructivist philosophy. If we accept that knowledge is cultural, an artefact made by man via a process of interpretation, then both truth and reality can be said to be ‘constructed’ and may be viewed as social phenomena. If cognition is a process of social construction rather than a natural process of the mind, then cognition must be regulated by social norms and is better understood as a socially organised process. It is to claim that thinking relies on intellectual tools and materials supplied and reinforced by culture, some of which are internalised and some of which are supplied by social, natural and built environments.

In the constructivist view, problem characteristics are not objectively discovered and analysed, they are constructed from the observer’s own perspective. Observation is affected by personal priorities and values, by the methods used as orientation aids, and by the particular interaction between participants during the construction process. Where differences in two analyst’s requirements emerge, they can be accounted for by differences in perspective. Similarly, constructivist analysts do not apply predefined methods, in fact there are no such things as methods *per se*—the constructivist analyst is ultimately concerned with the processes of selecting methods and then adapting them *in vivo*. The process of software and systems design from a constructivist perspective is essentially one of method assembly, because to assume the existence of a predefined, optimal or best-practice path through these realms is an oversimplification.

Gadamer reminds us that ‘it is not so much our judgements as it is our prejudices that

constitute our being' (Gadamer 1976, p. 8). The emphasis shifts from analyst as transcriber of a fixed (albeit complex) picture to analyst as individual, idiomatic observer of a scene and constructor of a particular interpretation. Floyd (1992a) portrays the analyst as 'making choices in an open situation, where there is more than one possibility' (p. 16). The analyst makes choices in selecting the aspects of the problem for inclusion in a model, choosing modes of interaction with the computer in determining the system's architecture, and in the implementation of key concepts. Design, when portrayed in this fashion, is more of 'a process of cooperative learning' than one of executing a deterministic engineering process (Keil-Slawik 1992, p. 182).

Of the choices made by a designer in a typical software design episode, only some are made explicitly. Others are made subconsciously via familiar, habitual actions and tacit design acts that close off consideration of alternatives. Some of these are motivated by highly idiomatic drivers—the avoidance of inter-personal conflict or difficult thinking, or the personal desire to extend boundaries or capabilities known only to the individual. As a result, software designs come about through a mixture of objective and subjective (even personal) reasons. Although the designer has at any point in time a vast array of options available, the individual may choose to impose artificial constraints in order to make the design task manageable:

In constructivist thinking, the ontological question of what *is* is placed in relation to the epistemological question of what we can know in a poignant way. Only what *we can know* is accessible to us, and it is accessible in those terms in which we know it. The seemingly safe ground of the given reality reveals itself as built up in processes of our own making. (Floyd 1992a, p. 17)

Learning is a process of continuously building on yesterday's models of reality and grounding our new models in our experience of each new event and episode in an endless co-evolutionary process. Lycett (1998) describes social regularities as being emergent and not given *a priori*, and recognises that they are constantly shifting and evolving. For example, Amann (1992) claims that the social foundations of knowledge systems make expertise socially emergent. In the long term, Lycett argues, methodical systems will always disappoint, as they do not allow internal variety to evolve in line with the environment, and represent temporal snapshots, ultimately leaving us with static systems that are expected to operate in a dynamic world. This view mirrors the history of built-world exemplars of modernist master-planning (Pruitt-Igoe, Levittown) discussed in the previous chapter. In response, systems researchers propose a shift in the design process to account for evolutionary complexity (Kaplan 2000; Lycett and Paul 1998; Taylor 2000b).

Dahlbom (1992) claims that software designers alternate between rationalistic and constructivist perspectives almost by whim, sometimes choosing two perspectives for the same object at different times. We think of construction as engineering when we wish to stress an artefact's functionality, as science when we wish to relate it to popular conceptions of truth, or as art when we wish to stress aesthetic, edifying or communicative qualities. Our constructions suit our perceived needs at the time. We construct reality because we wish to change reality—in fact the possibility of change is a strong motivation for the idea of a constructed reality. Constructivists who argue against a science that perceives itself as mapping reality (or more generally against the Cartesian idea of knowledge as representation) are driven by the concern that such ideas stand in the way of change. This is consistent with Feyerabend's (1993) desire to defer or avoid commitment. More generally, the ultimate purpose of constructivism is to provide a basis for action. As Dahlbom notes, 'we are all masters at adjusting wildly varying descriptions to the very same actions' (p. 121), but in the final analysis, designers act and are measured by their actions as reified in their designs. Constructivists argue that when we change our reality we change our actions, and to the extent that this is true, constructivism has relevance in software design practice.

### 3.2.5 Romantic Constructions of Design

By the early nineteen nineties, a degree of unease with the prevailing software lifecycle models was emerging. Floyd (1992b) expressed doubts about the conventional view of software development as 'production' on the basis of fixed requirements, the separation of production from use and maintenance, and the division of production into linear phases. Floyd also criticised the view of methods as rules laying down standardised working procedures to be used without reference to the situation at hand, and the emphasis on formalisation at the expense of communication, learning and evolution. Floyd records her motivations for this attack as 'the glaring contradictions' between software engineering and the reality of software projects in both industry and academic settings, despite the fact that many of these projects were ostensibly conducted along traditional lines (p. 87).

Floyd's alternative constructivist definition of software design is 'the creative process in the course of which the problem as a whole is grasped, and an appropriate solution worked out and fitted into human contexts of meaning' (p. 87). Software design is 'a self-organising, dialogical process in the course of which a gradually materialising web of design decisions is stabilised' (p. 74). Software development is an insight-building process, expressed in terms of multi-perspectivity, self-organisation and dialogue. The design of

systems involves a series of decisions, each as a result of particular forces existent or perceived in the context of the decision at the time of its making. When viewed holistically, a web of design decisions emerges, where each decision point resolves the considerations of its predecessors, and in the case of reviewed decisions or rework, its successors. Design is successful as a whole if the web of design decisions is stabilised in the course of revisions—that is, when it withstands evaluation and is acknowledged as ‘good’ or ‘suitable’ by those involved in the design process and the corresponding system’s use. Absolute notions of completion, such as the demonstration of functional or non-functional behaviours, are supplemented with the designer’s observation of the web’s convergence on certain completion criteria. For the designer, the decision web must be *viable*, not necessarily optimised or universally correct.

These characterisations of design have their origins in what Maturana and Varela (1980) refer to as autopoietic systems. An autopoietic system is a network of processes or components that combine in a holistic fashion to form a unified entity, and continuously regenerate by using the processes that produced it in the first place. Autopoietic models have been used as the basis of distributed control systems. In contrast with rigid, top-down hierarchical goal-driven control systems, autopoietic systems thrive on uncertainty and reconstruct themselves on the basis of what they learn from their environment. Goguen (1992), Floyd (1992b) and other constructivists lean towards autopoiesis as a basis for alternative behavioural models of software design processes in team settings. A software development team illustrates survival-driven autopoietic characteristics by the way it responds to a crisis. An unhealthy project might respond to crisis by reorganising, reassigning responsibilities, redefining sub-projects, and even trying to re-constitute the conditions that defined its initial success. On the other hand, a healthy project might respond to a crisis by developing new tools to enhance its own productivity (Goguen 1992).

Post-rationalists oppose the use of rationalistic approaches to system design for another fundamental reason. They argue that the objects of enquiry in information systems design are different from those in the natural sciences, because users, developers and other stakeholders are not natural objects but conscious subjects. Consciousness is a characteristic which the natural sciences so far have not dealt with. Consciousness is important for conceptual design because information systems are social communication systems, formed around shared meanings. Viewed in this way, the design of information systems is like the design of human communities, and such design requires a different



approach to that practiced in the natural sciences. In considering data modelling, Klein and Lyytinen (1992) make an important distinction that equally holds for all conceptual modelling. Questions of ontology concern whether the universe of discourse is given (ontologically exists prior to any human perception) or socially constructed. Rationalists regard the universe of discourse as given, while constructivists regard it as socially constructed through processes of communication. Habituation, language tradition and institutionalisation through roles and norms play key roles in social construction. Klein and Lyytinen's description of the 'sense-making process'—of design practice as a consequence of activity in the world—draws from Boland, who distinguished between decision-based (rational) and action-based approaches to information systems design:

The design of an information system is not a question of fitness for an organisational reality that can be modelled beforehand, but a question of fitness for use in the construction of an organisational reality through the symbolic interaction of its participants. In essence, the information system is an environment of symbols within which a sense-making process will be carried out. (Boland 1979, p. 262)

Applying this idea to conceptual design, Klein and Lyytinen suggest that models should attempt to represent the language by which the users communicate in the system's domain. This language-driven development view conflicts with the traditional reality-mapping view of the rational approaches. User languages are rich universes of discourse that convey understanding, knowledge and orientation, as well as substantial amounts of sociological 'noise' such as explanation and justification. The modeller engages with the process of selecting and interpreting a subset of the user language appropriate for the goals of the system being developed.

<b>Paradigm Characteristic</b>	<b>Rational</b>	<b>Situated</b>
Philosophical foundation	Modernism.	Postmodernism.
What is design?	The execution (and further elaboration) of a well-defined and well planned process.	A creative activity of unplanned discovery, elaboration, imposition of change and immediate reflection, that exists only in relation to situations.
The nature of problems — how do problems manifest?	Problems exist, can be discovered and described, are stable with respect to the observer and the timeframe of the designed solution.	Problems are a relative construction of the observer. Problems are wicked—deeply embedded within a context, unstable, dependent, constantly changing....
Problem definition — how is the problem to be solved demarcated?	Performed through objective analysis. A problem, like a law of nature, is there to be discovered. More or better analysis leads to a better description.	Designer is an active participant in structuring the problem. Designers evaluate their own (and others) actions in structuring and solving the problem.
Process definition — what defines the design process?	Driven by the objective processes of scientific method and subject to the paradigm of the day.	Driven by the individual designer in-situ.

Table 2: Comparison between rational and constructivist (situated) design.

In design research, the constructivist view finds expression in situated models of cognition and action. These are explored more fully in Chapter Four. At a summary level, Table 2 contrasts the characteristics of the rational and constructivist (situated) paradigms as they apply to design.

### 3.3 Models of Design

Through history, design has been portrayed with principles, metaphors and models, and there are about as many classes of models of design as there are domains of design knowledge. Models are generally used to organise and illustrate a group of related characteristics of design for a particular purpose. The brief survey of design models that follows commences with a discussion of the types or categories of design models and then continues with those models that have had the most impact on conceptions of software design. In each case, the model's history and purpose illuminate its place in the epistemology of design theory.

### 3.3.1 Categories of Design Models

Types of models of design vary from the extremely simple to the arbitrarily complex. One simple model has achieved extreme longevity. The Roman architect Vitruvius identified ‘firmness, commodity and delight’ as the three fundamental characteristics of good design, and illustrated their mutual dependency with a three-legged stool, a trivet reinterpreted in every age of design. In Vitruvian style, Schumaker proposed ‘reason, perception and soul’ in 1938 (Schirmbeck 1987, p. 148) as the essential dimensions of design appreciation and evaluation. Schirmbeck decomposed the designs of nine prominent architects into three broad categories—rational, symbolic and psychological. Rational principles describe functions that have a rational objective (such as a geometric layout). Symbolic principles transmit a carefully chosen meaning through familiarity, and psychological principles follow from the combination of rational and symbolic principles and are concerned with aesthetics, habitation and use. Mayall’s (1979) three-dimensional design map for a stone axe (Figure 5) illustrates the impact of Vitruvius’ model on contemporary design theorists. Mayall’s model adds the temporal dimension, demonstrating how the relative importance of each leg of the Vitruvian stool may vary with time.

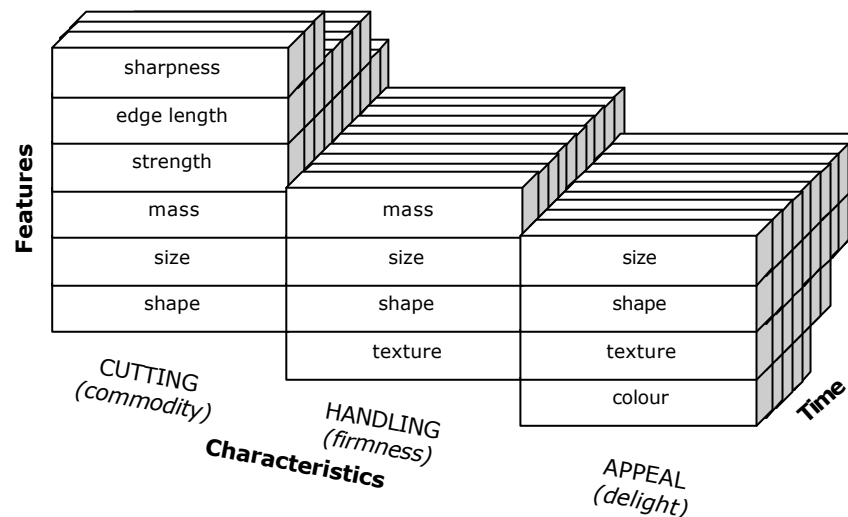


Figure 5: Three-dimensional Vitruvian design map for a stone axe, reproduced from Figure 4 in (Mayall 1979).

Beyond the Vitruvian canon and its descendants, architecture has demonstrated an unfortunate predilection to protect and even mystify rather than explain design and the design act. The architecture profession’s motivations for such concealment are not difficult to discern. Schirmbeck (1987), for instance, thinks that ‘it is almost impossible to give people in other specialised disciplines, who are concerned with planning, any direct

description of the architect's procedure when he is engaged in design' (p. 3). Further to this, 'there are no direct rules as to how architects should act in those circumstances' (p. 3). The modern movement claimed to have externalised the mysteries of design through the doctrine of functionalism. But even functionalism, with its notionally explicit design language, relied on a degree of intuition. Venturi (1977) observed that Le Corbusier, Bauhaus doyen Laszlo Moholy-Nagy and other leaders of the movement referred to 'intuition', 'imagination', 'inventiveness', and the 'free and innumerable plastic events' that regulate architectural design (p. 133). The result was a tension between two apparently contradictory ideas—the modernist process of 'form follows function' and free expression. What appears on the surface as a hard rational discipline of design turns out rather paradoxically to be a 'mystical belief in the intuitive process', Venturi concluded (p. 134).

Perhaps as a result of these protectionist attitudes, design theorists, working outside of the traditional architecture milieu have contributed a more diverse range of design model types. Two significant families of design models are the generative and taxonomic (or typological) models. Generative models depict design as the mechanical, prescriptive combination of sub-solutions or components. The earliest generative approach to design is evident in Aristotle's assertion that all animals can be described as compositions of a small number of body parts (legs, ears, eyes, tails) which vary in their size and exact form (Mitchell 1977). Generative models represent design as a traversal through discrete states in a (typically large) state space, from an initial state (in which requirements are known) through intermediate states (in which the design is incomplete) to a final state (in which the design realises all the requirements). Rules or actions define allowable operations in each state. Because the state spaces are immense in most real-world contexts, strategies and heuristic knowledge are employed to achieve progress.

From analogy with biological taxonomies that rely upon classes such as species and family, taxonomic models of design assume a typology of basic, recurring forms from which the designer performs selection and instantiation, with or without modification (Steadman 1979). In architecture, the elements in the typology are familiar—bridges, churches and domestic houses. Each of these may have subtypes that define the building's details, such as cathedrals or chapels, each of which may then have options for design elements such as doorways, arches or columns. This process of instantiating a specific design from generic design templates is the inverse of analysis, the determination of generalised designs from many instances.

Broadbent's model of design (1973) typifies the typological class. It offers four classes—

pragmatic, iconic, analogic and canonical. Pragmatic design is motivated by function alone, iconic design by adherence to semiotics or an accepted type, analogic design is concerned with the communication of ideas from other domains or discourses in the object at hand, and canonical design preserves the rules of a well-known or orthodox system. Beyond typologies of generic design forms, there are classifications of the heuristic rules and constraints that designers use to constrain the problem space when designing. Rowe (1987) suggests anthropometric analogy (physical occupancy of a space drives the design, such as the act of ascending a staircase); literal analogy (an unrelated object is projected onto a design, such as the rendering of a tall ship's billowing sails in the shells of an Opera House); and environmental relations (the design strongly relates to other elements of its immediate context).

Taxonomic and typological models of design assume relatively fixed, stable contexts in which the designing occurs. One of the first theories to acknowledge the distinction between stable, closed systems, and open, volatile systems, was systems theory (Checkland 1981; Checkland and Scholes 1990; Churchman 1968; Senge 1992). Systems theory and its derivatives claim a unified set of principles of systems and systemic knowledge. The model asserts that systems are closed or open. Closed systems are those that operate largely independently of environmental factors (such as most machines) and for which provable and testable theorems can be established, whereas open systems rely upon complex transactions with their environment (such as economies, organisations and organisms) and are not predictable or well understood. It is generally not possible to prove theorems of open systems or their behaviour.

As this distinction illustrates, universal models of design are invariably compromised because they cannot describe both open and closed system behaviours. Despite warnings in systems theory against applying the predictable principles of closed systems to open systems, attempts to do just that are common. In management science, for example, principles of operations research are applied at the organisational level (Hesse et al. 1980). In design theory, Simon (1985) campaigned to establish a science of design, 'a body of intellectually tough, analytic, partly formalizable, partly empirical, teachable doctrine about the design process' (p. 58). While these are commendable aims in the conservative tradition of science, design theory is not as yet science, and care is needed wherever universalism is claimed. Attempts to formalise a science of design are fuelled by partial successes with subsets of design that are amenable to closed systems theory. Generalisation of the behaviour of closed systems to open systems must always be treated

with caution.

Open systems such as organisations (for which information systems are designed) are generally not well behaved—they do not regularly follow rules, and what rules might exist at a point in time are constantly changing. Typological models are Aristotelean and do not support the designer working in environments that call for change or reinterpretation of the taxonomic model. All taxonomies (and methods anchored on taxonomy)—from history’s biological classification schemes (Steadman 1979) to contemporary analytical problem frameworks (Jackson 1995)—suffer from such misfits. The emergence of computer technology as infrastructure upon which social and economic systems are built exposed system designers to open systems. Problems that are deeply embedded within dynamic social and cultural contexts have been referred to as ‘ill-defined’ or ‘wicked problems’ (Rittel and Weber 1984). Wicked problems exist in environments that are inherently unstable and unpredictable, such as Checkland’s (1981) open systems. By definition, predictive taxonomies cannot mature in such environments, and in response, candidates for design models that take some account of ill-formed problems have emerged. For example, Broadbent (1973) advocated abandonment of formal methods so as to leave room for empirical evidence and interpretation, citing emerging techniques such as brainstorming, synectics, and the use of tabular data and process maps as guides rather than as prescriptive rules. Any approach to system design for open systems involves highly contextual planning over short horizons (Rittel and Weber 1984). Checkland’s (1981) soft systems methodology and autopoietic models are examples of alternatives.

### 3.3.2 Vernacular design models

Another model of design has proven particularly successful in open environments. Design techniques, specific designs and a wealth of information about design and fabrication have transferred for centuries from master craftsman to apprentice by way of the processes of craftsmanship. Craft is defined by the application of talent, skill and the learned outflows of expertise. Craft—being colloquial and vernacular—is the antithesis of modernism which treats all places and all cultures in the same way and elevates theory over everyday experience and learned, tacit knowledge (Thackara 1988). In architectural terms, vernacular buildings are seen as the opposite of whatever is academic, high style, or polite. The term ‘vernacular’ was borrowed by architectural historians from linguists who used it to mean ‘the native language of a region’ (Brand 1994, p. 132). Vernacular design is a highly effective communication medium. The Cotswold villages express a localism and subtle parochialism in grey stone that paints the inhabitant’s life and times vividly, while

New England's Cape Cod houses, completely different stylistically, are also highly evolved forms built to local conditions and with available materials. Vernacular design is typological in that it draws upon the prevailing cultural norms of what a house, a barn or a wagon should be. In vernacular design, there is no need for a formal plan, in fact the amount of detail on a plan is an indicator of the degree of cultural disharmony—the more minimal the plan, the more completely the architectural idea abides in the separate minds of architect and client (Brand 1994).

A principal difference between industrial product design and pre-industrial craft evolution is that the designers are separated from production (by scale drawings) in place of the product as the medium for expression, experimentation and change. Where craftspeople interact directly with the object being made, industrial designers manipulate scale drawings. The use of drawings as a model of reality permits larger projects than an individual craftsman working alone can attempt. But while making increased rates of production possible, the introduction of drawings risks damage to a feedback loop between designer and fabric, and increases the change-and-evaluate cycle. Figure 6 from Walker and Cross (1976) illustrates the progressive separation and specialisation of maker, designer and patron roles in the progression from vernacular to rational design.

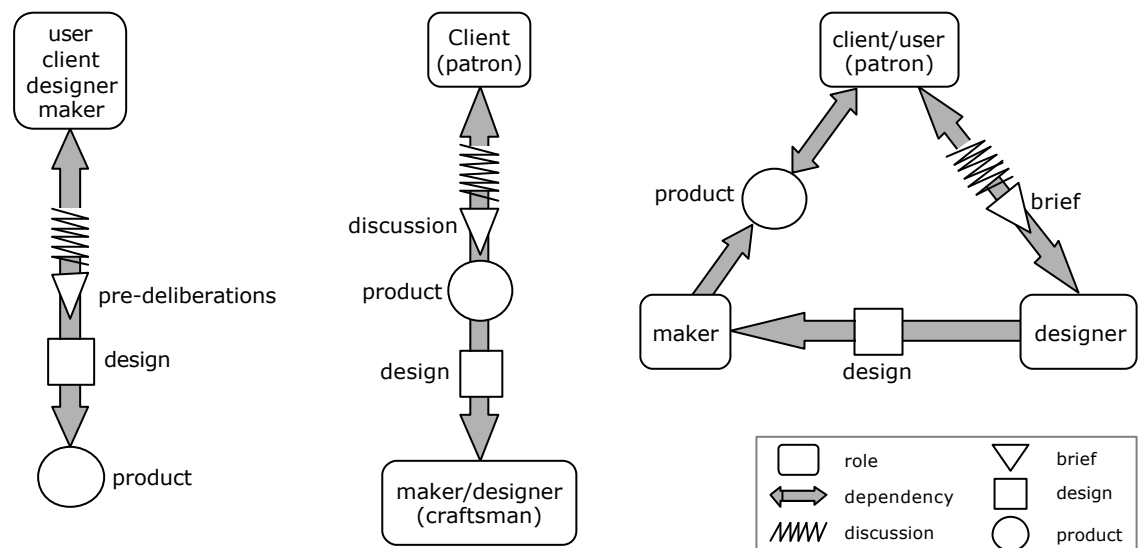


Figure 6: 'Vernacular design' (Fig 72, p. 58); 'the empirical exchange' (Fig 71, p. 58); 'direct patronage' (Fig 70, p. 57) (Walker and Cross, 1976).

The practice of craft is partly a consequence of the phase (but not the maturity) of design knowledge in the particular context. Alexander (1964) defines 'unselfconscious' and 'self-conscious' design as an attribute of culture:

I shall call a culture unselfconscious if its form-making is learned informally, through imitation and correction. And I shall call a culture self-conscious if its form-making is taught academically, according to explicit rules. (Alexander 1964, p. 36)

The unselfconscious process is that which goes on in the traditional craft or architectural vernacular contexts, while the self-conscious process is that which is typical of present-day, educated, professional designers and architects. The distinction is not an absolutely sharp one, as Alexander admits, and in the historical development of design a gradual transition from unselfconscious to self-conscious methods is present (Steadman 1979). The important difference, in Alexander's view, is seen in the way in which design and production of objects is taught in either case. In the unselfconscious craft situation, the teaching of craft skills is through demonstration and by having the novice imitate the skilled craftsman until the apprentice gets the feel of the tools and techniques. In the 'self-conscious' process the techniques are taught by being explicitly formulated and explained theoretically. In the unselfconscious culture, the same form is repeated over and over again, and the individual craftsman must learn how to copy the given prototype. The designer's output is another indicator of which design mode is employed (Walker and Cross 1976)—the self-conscious designer produces a design to be fabricated by another party whereas unselfconscious designers fabricate their designs themselves and make no distinction between designing and fabricating. How knowledge is represented and manipulated in the design process is therefore an indicator of the supposed maturity of design practice.

Nonaka and Takeuchi's (1995) spiral model of knowledge (Figure 7) distinguishes the tacit from explicit phases of knowledge and illustrates the role of tacit-to-explicit externalisation techniques and explicit-to-tacit socialisation techniques. Tacit knowledge is made explicit through processes of reflection and externalisation, where it is tested against and combined with the body of explicit knowledge. Proven explicit knowledge is made tacit as it is operationalised and adopted into the work culture of the organisation or the knowledge domain. The cycle repeats itself, as tacit knowledge that can be identified and judged to be potentially useful is selectively externalised. Knowledge management is effected by stimulating, managing and monitoring this cycle.



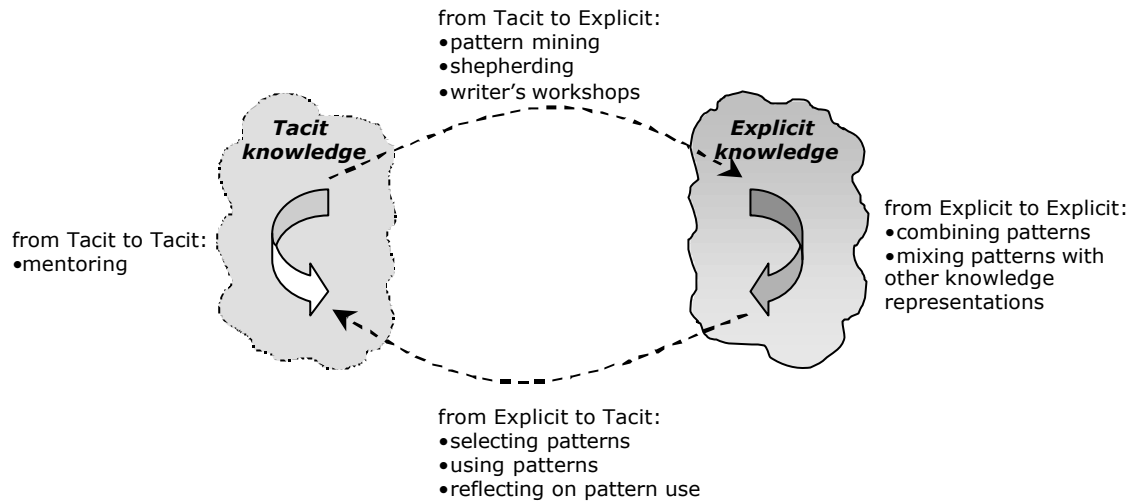


Figure 7: Spiral model of knowledge phases and transitions (Nonaka and Takeuchi 1995).

Until relatively recently, software design methods have generally not addressed techniques to externalise tacit design knowledge. Nonaka and Takeuchi's externalisation process is one of literate reflection, often benefiting from metaphor to bridge between understood and partially-tacit concepts, and analogy to clarify both the alignments and misalignments of detailed procedures or practice. Once metaphors and analogies have been agreed upon, the knowledge extraction process can move to more detailed modelling in whatever the most appropriate notation might be. Pattern writing (Gabriel 1996) is a knowledge externalisation technique. Pattern authors use a 'writer's workshop' process for the expression of design patterns. Once explicated, proven design patterns are able to be related to the existing body of knowledge of software design within a particular domain, and may be combined or further developed by others working in different contexts.

Despite the historical fact that the industrial revolution comprehensively crushed vernacularism, unselfconscious design is still regarded as viable and appealing for design in certain media. It was capable of producing artefacts that were ingenious in their design in the ways that they exploited physical effects of properties of materials, and the products of unselfconscious design were achieved within very severe limitations of material and manufacturing technique (Steadman 1979). Analysis of the craft and bricolage design metaphors (Louridas 1999) suggests that unselfconscious design is successful—sometimes even brilliantly so—because its scope is limited by available materials and tradition. The unselfconscious craftsman or bricoleur has limited latitude to extend or change highly evolved traditional designs and so their crafting leaves little room for failure.

The applicability of vernacularism to software development occurred to Jones, a theorist in the design methods movement in the United Kingdom, as early as the nineteen eighties:

The more I see of software designing the more I notice resemblance not to design in other fields but to craftsmanship. In each the designing, if such it can be called, is done by the maker, and there is much fitting, adjusting, adapting of existing designs, and much collaboration, with little chance of a bird's eye view, such as the drawing board affords, of how the whole thing is organised, though, in craft evolution, if not in software, the results have the appearance of natural organisms or of exceptionally well integrated designs. (Jones 1988, p. 219)

Jones notes that the context has to be stable (within limits) for long periods of time, even centuries, for craft evolution to be possible. Farm wagons exemplify highly crafted design—wagons adapted so closely to their environment that, to understanding eyes, they looked almost like living organisms, Jones writes. The provincial wheelwright could not avoid reading from the wagon the types of available timbers, the hardness of the ground, the uses of the vehicle and the nobility of the owner's horses. 'Is there any way', Jones asks, 'in trying to learn from these various modes of evolution, for the makers of software... to attain this almost magical accord with context when context is itself in flux?' (p. 219). Successful vernacular design emerges in stable societal contexts such that the essence of the craft is not substantially changed between generations of artefacts or craftspeople. Being evolutionary in nature, vernacularism cannot respond quickly to environmental changes, and the rapid emergence of new cultures and contexts for which traditional solutions are inappropriate or inadequate provides the most compelling argument against it. By explicating theory and theoretical understanding from practice, new forms that meet new needs can be devised quickly. The need to manage, scale and transfer design effort additionally promotes self-conscious design over vernacular design and continuously motivates efforts to explicate theory in a useful and reusable forms.

Vernacular design models account for the basic acts of continuous design, creation, fabrication and evaluation by the master practitioner. Craftsmanship implies highly developed skill and deep knowledge of the designs and of the materials. It accounts for the expression of the craftsperson's knowledge and skill in the form of the crafted object. Craftspeople pursue perfectionism and pleasure through making, and are often portrayed as being totally immersed in and consumed by their work. On the face of it, craft provides a useful descriptive metaphor for the personal behaviours of software designers during intense episodes of software development. Vernacularism may also be a useful model to understand tacit design practices of practicing software architects. However, its dependency on stable culture and context flaws the model for many theorists (McBreen 2002; Taylor 2001e; Taylor 2003; Taylor 2004).

### 3.3.3 Evolutionary Design Models

Vernacularism relies on a form of evolution for design selection, modification and propagation. The theories of evolution of species provide models that equally explain the propagation of software designs and design knowledge over time (Stebbins 1971). Certain physical artefacts reveal evolutionary progress over time in both primitive and modern societies. Hunting tools used by hunter-gatherers (Steadman 1979), farm wagons, aircraft and sports cars (Lawson 1997) illustrate evolutionary progression of form to improve both function and aesthetics. Most of these examples can be explained in basic Darwinian terms. Darwinian evolution relies on offspring inheriting its parent's characteristics. In any evolving inanimate object that does not reproduce but relies on humans to create copies, inheritance is realised by the copier's ability to perform the reproduction. To effect high fidelity copying, objects should be self-describing, physically and conceptually well designed and implemented, and able to be fabricated easily. In short, the artefact must be designed with copying in mind.

A recent model of cultural evolution that accounts for such evolution in ideas rather than life forms is memetics (Dawkins 1976; Dennett 1995). 'Memes', Blackmore (1999) argues, come from the endless variation and recombination of earlier thoughts from language, songs, works of art, mass-media and cultural stories—'human creativity is a process of variation and recombination' (of memes) (p. 15). Designs, or more specifically the knowledge required to reify a design, is more concrete than Blackmore's memes because they inform a design act which results in an artefact, an instance of the 'unit' of design knowledge. Like the subtle design changes made to Mitchell's wagons over time, they are selected less by the randomness of the cultural environment and more by the degree of success or otherwise of the resultant artefact. Design knowledge is a stronger concept than memes in that it is tied to physical artefacts, which allow other human 'readers' separated in time and space to reify the original design knowledge by conceptually or physically taking the object apart.

Both memetics and the more general wholesale application of Darwinian evolution as a model of knowledge propagation suffer from the metaphorical bridge between the organic (genetic) and epistemological worlds. Memetics as a theory faces a substantial impediment in that 'evidence' in a rationalistic sense can never be proven (Dawkins 1976; Dawkins 1996). Empirical research in programming has independently turned up many evolutionary ideas. Lehman (1985) studied programmer productivity, observing that the frequency and speed with which programs are executed draws almost immediate attention to any

shortcomings leading to a constant stream of enhancements. Kaplan (2000) suggests that evolutionary models might account for how software design occurs. Rather than imagining that there exists a privileged designer's stance or perspective, Kaplan states that design should be seen as a stepwise uncovering of evolutionary 'good design tricks' as well as a series of evolutionary 'moves' (or changes) that individual software artefacts are subject to over time. Frequently, 'useful moves' turn out to be either unanticipated *a priori* or to have unexpected or unintended consequences. Kaplan suggests that systems, like species, do not evolve under the selecting effects of a fixed environment, but rather they co-evolve, such that any change in a given artefact may have side effects that change the environment of others, and vice-versa. In this co-dependency, all artefacts implicitly exert a force of varying weight on each other's evolution. This model is consistent with Brand's (1994) notion of shearing layers, and with Kauffman's (1995) theory that evolution requires conditions that are to be found at the transition between systems of order and chaos.

#### 3.3.4 Technology Maturity Models

Technology maturity models express the inevitability of progress from informal to formalised design and production processes, and ultimately to industrialisation. The common characteristic of all technology maturity models is a near-linear progression of successive mechanisation, from handicraft to manufacture, and finally to the production line of the factory. The environments in which the work proceeds (workshop to factory to production line) exemplify the commonly accepted stages of maturity. Technology maturity models are framed in terms of tools—from hand tool to numerically-controlled production cell, the tool's level of automation directly maps the technology processes' relative maturity. Walker and Cross' (1976) four stage model of design modes (Figure 7, Figure 8 and Figure 9) demonstrate this technology maturity theme.

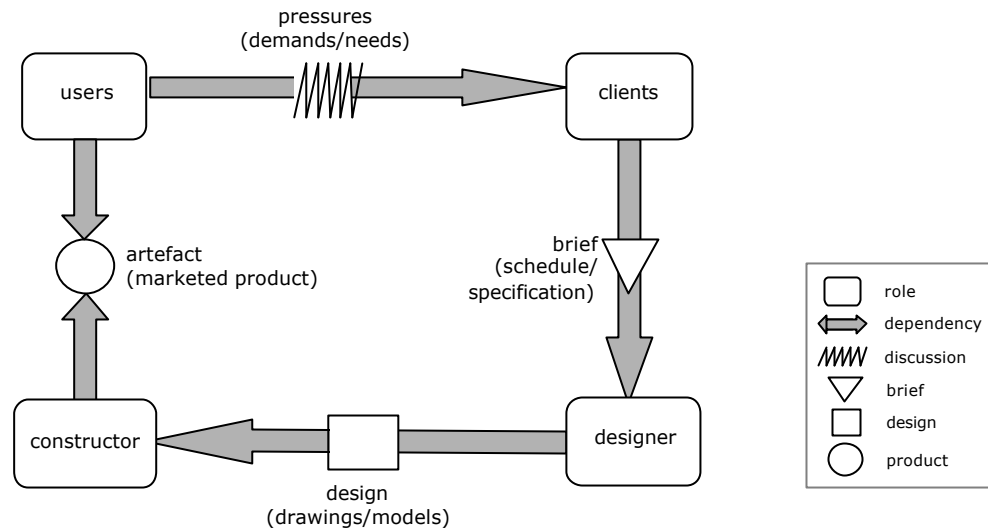


Figure 8: The ‘mature’ design process—the ‘rational design network’, from Fig 70 in (Walker and Cross, 1976).

Budde and Zullighoven (1992), however, suggest that different stages of technological development should be characterised not by the types of tools used but by the means of cooperation and the division of labour. Looking beyond the contrast between the workshop’s hand tools and the factory’s automated machinery, they observe the different roles played by the technical equipment within the processes of specialisation and cooperation. In the workshop, there is emphasis on supporting individual activities by employing tools and automata, so that cooperation between workers possessing different skills and qualifications can be optimised. In the factory, individual craftsmanship is replaced by appropriate methods, such that the division of labour is encapsulated within specialisations based upon the dictates of the machine, or production line, and individual effort becomes currency in a Taylorian (1911) economy. As work becomes routine and mechanised, people increasingly play the role of the unknowing link between a conglomerate of technical implements. By coupling workers and technical implements to form integrated units, the production line reduces individual skills to the lowest common denominator required to enact mechanised routines.

By contrast, tools used within a workshop setting primarily support skilled workmanship. A workshop offers a set of tools, but does not implement an overall strategy (ie. a methodology) other than certain techniques that automate routine activities. Cooperation (the successful distribution of tasks) is maintained by people, not by machinery, although technical equipment supports this process. Budde and Zullighoven (1992) find that the work of software developers is characterised by teamwork, cooperation, skill and expertise of the individual. Certain divisions of the software development labour force, such as the

distinction between analysts and programmers, are now regarded as an artefact of waterfall lifecycle models rather than of the nature of the work itself, they claim. Budde and Zullighoven conclude that ‘the tasks of software developers can be more suitably compared with the work of a craftsman in a workshop than with the work in a factory’ (p. 264).

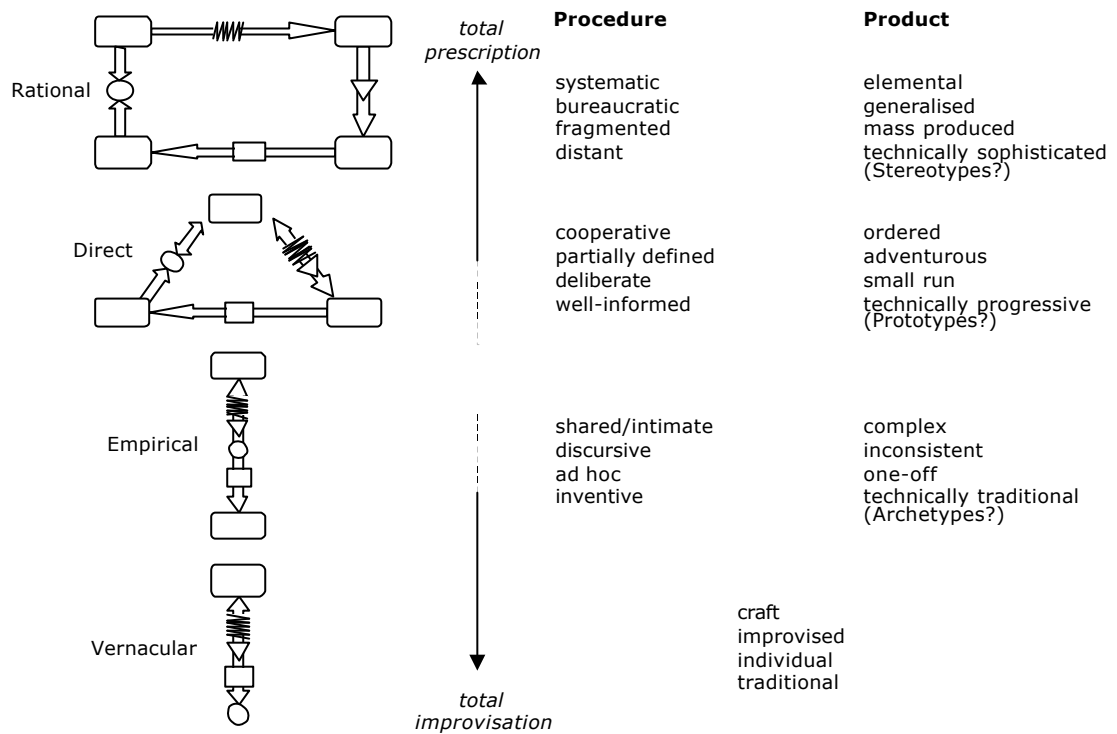


Figure 9: The four design modes compared, from Fig 73 in (Walker and Cross, 1976).

The debate between the proponents of the software workshop and factory models continues apace. The advocates of the software factory argue that software as a design medium is not sufficiently unique to warrant different design or management processes, and that the medium’s propensity to frustrate attempts at automation are due more to the worker’s resistance than limiting characteristics in the medium itself. The software craftspeople (AgileAlliance 2005) counter-argue that software is a projection of its maker’s creativity and that to try to automate its production misunderstands its nature. They argue that only the most routine and low-value software artefacts have been successfully mechanised, and that in the so-called ‘software factories’, a separate *ad hoc* and unacknowledged dimension of creative network behaviour must occur in order to account for whatever software ‘production’ occurs. Middle ground in the debate can be occupied by distinguishing between conceptual design (to be done by craftsmen/analysts) and routine work (to be done in the factories). Offshore outsourcing is sometimes used to

evidence the viability of this hybrid position.

### **3.4 Conclusion**

Models of design must marry rationality with the *a*-rational design act. This gulf stems from the core of human experience of the world, is deeply rooted in human psychology and sociology, and has found expression in every age since the Enlightenment. The conception of a design for a new structure in any medium can involve as much a leap of the imagination and a synthesis of experience and knowledge as any artist is required to bring to his canvas or paper (Petroski 1992). Once that design is articulated by the designer-artist, it must be analysed by the designer-scientist in as rigorous an application of the scientific method as any scientist must make. Some theorists have attempted to pull all of design into the realm of science (Simon 1985; Warfield 1994). Others argue that to make the design process itself ‘scientific’ is ‘not only as nonsensical, but ultimately highly dangerous’ (Steadman 1979, p. 2).

Because every design model explicitly or implicitly expresses a relationship with a notion of reality, design theory is unavoidably positioned within a philosophical milieu. Suchman (1987), Winograd and Flores (1986) drew upon Heideggerian views to motivate a substantial change in direction for artificial intelligence research. Dahlbom and Mathiassen (1993) used the philosophies of Aristotle and Plato, Descartes and others to contrast the mechanistic and romantic schools of systems development when a broad awakening to the social context of systems development was gaining momentum. An awareness of philosophy amongst theorists and practitioners can contribute a framework that builds bridges between the disciplines and brings an increased sense of perspective. Models of design, regardless of whether they come from design research, architecture, cognitive science or information systems research cannot be used to accurately predict design processes or designer behaviour, and as a result, fail to achieve the status of models in science. No design model is universal—rather, design models describe particular aspects of design and designing. Coyne (1995) concludes that it may be useful to regard design as a progression through different models, and ‘a dialectic between a formal view (model) and a phenomenon’ (p. 247).

This chapter’s survey of design theory and software design orients the thesis’ research goal in a philosophical dimension. The next chapter surveys design from a situated and ethnographic perspective, after which the detail of the research method can be designed.

# **Chapter 4:    Situated and Ethnographic Accounts of Design**

Design is not a neutral tool; it is a planning activity whose aims and procedures are dictated by commercial and political interests. Design is about decisions and priorities, not equations and logic. (Thackara 1988, p. 12)

## **4.1        Introduction**

Situated action and cognition (‘situatedness’) was introduced in Chapter Two as a basis for accounting for design practice, particularly as an alternative view of activity, and in Chapter Three as a theory to demonstrate constructivist philosophy. If the task of explicating expert designer’s accounts of their practice is to be done effectively, the notion of situatedness as it applies to software design needs to be reduced to some recognisable phenomena and characteristics. This chapter provides the theoretical background against which the analysis and case study chapters can pursue this goal. Firstly, a summary of the theories of situated cognition and action is presented. Secondly, the ways that design is typified in three generations of software engineering lifecycle models are discussed. Finally, selected socio-cultural models of design from information systems research are surveyed. The chapter concludes with assertions on situatedness and ethnography that collectively allow a research method to be designed (Chapter Five).

## **4.2        Situated Cognition and Action**

The situated movement—situated language, situated cognition, situated action—may be seen as a reaction to the historically dominant classical view of mind. In constructivist terms, cognition is viewed as experiential processes of bringing forth concepts and insights that fit our experience, and are viable for achieving our aims in open situations in which we



interpret our needs (Bateson 1980). Although frequently left undefined or ambiguous in the literature, situatedness is not a difficult concept. It has its simplest and most tangible manifestation in language. Different uses of words denote diverse meanings based upon situation (Searle 1969). As even young children realise, words like ‘here’, ‘I’ and ‘now’ can be used on separate occasions, by different individuals, to refer to distinct people, places and times. When two people in heated conversation shout ‘I’m right!’ their utterances coincide in meaning but reveal differing interpretations (Barwise and Perry 1983). Intended meaning is therefore a function of context and interpretation (Maturana and Varela 1980).

Kirsh (1995) points out that people use context as a mechanism for efficiently compressing cognition and communication. It is easier to determine which of two objects is longer by placing them side-by-side than to deal with numeric comparisons of absolute dimensions—a simple arrangement of the objects in context does the computing for the observer. An oft-cited situated cognition story attributed to Nardi (1996) recounts a Weight Watcher who, when asked to make three-quarters of a recipe that called for two-thirds of a cup of cottage cheese, measured out two-thirds of a cup of the cheese, flattened it into a circle, and cut away a quarter of the resulting disc. Unconventional but effective behaviour such as this demonstrates how human cognition is often unplanned, opportunistic and situation-specific, and how rational application of theory (such as doing the maths) may get passed over in favour of pragmatic methods that are promoted by the situation and enabled by the tools at hand.

It can be argued (as Clancey (1993) does in his debate with two leading symbolist cognitive scientists) that the ‘discovery’ of situated cognition (or the elevation of pragmatism over detached theorising) occurred somewhat independently in a range of epistemology as diverse as design, philosophy, architecture and cognitive science over a decade for the same reason—the perceived failure of rationalistic theorising. Clancey makes this argument with the following illustration. In the rationalistic (scientific) paradigm, regularities in nature are caused by laws—for example, a falling object accelerates so as to obey the law  $f=ma$ . Theories epitomise knowledge, and rational behaviour of systems and people obey general laws of logic. Therefore, human knowledge consists of facts and laws stored in memory, that drive observed regularities in behaviour. Clancey claims that this rationalistic view of cognition and action has distorted human activity—a claim which he finds support for in architecture (Alexander 1979), organisational learning (Nonaka 1991), professional training (Schon 1987), musical invention (Bamberger 1991), the design of

complex devices such as photocopiers (Suchman 1987) and the use of computers in business (Winograd and Flores 1986). Much of the material collected in Chapters Two and Three share the common characteristic of disputing or confounding the causality implied by Clancey's cameo of traditional cognition. In all of these fields, situated alternatives have emancipated researchers from the chains of the theory-leads-practice schools.

Of the themes that pervade situated cognition and action, two are particularly pertinent to software design—goal setting and planning. In the situated model, goals are formulated not ahead of action but in the midst of it, and in response to contextual cues. Chapter Two argued that planning in the traditional sense of action following a plan is a somewhat discredited concept. In a post-rational or situated model, both the goals and activities of planning and plans need to be redefined. The key questions in evaluating action in a software design situation are therefore 'when, how and why are goals formed?' and 'when, how and why are actions initiated?'.

#### 4.2.1 Situated cognition

The first generation of cognitive models assumed static Aristotelian concepts and logic. Logical reasoning with these models followed a process of establishing facts, applying single or causal chains of assertions, and evaluating the logical proposition with the highest associated probability (the most plausible proposition). Reasoning was thought to involve symbol manipulation by means of propositional logic and inference. They proved inflexible when applied to problems on a useful scale and were extended by adding measures of belief, evidence, certainty and probability. While they were suitable for constructing abstract systems of logic, the models fell well short of duplicating human cognition.

Cognitive models based on classification (Schank and Abelson 1977) assume that we act in situations and in the presence of certain objects by following generic responses that most closely match the current situation. Representations of such conditioning take the form of schemas, scripts or other generic descriptions of objects or situations. Contextual cues guide both the selection of the appropriate schema and its parameterisation or modification. Applied to design, the model suggests that the designer recalls a schema for the type of artefact and commences design by inserting parameters, fitting and modifying it for the current context. Schemas are thought to relate hierarchically, such that the schema for a type of artefact defines the generic layout, arrangements and features, and links to other scripts that define the generic form of the features at the next lower level of abstraction. The schema may additionally retain strategies and rules. The link back to

typological models of design (3.3.1, ‘Categories of Design Models’) is both clear and unsurprising. These models represent designing as a process of constructing schemas over time, matching presenting design situations to retrieved schemas, and instantiating them with localised modification (Stanfill and Waltz 1986). Classification-based cognitive models explain the role of experience and familiarity in a designer’s expertise, but do not explain acts of creativity in which features of one schema are cleverly combined with those of an unrelated one to produce a new design. Neither do they explain how a new schema is initiated. Also, the combinatorial explosion that typically results when attempts are made to move these models out of the laboratory and into real world settings presents a major limitation (Hamilton 1992).

In response to the dominance of symbolic cognitive science, Lave (1988) and Suchman’s (1987) theories of situated cognition contributed toward a significant shift in the philosophical basis of cognitive science. Lave studied people performing on-the-fly arithmetic in supermarkets. Lave recognised that quantitative procedures (such as people totalling their purchases) appeared to take their character from ongoing activity rather than the imprints of canonical forms. She concluded that people formulate a math problem only when they have also formulated a sense of an answer and a process for bringing it together with its parts, and that problem solvers proceed *in action*, engaging body, self and the setting in an integral fashion. Lave proposed new units of cognitive analysis—‘persons-acting’, ‘contexts of activity’, and ‘dialectically constituted activity’—to characterise the ‘value-laden, active, integrally contextualised character’ (p. 20) of arithmetic problem solving in practice. Lave argued that the most appropriate unit of analysis is ‘the whole person in action, acting with the settings of that activity’ (p. 17).

Shifting the boundaries of activity ‘outside the skull and beyond the hypothetical economic actor to persons engaged with the world’ (p. 18) opened the door to social relativism and hermeneutics (Gadamer 1976; Palmer 1969). Hamilton, amongst others, (1992) observed that hermeneutics was an essential counter-philosophy to rationalism for cognitive research. Hermeneutics offers a postmodern perspective of action by explaining the unavoidable relationship between social context and individual or group action. Gadamer claims that contextual factors dictate all our behaviours, actions and speech:

The phenomena of background and interpretation pervade our everyday life. Meaning always derives from an interpretation that is rooted in a situation... relativity to situation and opportunity constitutes the very essence of speaking. (Gadamer 1976, p. 88)

If situated cognition involves an inseparable co-dependency of thought, action and situation, it is not a phenomenon that can be identified and observed in isolation of others. This has made its definition difficult. Clancy's (1993) exploration of some common misconceptions about situated cognition provides some clarity. Situated cognition replaces the 'CPU view' of cognitive processing with a dialectic mechanism that simultaneously coordinates perception-action. It does not reject the value of planning and representations, rather, it seeks to explain how plans are created and used in already coordinated activity. It does not deny the existence or importance of representations (such as models or symbols), rather, it seeks to explain how perceiving and comprehending are co-organised. It does not dispute the value of cognitive science's schema and classification-based reasoning, rather, it attempts to explain how these kinds of regularities develop in behaviour, and how the flexibility for improvisation cannot be captured in symbolic models. It does not dispute that symbolic computer programs can construct and reason over a problem space, rather, it reveals why such programs cannot step outside pre-stored ontology.

To take a situated view of cognition to a design scenario does not mean ignoring or discounting the designer's plans, models or stated intentions to pursue goals. Rather, it means looking for holistic explorations of how these are coordinated, arranged and used before, during and after the design act. It also means forgetting positivistic preconceptions about scientific method and causality in order to allow other phenomenological insights to occur. Observations of situated cognition in design practice therefore take the form of rich, 'thick' descriptions of observed or self-reported behavioural patterns, illustrated by the selection and application of tools (for example, the Weight Watcher's rolling pin) and heuristics (the use of spatial division rather than multiplication of fractions) to construct contextual meanings and achieve outcomes (such as making a recipe in reduced quantity).

#### 4.2.2 Situated action

Humans experience the objective world through actions and activity, and human knowledge about the world is a reflection obtained through such activity. Actions are motivated by the desire to satisfy a perceived need as a result of interacting with an object in a context. Even though the goal of an action may be discretely identified independently of the situation in which it takes place, the practical process of realising the action cannot be detached from its context. Models of action complement models of cognition. Rationalistic models of action generally assume causality, evidenced by some kind of *a priori* planning and subsequent execution—that is, action follows or results from cognition. From a constructivist perspective, cognition, action and context are irrevocably coupled

(Gero 1998a; Gero 1998b).

Schön (1987) pioneered the observation of expert's reflection in action. Schön analysed expert practitioners at work in a variety of disciplines and described a model of their goal-setting and acting. Actions, when learned, become transparent and automatic, just as knowledge becomes tacit. Schön used the term 'knowing-in-action' to refer to the sorts of know-how revealed in intelligent action, in both the public performance (for example, riding a bicycle) and the private actions (such as instantly analysing a balance sheet). In both cases, the *knowing* is in the *action*. When tacit 'knowing-in-action' breaks down, the actor is presented with an opportunity to reflect, and may do so in one of three ways. He may do so after the fact, in tranquillity; he may pause in the midst of an action in a kind of 'stop and think' instant; or he may reflect in the midst of an action without interrupting it. In an 'action-present'—a period of time, variable with the context, during which the actor can still make a difference to the situation at hand—the actor's thinking serves to reshape what he does while he does it. Schön's descriptions address goal-forming behaviours. Rationalistic and situated perspectives differ on the point of how goals are formed. Rational planning theory holds that goals are formulated during planning and as a function of the execution of plans, and that actions are causally related to goals. Situated action holds that goals form and are re-formed in-the-moment, as per Schön's account.

Lave (1988; 1991) also considered the relationship between goals and action. After observing people making decisions about whether to purchase fruit and vegetables, she concluded that actions are not motivated by goals but that goals are constructed, in the moment of acting. Lave concluded that the linear view of action as a means of achieving a goal was a projection, that action is instead not goal-directed but goal-forming. When action is constituted in circumstances (as it always is) an activity and its values are formed simultaneously, and motivation for an activity appears to be 'a complex phenomenon deriving from constitutive order in relation with experience' (p. 184). Lave suggests that 'expectations' (a better term than goals) are dialectically constituted, changed, reversed and inverted in activity, as people initiate action, respond to other's action, and respond to their surroundings. 'People act inventively in terms of expectations about what has happened, is happening and may happen' (p. 185).

Some work environments support this form of dynamic goal-setting more effectively than others. The tension between automating work processes without removing options for the actor to respond to unforeseen circumstances in new situations is always present in system and software design. Suchman responded to this criticism (1994) in the years after the

publication of her seminal book (1987) by explaining how systems should be organised to support situated work. She suggests designing ‘technologies of accountability’ which store (and provide tools to manipulate) the state of the objects for which people in an organisation must be held accountable:

By technologies of accountability I mean systems aimed at the inscription and documentation of actions to which parties are accountable... in the sense represented by the bookkeeper’s ledger, the record of accounts paid and those still outstanding. (Suchman 1994, p. 188)

Suchman suggests designing systems around the ‘objects’ of activity so as to leave the exact sequence of actions to be emergent. This suggests designs that rely on toolkit, workshop or desktop metaphors. To explain an individual’s actions at any instant is to incorporate an understanding of all these environmental and personal factors and to trace the formation and reformation of expectations. Clearly, this is not something that can be reconstructed after the fact. Accounts of situated action, like those of situated cognition, must describe the context and its influence on the design act and design outcomes. This realisation points us toward socio-contextual models and ethnographic research methods.

#### 4.2.3 Investigations of situatedness

Most research of situated cognition and action is ethnographic in nature, and uses examination of situated action and decision-making in specific domains. The following brief summary illustrates relevant research themes. The use of plans in situated decision-making has been researched. Bardram (1997) studied examples of patient diagnoses to illustrate the use of checklists and procedures in highly situated work. From his observations, he proposed an activity theory model based on fundamental units of action and activity. He concludes that plans are useful only as statements of goals, and at best, statements of abstract (but not detailed) actions.

The lack of influence of methods on the designer’s shared understandings of an emerging design has been documented many times. Empirical studies by Curtis, Krasner and Iscoe (1988) and Waltz, Elam and Curtis (1993) found that the communicative mechanisms which support shared social knowledge on a software design team are more critical to the design process than the use of a particular methodological approach. They concluded that active perceptions of knowledge are more important at most stages of the design trajectory than the representations to be found in project or methodological artefacts.

Theories of distributed (or socially constructed) cognition have emerged that explain how

groups converge on solutions in a collaborative design process (Norman 1991). Individuals hold partial mental models of a situation that—while inadequate as a basis for successful action or design—can be combined with those of others to constitute a ‘shared meaning’ and a structure for group ‘sense-making’ (Clegg 1994). The focus is no longer on the individual as ‘decision-maker’ but on the individual as ‘conversation-maker’, both through reflective action and through interaction with other stakeholders in the collaborative design process (Boland et al. 1994).

Situatedness has started to influence software product design. Lueg (1998) incorporated consideration of situated cognition in the design of an internet news reader. His reader avoids the gap between the conventional user profile and the user’s interests at the time of reading, by dynamically constructing a ‘situated’ profile influenced by immediately recent user behaviour (selections, omissions and article viewing times). Lueg’s ‘situated information filter’ helps users cope with high volumes of news content.

### **4.3 Design in Software Engineering**

Not surprisingly, the same rationalistic and pragmatic themes evident in models of knowledge, cognition and action find expression in software’s epistemological base, and more specifically, in software development methodologies. The history of software lifecycle models can be viewed in three broad generations—waterfall, post-waterfall and All-at-once. This progression over three and a half decades represents a gradual deconstruction of the universality of a technology maturity model in the face of increased diversity and complexity of problem types.

#### **4.3.1 Waterfall**

The waterfall software lifecycle model encapsulates the notion that design should follow a path of successive refinement from abstract to detailed specifications, and finally to code (DeMarco 1978). Waterfall lifecycle processes were encapsulated in most of the structured methods (DeMarco 1978; Gane and Sarson 1979; Page-Jones 1980; Sutcliffe 1988; Yourdon and Constantine 1979). The model became entrenched when NASA and the United States Department of Defence demanded waterfall development in their contracts. The resulting widespread and large-scale adoption of the model resulted in reports of problems (DeGrace and Stahl 1990). The assumption in the model (and in the associated structured analysis and design techniques) that the problem description could be documented in complete isolation from the solution design proved difficult for designers

(Zave 1984). It emerged that in practice, designers were more likely to consider aspects of the problem and a range of possible solutions together. Adherence to the model also resulted in a communication gap between end users and developers. Insistence on separately managed phases made it almost impossible for a programmer to influence or correct the specifications passed down to them from the analysts upstream. Participants in the development process interfaced only with upstream and downstream collaborators, and as a result, specialisation was encouraged. This separation of responsibilities and specialisation of skills served management expediency at the expense of cooperation and communication, and the delivered software systems suffered in terms of fitness-for-purpose and extensibility as a result. The parallels with other rationalistic planning regimes such as the Pruitt-Igoe modernist housing complex—and its ultimate fate—are conspicuous.

Boehm (1976) discovered that the cost of implementing a misinterpreted requirement detected during downstream phases increased non-linearly. In using this result as an argument for even stricter requirements analysis and capture, the waterfall model's proponents missed an opportunity to discover the nature of the mismatch between the model and the types of problems it was being used for. The model also contributed to incomplete design—because analysts and developers were encouraged to defer knowledge of the target implementation technology, they could not design certain non-functional characteristics such as exception management or response times (McFarland 1986). The resulting incomplete designs caused expensive programming rework in the testing phase.

Much of the time, DeGrace reports, claimed success with the waterfall model was dubious, because the model's advocates had not actually followed a true waterfall process. For example, reports of success came from sponsors and subcontractors motivated by winning follow-on work rather than from developers. When the high cost and inflexibility of waterfall development finally became apparent, US government employees were encouraged (in 1983) to apply standards selectively and to tailor them before use on a contract (DeGrace and Stahl 1990). The death of waterfall dogma was effectively announced by Parnas' (1986) declaration that the rationality of the software design process claimed by the waterfall model was nearly always faked.

#### 4.3.2 Post-waterfall

These modifications to waterfall orthodoxy mark the commencement of the second era of software lifecycle models. Many had pragmatic origins—programmers modifying their approach to get the job done—rather than extensions to theory or the model, which had



assumed canonical proportions in both academia and industry. The variants generally involved weakening the sequential dependencies between phases or the activities within phases such that they became overlayed or interleaved. The whirlpool (or spiral) variant introduced iteration into the waterfall (Boehm 1988). The timing relationships between phases are relaxed, and the duration of phases in each epoch of the whirlpool differs from those in succeeding epochs. In use, the spiral model addressed many of the limitations of the waterfall model by allowing ‘what’ and ‘how’ activities to overlap within a cycle.

An example of a variant was developed at Boeing (Gilchrist 1989) in which the waterfall proceeds as usual through completion of the feasibility, requirements and preliminary design phases. Then, multiple mini-projects are spawned, each with the goal of implementing *some* of the requirements. Each thread executes the remainder of the waterfall in its own time, and the system completes when the last mini-project completes. Clearly, the top-level functional decomposition must be fixed before the division occurs, and each mini-project must be tasked with a functionally cohesive chunk of the system. Designing and maintaining clean interfaces between the modules is critical to the success of the approach, as is the early identification of a near-complete set of requirements.

Meanwhile in Japan, engineers noted that speed and flexibility of product development were as important as product quality and the predictability of the process. They responded by reducing the number of phases to four (requirements, design, prototype and acceptance) and overlapped them, forming the ‘Sashimi’ model, so named for its resemblance to traditional Japanese sliced raw fish in which each slice rests partially upon the slice before it (Takeuchi and Nonaka 1986). Some overlapping of phase boundaries was not new, but Sashimi’s complete overlap sanctioned the tighter coupling previously considered harmful in conservative waterfall thinking.

All waterfall variants suffered from the need for a substantial investment before anything visible was delivered, which raised the risk of dissipation of project momentum and stakeholder divergence. Prototypes were introduced to reduce this risk by supporting early and continuous feedback. A prototype is a kind of proto-system that is intended to demonstrate or prove some aspects of the intended final system. The objective of prototyping is to clarify the characteristics and operation of a product or system by constructing a version that can be exercised (Agresti 1986). Prototypes are therefore a risk reduction device (DeGrace and Stahl 1990). The prototype itself may have one of two distinct fates—it may become a reference upon which the *real* system is based, or it may undergo a conversion into the *real* system. This conversion is not always performed as

systematically or rigorously as it should be.

Prototyping can be useful for ‘wicked’ problems because it can introduce a partial solution into the environment and stimulate further requirements based on its effect. Designers and users become engaged in the system design process, and both perceive the relative advantages and disadvantages of each design decision. A successful prototype becomes a mediation device and a tangible realisation of the current state of play. Prototypes typically resolve user interface decisions, but not necessarily architectural ones—to avoid this, the software architecture should also be prototyped (Booch 1994). Proposed changes to the prototype’s functionality may then be applied against the system architecture before commitments are made. Prototyping, DeGrace (1990) concludes, is empowering because it moves us away from the ‘revealed word’ of monolithic systems to ‘a place where we can depend on ourselves for our knowledge about solving problems with computers’ (p. 152). It brings things back to a human scale and allows us to make the tool fit the hand, rather than the other way around.

#### 4.3.3 All-at-once

The third era of lifecycle models is grounded in the notion that any attempt to compartmentalise design into a discrete phase is, and always was, artificial. Zelkowitz (1988) questioned the validity of the phase concept after his study of a significant waterfall project revealed that only 50% of the project’s design activity had been done in the design phase—of the remaining half of the reported design effort, 34% was performed in the coding phase, 10% during the integration phase and 6% during the acceptance test phase. He concluded that certain activities could not be bound to phases, and that phases are, under scrutiny, abstractions superimposed over projects to assist with resource planning and budgeting. Like prototyping models, All-at-once lifecycle models (DeGrace and Stahl 1990) support many concurrent activities, but unlike prototyping, the amalgam of activity is focused on developing the system, not a prototype. All-at-once models support the way that experienced practitioners proceed when they know neither the solution nor the problem—when they are trying to solve a ‘wicked’ problem.

Takeuchi and Nonaka (1986) compressed their Sashimi model further by jamming the slices together into a nebulous ball. They termed this model ‘scrum’, after a rugby pack that does whatever is necessary to move the ball downfield. Scrum as a metaphor for software development has appeared in many guises. More recently, Beedle (2000) expressed the idea as a pattern language, eXtreme Programming (Beck 2000) has

incorporated some elements in the form of ‘essential practices’, and the Agile Alliance (Cockburn 2002) has reinterpreted All-at-once ideas in their manifesto on software development. The Scrum life cycle model demands flexible developers and a certain degree of management faith. Flexibility and high skill levels are needed because team members analyse, design and code alternately. Management faith is required because the team is largely self-governing and self-directing—they make progress in multiple dimensions all-at-once, and they are prone to throwing away non-trivial chunks of architecture, design or code when a more satisfactory design emerges from the iterative process. Scrum recognises the intimate relationship between requirements discovery and validation, and that designing and coding are not inherently sequential. ‘When someone writes code, they are also writing requirements, functional specifications, and design notes at the same time’, DeGrace notes (p. 158).

Because Scrum lifecycles empower team members with horizontal responsibilities across the team’s activities and the software deliverables, the model works better with skilled and experienced multi-specialists who are prepared to subjugate the background noise of a professional working life to the consuming challenge of rapid development. Scrum teams become autonomous and members begin to act like entrepreneurs. Team members are freed from the need for lengthy, passive third-person communication, documentation and authorisation protocols. Discussions of All-at-once, Scrum or eXtreme Programming cells often describe the phenomenon of the emergence of a culture. The team builds a shared design capability, and in an environment where efficiency is paramount, even the dialogue between individuals becomes introverted and compressed as the names of software components, idioms and design patterns replace open communication. A Scrum clique reinforces the team’s existence and the individual’s right to membership, and protects the team from external influences that may slow productivity. Team members self-select into particular roles so as to internally balance activities with personalities, for example, ‘gatekeepers’ manage the interface between the cell and external stakeholders (Coplien 1995). All-at-once (or ‘agile’) process models are most suited to dynamic business contexts, but their encouragement of on-the-run architectural design remains contentious (Glass 2006).

#### 4.3.4 The engineering metaphor

Finally, it is worth noting that design in the software domain has been profoundly shaped by the adoption of engineering as the metaphor of choice. There is no obvious point in the early history of software when the suitability of the engineering metaphor as a basis for the

nascent discipline was openly debated. McIlroy (1968) demanded a hardware-like component-based software discipline to reduce risk in the mushrooming United States' Department of Defence software investment. Boehm (1976) first used the term in the context of his separation of a project into design and implementation phases, with a project management regime for each phase. McBreen (2002) claims that the engineering models of the time typically involved bespoke hardware creation or modification, and as a result, conformance to an engineering project regime occupied the software team while the bespoke hardware was developed.

The engineering metaphor implies the degree of formality and repeatability in software construction as is found in other engineering disciplines. Many regard the metaphor as inappropriate and misleading. Eaves (1992b) asserts that there is little reason to argue for a formal basis for software engineering, and that this is unlikely to change in the future. Serious experimentation is impractical and expensive, he claims, because formal experimentation and quantitative information to support comparison between alternative design or development approaches is not feasible—it is impossible to reduce software development experiments to a set of meaningful formal measurements. Even if such a set could be approximated, the number of variables in such experiments could not be controlled, and replication could never be achieved. As a result, he proclaims, 'the standard model of scientific research is not applicable in the domain' and 'we are (still) dealing with a craft' (p. 15). Eaves (1992a) also refers to what he calls a 'rage for order'—the 'human instinct which forces the creation of illusory or aesthetic order out of chaos, if no other order is to be had' (p. 11).

In adopting the engineering metaphor, the emerging software industry revealed its desire to appropriate legitimacy and maturity. Software methodologists have continued to exhibit a propensity to envy the mathematical formalism evident in certain sub-domains of system design—for example, the way relational algebra underlies database design (Baragry and Reed 2001; Glass 1999). Chapter Three's survey of design theory reveals, as Coyne (1995) puts it, the 'plurality of inadequate and speculative models for which there is no unifying theory or overarching model' (p. 238). This qualifies design as a weak (or immature) epistemology. That engineering continues to be questioned as the right metaphorical basis for software design going forward is not completely surprising (Glass 1999).

#### **4.4 Socio-cultural Models of Design**

An appropriate place to conclude this review of design theory and software design is with

some of the socio-cultural models published in information systems research in recent years. These models address both system and software design and provide holistic views of design in context. This class of design model is demarcated from those surveyed so far by its foundation in ethnographic research. Socio-cultural models of systems design are heavily influenced by epistemological (rather than ontological) treatments of methodology use. How individuals structure their thinking, and as a result, how they interpret structures of the methodologies, depends very much on the way they view the world. The philosopher Dilthey (1931) defines this as the *Weltanschauung* (the ‘world images’) of the individual (Kluback and Weinbaum 1957). Checkland defines *Weltanschauung* as ‘the particular non-absolute world image which we take for granted and through which we construct/attribute meaning to human activity or interpret reality’ (Checkland 1981, p. 27).

#### 4.4.1 Ethnographic design research

As a result of the convergence of computing, systems theory and sociology, information systems researchers gradually turned their attention to social science and investigation, including anthropology (Jagodzinski et al. 2000a). The pivots upon which this movement hinges include culture, and a general sense that the ‘complex, ongoing and multi-faceted nature of commercial design projects’ does not lend itself to experimental research methods (Ball and Ormerod 2000, p. 404). Ethnographers have long claimed that a lack of contextual awareness has undermined much social and empirical research in the past. Ethnographic methods such as grounded theory have been designed from a non-positivist perspective that avoid the problems of cross-cultural and context-independent interpretation (Glaser and Strauss 1967; Saule 2000; Strauss and Corbin 1998). Ethnographic research takes a phenomenological basis (van Manen 1990) rather than a causal one, in that it seeks to explicate the experiences of participants through observation and to account for the impact of culture and social environment on observed outcomes. Ethnographic research results are intended to be read critically (as per Coyne’s (1995) critical theme), meaning that the reader takes the opportunity to understand the researcher’s process and question interpretations, rather than to look only for causality, conclusions or digested results. The value of ethnographic research is that new understandings can continuously emerge through such interpretation (Hammersley 1997).

Ethnographic research is distinguished from other research paradigms by *in situ* observation through cultural immersion, and much ethnography implies ‘living with the tribe’ as a member and as a peer. Ethnographic descriptions are rich and multi-dimensional, the observations are detailed, and the researcher must at all times be open, autonomous yet

empathetic—observing, yet self-reflecting. The design of information systems has been a key focus for the information systems research community, and as a result, there is a growing base of accounts and ethnographic models to be found in the design and information systems literature (Jagodzinski et al. 2000a). One early and influential socio-cultural model of systems design is that of Lyytinen (1987) (Figure 10). The model's narrative is straightforward—the Development Group perceives existing systems, alliterates concepts and designs structures in response, and uses language to enable a construction of a representation or form of a changed or new target system to be shared amongst all stakeholders. Lyytinen's framework was significant in its time for raising the issue of subjectivity and the multiplicity of target systems. It also highlighted the centrality of language as the medium of group interaction and the mediator of meaning.

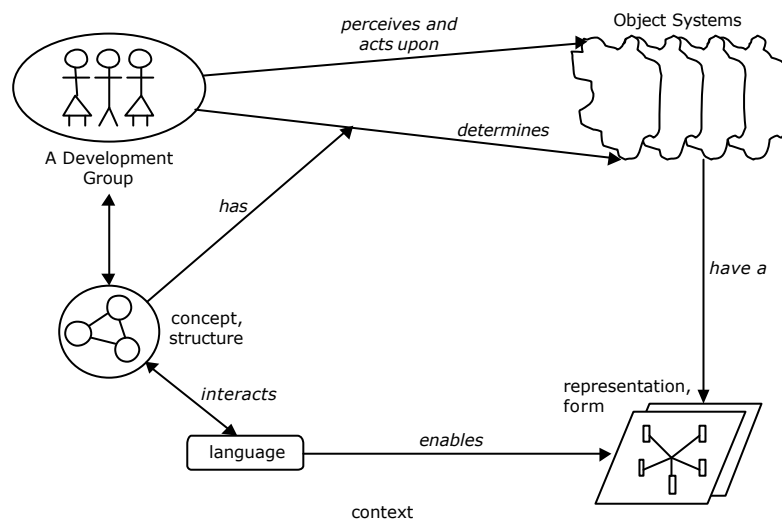


Figure 10: Lyytinen's systems development process framework (Lyytinen, 1987).

Gasson's model (1999) is indicative of more recent ethnographic work. Gasson carried out an interpretive study of an information systems design team at Fujitsu Telecommunications over 18 months, concluding in 1997. Using interviews, meeting transcripts and video, design notes and workshops, she synthesised a comprehensive model of the design team and its context (Figure 12). This model is of interest because it confirms and relates many of the design drivers and contextual factors raised earlier in this thesis. Gasson builds upon Lyytinen's model by elaborating the actors, the context and the target object systems. The actor types reflect the organisational structure and roles present in the typical information systems department of the late 1990's. Gasson's context separates informal and formal

system boundaries, distinguishes between preconceptions, perceptions and a shared representation of the design problem, and the forces that influence each of these. Her representation of the design act shows how investigation of the problem situation and analysis proceed via an iterative process of creating and modifying abstract representations and concrete analogies in parallel—a process that generates new design goals as it proceeds.

		Basis of influence	
		<i>Issues of fact</i>	<i>Issues of value</i>
Scope of influence	<i>Design goals</i>	Framing	Conceptual
	<i>Design process</i>	Interpretive	Symbolic

Figure 11: A framework for the management of meaning in design (Markus and Bjorn-Andersen, 1987).

Of particular interest is Gasson's observation that design influence varies in terms of the dominant form of knowledge at different points in the design's trajectory. Using Markus and Bjorn-Andersen's (1987) framework for 'the management of meaning in design' as a starting point (Figure 11), Gasson identified four types of influence. Initially, the IS manager's existing expertise in defining organisational information system design in terms of *concepts* was dominant. Next, the team engaged in '*symbolic* debates' about the values that should be embodied in the design process, dominated by the views of three individuals who were perceived as having superior understandings of organisational strategy. The next (and longest) phase of design influence occurred as a result of the more experienced team members *interpreting* design issues (and issues of the design process) for the other team members. This period was dominated by the two individuals who possessed the most experience in managing the systems design process. The final phase was driven by external pressures for design closure and was based on *framing* design goals in terms of issues or facts grounded in current business processes, and was dominated by the individual with the most experience in the application domain. Gasson does not claim that this sequence of design influences is universal, rather, that all design trajectories experience these kinds of

influence, and that circumstances dictate their impact and sequence.

Ethnographic research of this kind is not without its critics. One frequent criticism is the lack of ontology, or even the most basic of definitions. When Love (2000) attempted to produce a glossary of design research literature, he found that there were almost as many different definitions of design and design process as there were writers about design. The ‘substantial amount of confusion with respect to the underlying basis of many theories, concepts and methods’ and conflation of concepts drawn from a range of sources has resulted in unnecessary and unhelpful confusion of the terminology of design research, Love claims (p. 295). This problem is not limited to ethnographic research. Cross (1993) compared two reviews of the state-of-the-art in design research published ten years apart and noted how the same terms and concepts were used differently. Oxman (1999) describes how a focus on the dialectic nature of designing changes fundamental definitions of design. Talukdar, Rehg and Elfes (1988) claim that neither practitioners nor researchers agree on what constitutes design activity. Parnas and Clements (1986) argue that, in software design, precise definitions are often not provided, and that there are many terms used for the same concept and many similar but distinct concepts described by the same term. Some of these problems can be circumvented if the researcher states their definitions and objectives with each piece of analysis.



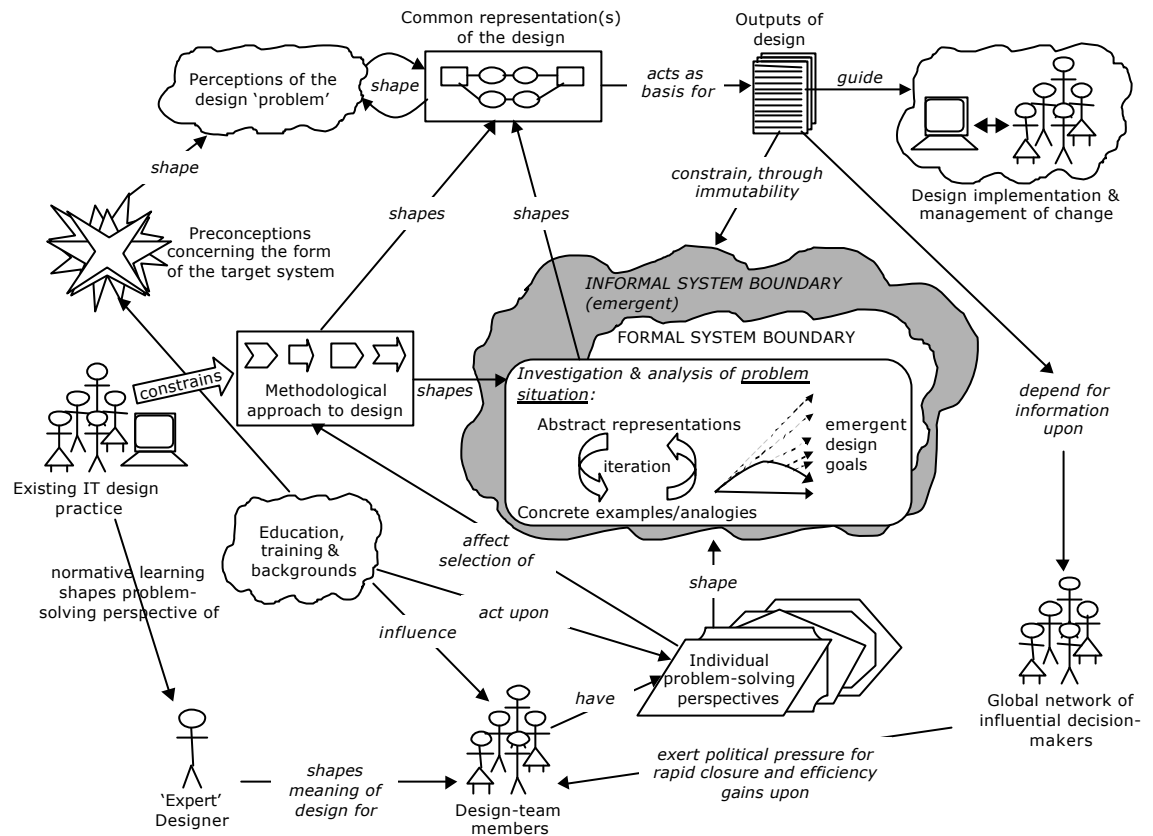


Figure 12: A social action model of the organisational information system design process (Fig 2 in (Gasson, 1999)).

#### 4.4.2 Engineering design research

Ethnographers have extensively researched engineering design. The following summaries illustrate some selected relevant research themes. Amann (1992) observed the casual discourse of a group of expert designers in work settings, in order to describe how designers worked. Amann observed a kind of ‘shop talk’ which, among members of the appropriate science culture, triggered previously non-obvious interpretations or suggestions. Amann drew three significant conclusions—expertise is not an accumulated or inert characteristic, but rather a socially emergent phenomenon; expert behaviour emerges under situated conditions of practice, and the status of an expert is socially constructed.

Busby (1998) assessed the use of feedback in the design processes of five different engineering design firms, finding that feedback to designers was often unreliable, delayed, negative and sometimes missing altogether. There was evidence that designers failed to learn from the feedback that was available, such as repetition of previous failures and the development and approval of plans at odds with previous outcomes. Busby attributes the lack of constructive feedback and attention to feedback processes to the desire to impose

customer management or marketing functions to filter interactions between the designers and users, and a basic over-estimation of the degree to which design activity is ‘feed-forward’ in nature.

Jagodzinski (2000b) studied electronics engineering teams in a large, multi-paradigm research project and concluded that multiple models (or views) of design activity are needed to relate the many perspectives. Jagodzinski concludes that both subjective (‘social relativist’) and objective analytical views are equally necessary in order to understand how design proceeds. Jagodzinski claims that the traditional socio-technical approach in which technical and social systems exist harmoniously but separately must be replaced by a ‘post socio-technical’ one in which subjective and objective views are inseparable.

Lloyd (2000) studied an engineering design and manufacturing organisation that employed 100 people to develop automotive testing systems for UK car manufacturers. The study found that an important dimension of engineering design is social experience, that design proceeds through a series of social agreements (some permanent, some transitory) and that storytelling is an important mechanism in forming these agreements and in building the social construction of a particular design. Lloyd found that these social constructions anchored to particular products, and that it was not straightforward to generalise findings across all of the organisation’s products. For any product, a language is invented which facilitates an ongoing experience of that product and its attendant design processes.

Baird (2000) studied four engine design teams at Rolls-Royce over 11 months, producing what the authors call typical ethnographic output—‘qualitative, diffuse, unstructured, highly interrelated, voluminous’ (p. 339) data from which a number of themes emerged. They found that designers relied heavily upon dialogue and communication outside of their team structures when designing, or changing designs. They found the importance of a culture of ‘judgement-based respect’ in which judgements and verbal promises were frequently interpreted by team members as constituting a commitment. The individual designer’s reputation rested on his or her ability to fulfil such commitments. In rapid time-to-market projects, the emerging design was observed as being biased by the more experienced engineers who tended to draw heavily on previous solutions. They also found social support mechanisms that engineers used to assist in managing time pressures. One of these was the practice of always citing the direct sources of important information when designing (an historical practice that protected the individual but also acknowledged the source and effectively served knowledge propagation). Another was the establishment and maintenance of social links to help engineers change their designs most safely and

efficiently. Another was the organisation's willingness to keep together pairs of engineers who had a history of working well together.

Bucciarelli's (1988; 1994) research in two engineering firms, one producing photovoltaic modules and the other producing X-ray medical diagnosis equipment, uncovered three themes. Firstly, Bucciarelli recognised that the designers coped with the complexity of the product by adopting a separation of concepts. Bucciarelli uses the concept of 'object worlds' to describe the different design spaces (mechanical, geometric, electromagnetic, managerial) and the associated systems of symbols used by the various design stakeholders. Secondly, Bucciarelli (like other design theorists (Lawson 1997)) classified the 'specifications and constraints' (the types of facts and laws) that designers use to restrict the emerging design. Thirdly, Bucciarelli typified the ways designers were observed to negotiate from different viewpoints during the design process as a design discourse. Although none of these notions were new, Bucciarelli used ethnographic methods to illustrate them richly.

#### 4.4.3 Software design research

Ethnographers have also researched software engineering design, although less extensively. Naur (a notable computer scientist) discussed programming in what would now be seen as ethnographic terms (1991). He typified software development as 'theory building'—the ongoing processes of increasing our understanding of an area of concern, in order to find ways in which computer technology can be applied to meet needs. Naur concluded that there can be no right method for theory building as each process unfolds in a unique way, and that the 'life' and 'death' of software depends on the availability of its developers who alone possess the 'theory' enabling them to make meaningful modifications and enhancements.

Nygaard, another prominent computer scientist, regards software development primarily as a social activity shaped by 'perspectives' (1986). Perspectives provide viewpoints from which the observer structures the cognitive processes in which he or she is involved. Nygaard's perspectives allow the designer to understand the software development situation in social terms, where harmony or conflict between participants provides a basis for the construction of conceptual models and where the use situation is anticipated. Naur and Nygaard's descriptions illustrate how constructivist perspectives enter the discourse on software design.

Early empirical research focussed on how programmers approached the program design

task. Davies (1991) performed empirical research on programmers and found that program design was not approached via top-down, functional decomposition as originally thought, but that elements of program design occurred in an asynchronous fashion at any level of abstraction within the solution space. The program design process was mediated by the serendipitous and opportunistic discovery of new knowledge and design constraints. Davies concluded that program design is neither top-down nor exclusively opportunistic, but a mixture of each that is determined by the goal-satisfying preferences of the particular programmer.

Seaman (1999) reported on a study of the use of commercial off-the-shelf software components by NASA software engineers. The study used semi-structured interviews exclusively. The study team had expected to find technical component integration problems, but found these to be minor compared with the problems of administration of procurement, licensing, and contracting. The unexpected diversion of focus from technology to social issues in Seaman's study bears out Busby's (1998) discovery that designers were more interested in reporting collateral activity than technology concerns. This, Busby suggests, 'confirms that design tasks in organisations are as much about social and organisational transactions as they are about individual cognition' (p. 114). Bucciarelli (1988) reaches a similar conclusion.

The reliance on methodology by expert designers is a theme that has received probably the most research attention. Research on the use of methodology adoption and use has generally indicated low levels of methodology use in practice (Carroll 2000). Dekleva (1992) found that practitioners did not regard methodology use as resulting in reduced development or maintenance time, but did lead to better extensibility over time. Dietrich (1997) studied the adoption of software development methodologies in two large organisations and identified that adoption was hindered by the degree of customisation required. Fitzgerald (1997) investigated the use of systems development methodologies in practice, finding a wide difference between the formalised sequence of steps and stages prescribed by a methodology and the methodology-in-action uniquely enacted for each development project. Fitzgerald found evidence that developers omit certain aspects of methodologies not from a position of ignorance, but as a result of a pragmatic assessment of relevance. Fitzgerald concludes that experienced developers are likely only to use heavily customised methodologies and that method adoption generally occurs for abstract, high-level frameworks but not for low-level detailed design activities.

On the reported results of methodology use, Bansler (1993) found that traditional systems

analysis is prone to a view ‘distorted’ by the method’s data orientation, and that subjective dimensions are overlooked in systems outcomes. Bansler recommends system design based upon many inputs of which systems analysis is only one. At about this time, software design researchers began to notice the existence and role of opportunism in design. Khushalani (1994) determined from observations of three expert designers that designers (systems analysts) discovered and adapted their problem solving goals and activities in response to the state of the problem and the environment in which the designing was enacted. Models of the role of opportunism in design followed, that illuminated (but did not formalise) the phenomenon.

Carroll (2000) performed a longitudinal field-based study of an e-commerce system development team with diverse stakeholders, and concluded that the designers ‘crafted’ a ‘unique situated methodology’ by selecting methodology fragments from candidate methods based upon contingent factors. The selection was based upon the architect’s preference and intuition rather than on any shared or formal evaluation. This supports Fitzgerald’s findings. Carroll concludes that the use of information system development methodologies may not be appropriate for all projects, and that the need to adapt methodologies is inherent in the development process.

Ball (2000) used ethnographic methods to investigate the reuse practices of design teams working on large-scale commercial design tasks in four leading international technology companies. The research initially assumed that designers work alternately with three categories of things—unresolved questions about key design issues, solution options, and the criteria by which options are assessed. Their analysis, however, pointed out that designers do not work on questions, options and criteria separately but all at once, in what the researchers termed a ‘focus constellation’, the group of inter-related issues that a designer deals with in a single design episode. The researchers identified that the designer’s memory (rather than literature, design or information repositories) was the most used source of reusable designs, design fragments or information to support designing.

Most significantly, Ball noted that their ethnographic research of design in team settings contradicted their earlier cognitive psychology research of individual designers using traditional protocol analysis techniques. They found that individual designers would compromise the quality or completeness of their designs in experimental settings in order to finish a task, whereas in a team setting they would actively negotiate various design reuse options with other designers to achieve higher quality design outcomes. This behaviour was not—and could never have been—detected in their earlier work using

protocol analysis. Ball terms this phenomenon ‘ecological validity’ and asks ‘whether controlled experimentation has any part to play in design research’ apart from objective useability testing of products and prototypes (p. 415).

Walz’s (1993) analysis of 19 videotaped design meetings over 5 months in a team designing an object-oriented persistence server reveals the emergence of shared knowledge of the domain and the emerging software design as the most critical success factors in system design. The researchers were ‘surprised’ to see how important context-sensitive learning was to the design process, how much information was presented to the team and never captured, and the extent to which ‘knowledge and expertise was the force behind participation and leadership of the design process’ (p. 74).

## 4.5 Conclusion

The situated movement—situated language, cognition and action—emerged from the convergence of sociology and cognitive science in the late 1980s, and was viewed by some as a reaction to the historically dominant classical view of mind. Situatedness holds that all action is irrevocably embodied in context, and that all meaning is consequential on social and contextual factors. For understanding design practice, situatedness demands the construction of rich, multi-layered and (sometimes complex) pictures of design and the designer *in situ*, inevitably pointing to interpretivist and ethnographic research methods.

The history of software lifecycle models reveals some influence from the emergence of the situated movement. First generation rational software lifecycle models such as the waterfall model helped software system developers manage complexity with the languages and tools available at the time, but proved contrived and unresponsive in dynamic contexts. The subsequent generations of lifecycle models largely solved these problems for specific classes of problems. We now face ‘wicked’ problems where solution design must be regarded as a continuous process, where requirements and goals emerge through action and interaction rather than through one-off discovery, and where architectures and artefacts can only ever be partial solutions. It is no longer acceptable to artificially impose constraints on software design to fit within objective, rationalistic models—such a stance is increasingly becoming parochial in a postmodern world.

Ethnographic research methods have uncovered new models of design in context. These constructivist models of design emphasise collaborative, dialogical decision-making, perspectivism and critical analysis of objective and apparently universal ‘truths’. Constructivism primarily serves to unseat orthodoxy. Constructivist-influenced design

methods are motivated by the desire to leave options open, and to stimulate alternative (simpler, enlightened, holistic) solutions.

The journey through philosophy, design theory and design models (in Chapters Two to Four) serves to highlight the nature of each epistemology and their inter-relationships. It also focuses the research question onto design activity rather than externalised theory or methodology. In applying these insights to the refinement of the research goal, we can conclude that statements of cause and effect in the domain of software design are simplistic—instead, descriptions of design must be phenomenological. That is, they must be rich, complex, contextual and complete, implying the need for an interpretive research approach. We can also conclude that waterfall and other lifecycle models provide a historical and pedagogical backdrop but are not prescriptive of how expert software designers perform or express their designing. We can also assert that models of the design act should not be prescriptive, that multiple models are acceptable, that models must be able to be interpreted for different design contexts, and that some form of organising framework is needed to manage this pluralistic approach.

## Chapter 5: Research Design

Hermeneutics achieves its actual productivity only when it musters sufficient self-reflection to reflect simultaneously about its own critical endeavours, that is, about its own limitations and the relativity of its own position. Hermeneutical reflection that does that seems to me to come closer to the real ideal of knowledge, because it also makes us aware of the illusion of reflection. A critical consciousness that points to all sorts of prejudice and dependency, but one that considers itself absolutely free of prejudice and independent, necessarily remains ensnared in illusions. For it is itself motivated in the first place by that of which it is critical. Its dependency on that which it destroys is inescapable. The claim to be completely free of prejudice is naïve whether that naïvete be the delusion of an absolute enlightenment or the delusion of an empiricism free of all previous opinions in the tradition of metaphysics or the delusion of getting beyond science through ideological criticism. In any case, the hermeneutically enlightened consciousness seems to me to establish a higher truth in that it draws itself into its own reflection. (Gadamer 1976, p. 94)

### 5.1 Introduction

This chapter tackles the design of a research method to describe the situated practice of software design. The chapter commences by confirming the most appropriate research paradigm. The mid-portion of the chapter presents and justifies the study's method. The chapter concludes with an assessment of the method that evaluates the researcher's relationship with the participants, the nature of expert recall, generalisability, ethics, relevance, rigour, and issues of hermeneutic interpretation. This chapter completes the literature and thematic survey and research preparation part of the thesis. The second half of the thesis presents the findings from the qualitative analysis of the data and the grounded theory that emerges (Chapter Six), two case studies (Chapter Seven), combined findings (Chapter Eight) and the implications for the research aim and hypothesis (Chapters Nine and Ten).



## 5.2 Applicable Research Paradigms

A research paradigm is a set of recognised concepts that allows researchers to state their position on a number of tightly interrelated issues of research philosophy, intentions, methods and techniques. As explained in earlier chapters, the two relevant traditions of research—positivism and interpretivism—differ to the point of dichotomy. The debate is fundamentally concerned with epistemological questions such as ‘what constitutes knowledge’ and ‘how is knowledge formed’ (Williamson et al. 2000). Positivism describes the objective experimental research paradigm used throughout the modern history of science to ascertain universal truths and test general hypotheses. Most scientific research assumes this model (the hypothetico-deductive model) which in its basic form consists of a theory that states a generalisation, and deductive reasoning which argues from the general to the specific. Research then takes the form of inventing or imagining hypotheses (rather than first obtaining them from logically defensible reasoning) and then seeing whether or not deducible conclusions are consistent with observed facts. A positive outcome constitutes support for the hypotheses which may then be regarded as strengthened, but not proved. Positivist research leads to more reliable knowledge but not absolute truth, because a case that contradicts the generalisation may yet be found (Popper 1969). Thus scientific research is iterative, hypotheses and theories are continually tested as new knowledge comes to light, and scientific proof must always be understood as the best available, rather than an absolute understanding (Weatherall 1979). Positivism holds that the social sciences should be investigated and explained in the same way, so as to pursue the (somewhat idealistic) goal of the ultimate unification of all sciences under common laws.

By contrast, empiricism refers to the grounding of fact in observation and experience. Questions such as how a population behaves or how attitudes are formed do not always reduce to measurable, experimental scenarios (Zikmund 1994). Weatherall illustrates the tension between positivism and interpretivism in claiming that ‘hypotheses are improved by making them quantitative’ and ‘observations are improved ... by making them in deliberately designed circumstances (as experiments) and by using apparatus to produce special required circumstances’ (Weatherall 1979). This, the interpretivists claim, can result in omission or over-simplification of many of the real world’s complexities, in order to achieve a fit with the conservative research paradigm.

Alternately, observation can drive theory in an inductive way (the ‘descriptive-inductive’ model). Inductive reasoning is associated with the hypothesis-generating approach to

research. Daly (1997) offers a simplified cameo of descriptive-inductive research. First, researchers read as much as they can to familiarise themselves with an issue (the literature review), then immerse themselves in substantive, organised observations, from which hypotheses or conclusions might later be drawn (descriptive observations). This approach is based on three different philosophies—empiricism (the need to go somewhere, in this case into habitats), phenomenology (so as to gain an understanding of the subject's point of view), and interactionism (to identify and describe the factors in the subject's environment which might account for why they behave the way they do). This model works well for research where an appreciation of the context is crucial to any meaningful understanding—Daly names environmental science, marine biology and anthropology as examples. In the social sciences, this approach is broadly known as 'qualitative research'.

### 5.2.1 Applicable research types

There are three basic types of research—exploratory, descriptive and explanatory research (Zikmund 1994). Exploratory research requires an ambiguous problem, and addresses the exploration of the problem and its context for possible definitions. Descriptive research requires an awareness of the problem, and works to clearly define one or more problems with which to do further useful work. Causal research requires a clearly defined problem, and attempts to establish cause and effect relationships between variables in the problem domain with a view to predicting, managing or controlling the system's behaviour. The qualitative approach is particularly suited to exploratory and descriptive research, or in general, any situation that requires understanding in depth. Exploratory researchers frequently use qualitative research methods such as case studies and phenomenological studies (Bryman 1988; Shanks et al. 1993). This research project fits in both the descriptive and exploratory categories.

### 5.2.2 Interpretivism

The interpretivist treatment of hypotheses departs markedly from the positivist treatment. Because interpretivist research is typically exploratory rather than descriptive or causal, the formation of a conventional research hypothesis necessitates too many assumptions. Consequently, the interpretivist researcher often declares research themes rather than a tightly formulated testable hypothesis (Carroll and Swatman 2001; Williamson et al. 2000). Such themes do not have to be vague—they must be specific enough to motivate concrete questions and lines of enquiry—but they do not need to be testable in a quantitative sense. Interpretivists do not normally test hypotheses, although they may develop working

propositions that are grounded in the perspectives of the participants. Neither do interpretivists generalise to the wider population, because such generalisations are not quantitatively based and may have no meaning. It is entirely acceptable for a qualitative study to be idiographic (meaning the research consists solely of intensive study of an individual case). In further contrast to positivist research, there is not the same emphasis on replication of results or even repeatability of the study itself. There is recognition that certain phenomena are confined to a particular time and space, and that the styles of observation and explanation which are relevant to one context may not be relevant in another. Sample sizes tend to be much smaller, participants are chosen for their ability to yield rich data, and the need for random sampling is not emphasised as it is in most positivist studies. The interpretivist paradigm looks to a different form of rigour based on the researcher's ability to perform the qualitative analysis thoroughly and relate the results to existing theory. Far from being a 'soft' research option, interpretivist research design and rigorous qualitative analysis is recognised as demanding and requiring high levels of skill on the part of the researcher.

Interpretivist researchers regard their research task as coming to understand how the various participants in a social setting construct the world around them. Interpretivists are essentially constructivists, thus the notion of absolutes or even 'truth' must be redefined in a relativistic acknowledgement of the individual's perception of phenomena (Williamson et al. 2000). They are concerned with the beliefs, interpretations and perspectives of their subjects, and are acutely aware of their own perspectives and how these can bias their observations. Interpretivist research techniques exist to assist the researcher to recognise his own non-neutrality and to factor this into analysis.

The conduct of qualitative research is frequently ethnographic—it is conducted through an intense or prolonged contact with the field or life situation, particularly in situations which represent normal, everyday ones. The researcher's role is to gain a holistic view of the situation or system under study—its logic, structures, patterns, its explicit and implicit rules. The researcher attempts to capture data on actors 'from the inside', through processes of attentiveness, understanding, and by suspending preconceptions. A main task for the qualitative researcher is to explicate the ways people in particular settings come to understand, account for, explain and justify their day-to-day activities. Of the many possible interpretations of qualitative material, the qualitative researcher must determine the most compelling for theoretical reasons, or reasons of internal consistency. Most analysis is done with words, which implies semiotic analysis, or the detection of patterns

through qualitative analysis techniques (Miles and Huberman 1994). In the analysis phase, researchers develop concepts, insights and understanding ‘from patterns in the data’ (Williamson et al. 2000, p.31). This, Williamson notes, is similar to the use of induction in grounded theory (Glaser and Strauss 1967) which is theory that is literally grounded in the field data:

For us, theory denotes a set of well-developed categories (eg. themes, concepts) that are systematically interrelated through statements of relationship to form a theoretical framework that explains some relevant social, psychological, educational, or other phenomenon. (Strauss and Corbin 1998, p. 22)

Choosing qualitative techniques to extract grounded theory illustrates a regard for sociological and phenomenological factors in situated design, whereas the use of these techniques to validate theory from other sources builds or refutes evidence for existing theory from contextual observation and interpretation rather than creating new theory. Good qualitative research can do both theory discovery and theory testing (Fergusson and Shaw 2004).

### **5.3 A Method for Researching Situated Software Design Practice**

The first research design question that must be answered concerns the focus of the research—who or what provides the best source of data on situated design, and how should this be accessed? The alternative foci are the designer and the team. Researching a development team (using ethnographic methods) is difficult for a number of reasons. Firstly, ethnographic research necessitates ‘living with the tribe’, a commitment not easily made by either party. Ball (2000) notes that the ‘often extreme intensity of traditional ethnographic data collection is unlikely to be cost-effective—and may even be impossible’ for most design projects (p. 408). The confidentiality and commercial sensitivities of access to intellectual properties are difficult to overcome in many business contexts. The timeframe over which architectural insights emerge is very long (sometimes years) (Foote 2000), and this makes intense ethnographic research over a system’s lifecycle impractical. Also, team-focussed research can only capture particular design practices if they emerge or are observed during the period of observation. On the other hand, the individual designer can offer his or her longitudinal experiences, and collective knowledge of architecture-related roles on many projects and contexts. Valuable insights will come from experienced designers who have had the opportunity to reflect on practices over many projects and environments. The team cannot yield this view—it must come from the individual designers.

Several important consequences result from this choice. Firstly, the decision to use individuals rather than a team as the study's focus means that the enquiry will uncover the designer's accounts of designing rather than the researcher's observations of designing—these are two different things. A designer's account of how design is done is subject to the individual's interpretation, recollection, and as previously noted, *post hoc* rationalisation and reconstruction. This point is explored further in 5.4.2 (The Nature of Expert Recall). To mitigate this risk, the researcher will use questions that encourage reflection rather than justification, the interviews will not be time-constrained, and a sufficient number of participants will be interviewed so as to saturate categories with convergent data.

### 5.3.1 Defining the Acceptable Participant

The next methodological problem is finding suitable participants. Given the focus on the individual designer, the population of interest is industry-experienced object-oriented software architects within Australia. The term 'architect' refers to the individual's assigned or adopted role as the designer of an architectural solution. Generally, the architect has responsibility for the scope of both the product and the development process. The term 'industry-experienced' suggests some measure of experience in an architect's role. The 'industrial' qualifier excludes academics or teachers who, despite their knowledge of object-oriented languages and technologies, cannot relate practical experience of how architectures are designed and managed in business and industry. Some exceptions, such as academics who have consulted in situations that allow them to relate this experience may be found.

The technology qualifier 'object-oriented' suggests experience with non-trivial object-oriented software architectures, as typified by object-oriented products, systems, frameworks, component or class libraries. Like the measure of experience, the participant's use of object technology in industry will need to be convincing, but need not be exclusive of other technologies. Many contemporary software architectures are heterogeneous through the use of mixed technology components and infrastructure, and very few architects will control a non-trivial architecture consisting only of 'pure' object technologies. Object-based technologies (such as certain visual or fourth generation languages) will be excluded from scope.

### 5.3.2 Research Methods and Techniques

In-depth interviews will be used as the primary instrument of the research. Methodologically, interviews are used when the understanding of the research topic is at a

stage where interaction with participants is required. Through open dialogue between researcher and subject, interviews allow the subject's expertise to be probed, an outcome not possible with other methods. In the in-depth (or semi-structured) interview, an interview outline ensures that all participants are asked about the same topics, but the researcher actively engages the subject to draw out the participant's thinking, experience and knowledge. The advantages of this technique include the large amount of rich descriptive data that can be collected in each interview and the ability to extract explanations (Daly et al. 1997). On the negative side, in-depth interviews are time-consuming to conduct, transcribe and qualitatively analyse, are dependent on the skills of the interviewer, and are subject to bias that cannot be systematically detected or corrected.

Focus groups and action research are two other research methods that have potential application. Focus groups are convened to gain a group's impressions or beliefs on a common topic. Like in-depth interviews, they produce a large amount of rich data, but unlike one-to-one interviews, they have the effect of encouraging some topics of discussion but inhibiting others. One area of critical importance to this study is privacy and commercial sensitivity—company, project and individual's names must be shared only between the participant and the interviewing researcher. This issue alone prohibits the use of focus groups.

Action research comes from the social research genre where researchers are themselves involved in the planning and execution of an activity of change, to 'transform the social environment through a process of critical enquiry' (Miles and Huberman 1994, p. 9). Under the action research paradigm, initial measurements are taken, the researcher effects a change in the environment under study, and subsequent measurements are expected to assess the effect of the change. The situation differs from a classical positivist experiment because the 'experiment' is performed *in situ* (with no control group) and is driven (rather than observed) by the researcher. As a research method for this study, action research will not be pursued because it would necessitate initiating one or more software architecture evolution and design transitions from within at least one (but preferably more) development teams. Finding these teams, gaining the necessary permission and affecting the design episodes, and gaining the necessary agreements for the research to be conducted would also be difficult.

### 5.3.3 Sampling

There is no obvious community or group from which suitably skilled software architects

can be sourced or sampled. One commonly used recruiting mechanism is to use referrals from participants who respond to an initial advertisement or invitation. This approach uses the participant's network of professional associations to identify new candidates, and has the advantage that the person suggesting new participants does so fresh from having just completed the interview. Daly (1997) calls this 'snowballing' and recognises it as a legitimate and widely used recruitment technique.

It is possible that referral subsets will share some similarity—the same employers, business or industry domains, systems, or the same attitudes and approaches to design, for example. This risk will be managed by limiting the size of a referral subset. In the context of the whole study, a mixture of recruitment methods will be used to avoid selection bias. Recruitment will end when the categories appear to be saturated—'no new or significant data emerge, and categories are well developed in terms of properties and dimensions' (Strauss and Corbin 1998, p. 215).

Recruits who respond to an advertisement or an invitation will be requested to complete a simple questionnaire to assess their suitability. If the recruit's questionnaire response indicates suitability, the interview will be scheduled. If not, the recruit will be informed that his or her response mismatched the study's suitability criteria.

#### 5.3.4 A framework for eliciting descriptions of situated software design

With the study's subject identified and a recruitment strategy in place, the next step is to define the set of topics that will be taken into the interviews on an agenda. These topics must be carefully chosen because they steer the participant's thinking and consequently act as seeds for the topics that the analysis will initially deal with. Using characteristics drawn from the reviews of situated cognition and action in the early chapters, the following framework of topics will be used to structure the interview sessions. Table 3 presents this framework which contrasts the rational and situated positions on each topic, with the kinds of questions (left column) that may be useful in introducing these topics into interviews. Appendix C presents the actual interview schedule at the end of the interview series.

<b>Characteristic of the design act</b>	<b>Rational</b>	<b>Situated</b>
1 Identity — who is the designer?	An educated and professionally recognised specialist.	A collective of stakeholders, in which all members are designers in varying degrees.
2 Planning — how is the design effort planned?	A way of achieving the end goal can be defined a priori. Clear criteria can be defined which indicate that the end goal has been reached.	A way of achieving the end goal is hard or impossible to define. The end-state is a consequence of the designer's interaction with the environment.
3 Design seeding — where do designs come from?	From analogy and metaphor to a logically consistent conceptual design that is subject to context-independent verification.	From the emergent properties of the interaction between the designer, context and other actors in the situation.
4 Design Collaboration — how do designers work with other designers?	Within the constraints of a mutually agreed method. All participants must converge on an agreed design frame before implementation can commence.	All stakeholders are designers to a degree. Cognition of the complete architecture is partially distributed. Designers require contextual cues to constitute their design knowledge.
5 Design Control — who controls the design process?	Monolithic, centralized and heroic. Lead designer dictates. Lead designer is appointed for the duration.	Invested in individual designers. Lead designers self-select, roles may change over time.
6 Method — how do designers use accepted approaches?	A prescriptive recipe to be followed strictly.	A descriptive example to be interpreted as appropriate for the situation.
7 Reflection — how and why is self-evaluation performed?	Performed by experts. Based upon quantitative analysis of large numbers of projects.	Performed by each individual designer continuously.

Table 3: Characteristics of the design act—a framework for structuring interviews.

## 5.4 Assessment of the Research Method

The following factors and risks have been considered in the design of this research method.

### 5.4.1 Researcher's Relationship to Participants

The motivation for this research can be traced to the researcher's background in object-oriented software development (Taylor 1992; Taylor 1993; Taylor 1995), technology transition (Taylor 1997) and architecture and design evolution (Taylor 2000a). The close link between the researcher's professional background and the research question is not unusual in qualitative research. In a classification of sources of research problems, Strauss and Corbin (1998) include 'personal and professional experience' and remark that 'professional experience frequently leads to the judgement that some feature of the profession or its practice is less than effective, efficient, human or equitable', and that a professional might adopt a research study in an area to more fully understand or reform certain practices. They disagree that the choice of a research problem through personal or



professional experience might entail additional risk, concluding that ‘the touchstone of one’s own experience might be a more valuable indicator of a potentially successful research endeavour than another more abstract source’ (p. 38). There is also a compulsion for a researcher to research his or her strengths. Dijkstra, when asked by a student about how a research topic should be selected, advised ‘do only what only you can do’ (Dijkstra 2002).

How to best access and explicate the ways in which software designers think about design is not obvious. The parts of the personal software development process that can actually be observed are limited, because ‘much of the software development work takes place inside a person’s head’ (Seaman 1999, p. 558). Participant observation is therefore of limited use because designing in software involves thinking rather than drawing or working with tangible artefacts. Seaman (1999) suggests that software designers reveal their thought processes most naturally when communicating with other software developers, and that ethnographic observations of designers over long periods of time reveal the best insights. Seaman also declares that such research is difficult and expensive to perform, and suggests that interviews provide an appropriate mechanism to collect digested opinions or impressions about important issues of experience. The option to conduct interviews with these designers rests largely upon the researcher’s ability to communicate as a peer. The researcher’s history, knowledge and experience constitute a two-edged sword—they enable research of this kind to be performed, but introduce a discrediting source of bias. This is true of most forms of qualitative research.

#### 5.4.2 The Nature of Expert Recall

Another problem with interviews is the nature of expert recall. When experts of any persuasion are invited to talk of their specialty—as is the case in an interview—all manner of facts, opinions, recollections and reconstructions will ensue. Concerns about the factual reliability of interview data are legitimate. Kotre (1995) documents some insights into the nature of human memory and recall. The first is the power of suggestion—that leading questions can insert ‘observations’ into the observer’s recollection. Loftus (1974) concludes that leading questions can change one’s perception of what happened, and that two kinds of information go into memory—what actually happened, and what information was supplied after the event. With time, the two become blended into a single memory that replaces what was originally present.

A related phenomenon called cryptomnesia accounts for the way that we unintentionally

recall fiction as fact. Cryptomnesia results from remembering *what* someone told you, but not that you were told. It is often subconscious, and is particularly common, Kotre claims (p. 36), amongst groups that do creative work. These and other related phenomena of memory contribute to what psychologists today refer to reconstruction—the convergence of fact, observation and fabrication. Reconstruction holds that memories do not sit passively in one’s consciousness, as do words on a recording medium, but are constantly re-fashioned. Kotre’s conclusion—that we recollect what we want an event to be, rather than what it was—constitutes a telling insight into human nature.

Reconstructive memory is not an accurate source of detail. As time passes we are more likely to recall *what* happened rather than exactly *when* it happened. But this apparent human failing illuminates the real purpose of autobiographical memory. The yielding of *when* to *what* and the metamorphosis of observed fact into a *post-hoc* fiction both point to memory’s purpose as ‘the creation of meaning about self’ (Kotre 1995, p. 87). As individuals and members of a social order, we are better served by a digested construction of interpretations than a row of filing cabinets of un-interpreted factual atoms. Before we can give an experience a lasting place in memory, we have to decide what it means. This process of interpretation, generalisation and abstraction is more personal than almost any other, and serves to condense the otherwise unmanageable volume of memories about events and objects. Outside the scope of this interpretation, we retain only unique events and first occurrences. As a result we ‘remember’ absolute facts, generalisations of classes of actual events that were true most of the time, and interpretations and inferences which recall the value that we projected (and continue to project) onto our past. To interview an expert is to invite recollection of all of these, with prejudice, and without distinction.

Kotre’s characterisation of memory suggests some pragmatic implications for the interviewer—avoid asking leading questions (or allow for their effect in the analysis of the response); recognise ‘always’ and ‘never’ as generalisations; mistrust the accuracy of any quantitative data; accept that a personal recollection may have been appropriated. In interviewing experts, however, some of these concerns are obviated by the line of questioning. We would not normally expect to get detailed and accurate numeric information during an interview, and neither should we expect to solicit historically accurate sequences of specific events from the life history of a project—project artefacts will always yield this information more accurately. If the line of questioning seeks to elicit accounts of experience, attitudes and levels of trust of particular techniques or approaches, intuitive senses and heuristic knowledge, we must expect generalised interpretations

intermingled with one-off or first-time factual accounts. In dealing with these inputs, the researcher and analyst must apply another form of interpretation. The risks inherent in using an expert's memory as a data source must be mitigated by the application of the qualitative analysis method and the researcher's skills.

#### 5.4.3 Generalisability

Another methodological issue is the way in which qualitative study results are considered generalisable, the study's external validity. Generalisability is increased by quality research practice at all stages, from appropriate participant selection, through skilful interviewing to ensure that the discussion is focussed and relevant, to careful and reasoned synthesis of a coherent account from the mass of unstructured transcript data. Uncertainty about the extent to which the experience of the participants represents that of others should be referred to the experiences and studies reported in the literature. In qualitative analysis, generalisability is addressed by carefully arguing the extent to which the results could apply to other groups. This is a difficult task, which many qualitative studies simply evade (Daly et al. 1997). Generalisability is partly provided by making the study transparent, useful, extensible, and clearly connected to the existing research literature.

#### 5.4.4 Ethics

With respect to ethics, this study's design is conventional in its use of empirical and qualitative research techniques. Confidentiality is perhaps the biggest ethical concern, and close attention has been paid to ensure the participant's confidentiality. The study was cleared by, and is subject to Monash University Ethics Committee guidelines (2000-469). The only clarification requested by the committee concerned the ethical use of 'snowballing' as a referral technique, such that no participant would be pressured in any way to provide additional participants.

Broader dimensions of ethical assessment are suggested by Schauder (2000), who entreats the research designer to ask 'To what extent is my research an intervention?' (p. 306). The activity of interviewing software architects has no impact on the architect's development team and is low-impact on the architect's time and workload. The impact of the interview on the architect's self-awareness, thought processes and 'psyche' is likely to be positive, as the interview session affords an opportunity for self-reflection. When the study's findings are formulated, these should provide valuable insights to the participants.

Schauder also asks 'Am I clear about my way of seeing—my theoretical perspective or

perspectives—and can I explain why I have chosen them?’ (p. 309). This question is important because of the study’s reliance on interviews and subsequent qualitative analysis. The need to anchor the perspectives from which the data collection and analysis is performed has motivated the adoption of Coyne’s (1995) four perspectives on information technology—this will be used mostly during the analysis, but also serves to organise the emergence of interview themes. Other frameworks that serve to orient perspectives will be applied in the analysis as needed.

Finally, Schauder asks ‘Do I sufficiently comprehend and respect the extent and complexity of the field of study in which I am engaged?’ (p. 310). It is true that contemporary software development is a field of potentially great complexity, both in the technology itself and in the social contexts in which design is embedded. As noted earlier, the researcher’s background in similar roles serves to respect—and in most places fathom—the depth of this complexity. It is not necessary for the researcher to understand each of the architect’s domains of knowledge, but rather to understand their approaches to software architecture and design within their particular domains.

#### 5.4.5 Relevance

Many information systems research studies (30% in Fergusson’s (2004) survey of information systems journals) do not explicitly state relevance criteria, such as who the study is aimed at, who is expected to apply the results, or even who the expected readers are. An argument for relevance is provided by Keen (1991) who states that relevance must drive information systems research. Keen argues that relevance must precede rigour, and that the high ground of theory is not of itself a justification for less relevant research. Shanks (1994) suggests that the quality of information systems research will be improved if researchers are aware of the advantages and disadvantages of the different research approaches. Hevner’s (2004) widely cited research assessment framework promotes relevance and rigour as the two primary determinants of good design science research. Relevance is not difficult to define—a relevant information systems research study is one that is potentially useful and accessible to its intended audience (Benbasat and Zmud 1999). Relevance is dependent upon an identified audience, and can be assessed by defining the intended audience for each stage of the work. Establishing the relevance of this study is straightforward—this study will inform software architects and developers about situated design, its consequences, implications and costs. Fergusson (2004) further suggests that relevance should be to either the practitioner or the theorist, and that an information systems research design should declare this and be judged accordingly. In

Fergusson's terms, this study primarily targets the practitioner, but positions the analysis in a theoretical frame.

#### 5.4.6 Rigour

Rigour in the qualitative context demands correctness, accuracy, attention to detail, rationality, trustworthiness and adherence to one paradigmatic research model, but is subject to a range of research designs and methodological paths involving those techniques. This openness does not imply a lack of rigour. Miles and Huberman describe qualitative research as 'more a craft than a slavish adherence to methodological rules... no study conforms exactly to a standard methodology; each one calls for the researcher to bend the methodology to the peculiarities of the setting' (Miles and Huberman 1994, p. 5). Rigour may be assessed in terms of a study's validity and reliability. Validity determines what value should be attached to the findings, or to what degree the research approach is really measuring what was intended. A quantitative instrument such as a survey of a known population might have high internal validity (it will yield valid data for the questions asked) but have low external validity (its findings may not generalise well to the population being researched). A small-scale study using in-depth interviews with a few participants might be criticised for having low internal validity, but it may exhibit high external validity because it describes in detail the reality of people's experience. In general, sociologists assess the validity of research by looking at evidence, at how the research was performed, whether anything could have interfered with the research process to obscure the results, and the strength of evidence to support the findings (Hall and Hall 1996). In this study, both validity and reliability concerns are addressed to a large degree by setting the study's two key parameters conservatively—the number of participants and the breadth of their experience covered. The number of participants will be sufficiently large to ensure that the key indicators emerge. As well, the scope of the interviews will be sufficiently broad so as not to miss any key concerns.

#### 5.4.7 The Role of Hermeneutics

Hermeneutics is the study of interpretation (Gadamer 1976; Palmer 1969), particularly the interpretation of linguistic texts, but also of human experience in general. Hamilton (1992) identifies two broad themes in hermeneutics as applied to information systems research—the awareness to pre-understandings or prejudices that the actors inevitably bring to any situation, and context. Awareness of pre-understandings legitimise the researcher's close attention to the participant's knowledge, history, tradition, customs and culture, and

promotes a phenomenological approach. Recognition of the significance of context acknowledges that cognition and action interplay, that cognition cannot be divorced from its situation, and as Winograd and Flores (1986) state, ‘meaning is fundamentally social and cannot be reduced to the meaning-giving activity of individual subjects’ (p. 33). Hermeneutic reflection has the potential to open up new understandings of seemingly known phenomena. For instance, Capurro (1992) observes that Winograd and Flores’ attempt to reflect hermeneutically on informatics resulted in their insight into ‘the non-obviousness of the rationalistic orientation of informatics’ (Winograd and Flores 1986, p. 17). As a consequence, they were able to offer the alternative foundation of language. This recognises the value of interpretivist work over other forms as that of opening opportunities for new understandings:

To understand a text is to come to understand oneself in a kind of dialogue. This contention is confirmed by the fact that the concrete dealing with a text yields understanding only when what is said in the text begins to find expression in the interpreter’s own language. (Gadamer 1976)

The interpretation of the text of an interview transcript is hermeneutic in nature, and as a result, the qualitative analysis of interview transcripts presents an opportunity for both the analyst and the reader of the analysis to build new understandings. For this to be possible, the analyst must strive to make both the texts and the interpretations transparent.

## 5.5 Conclusion

This study will use in-depth interviews to understand the values and drivers that give shape to object-oriented software architecture in industry and business contexts. The sample size will be driven by the needs of theoretical sampling, but it is anticipated that around 20 interviews will be required. Participants will be recruited using advertisements and by referral. An attempt will be made to balance the numbers of participants sourced via advertisements and from referrals.

This research falls within the interpretivist research paradigm and will draw upon phenomenological accounts of experience. The strengths of this approach lie in its ability to represent a reality, as a result of the individual’s continual validation and questioning of assertions and presuppositions about the nature of this reality. The principal advantage (as applied to empirical software engineering research) is that the researcher is forced to delve into the complexity of the problem rather than abstract it away (Seaman 1999). Weaknesses include the dependency on the skills of the researcher and the analyst’s ability

to identify and compensate for biases and preconceptions (Galliers 1992). The study is exploratory and therefore attempts to generate rather than causally test hypotheses. The normal concerns about the rigour (validity and reliability) of interpretivist research are addressed by careful and conservative choices of the participants (and the number of participants) and the ground covered in each interview (Taylor 2001f).

This chapter completes the first half of the thesis. Before transitioning into the analysis chapters, the reader should review Appendix D for the results of the preliminary survey and a detailed profile of the participants. The next two chapters present the results of the qualitative analysis.

# Chapter 6: A Grounded Theory Model of Software Design Practice

I think a lot of what we do, particularly at the architecture and high level design end of building software is a highly subjective game... so many times one person can look at a solution and think it's the best thing that's ever been invented and the next person will look at it and say 'my God, why on earth did you waste your time building all that stuff?'  
—Kahn (*interview participant*)

## 6.1 Introduction

Qualitative analysis of the interview transcripts resulted in five topic clusters—the architect's definition of software architecture, their definition of software design (as distinct from architecture), descriptions of the architect's role and responsibilities, the role of methodology in design, and their personal accounts of 'the design act'. Of these, the first four topic clusters describe the architect's understanding of design and the context in which they design. The architect's accounts of designing and design activity comprises the largest topic and represents the bulk of this chapter's content. Emergent themes include use of patterns and archetypes, the aesthetics of software design, and the significance of 'episodes' and 'breakdown events' in the design of software. These three themes are significant because they represent how the architects organise their design knowledge (patterns and archetypes), the basis of their judgement (aesthetics) and the significant triggers which initiate design activity (breakdowns). Together, these accounts constitute a model of the architect's personal design process.

### 6.1.1 On the use of pseudonyms

Each participant was allocated a pseudonym (the name of an historically significant architect) which is italicised in the text to avoid any confusion with the actual historical



figure. Appendix D profiles the participants in detail.

### 6.1.2 On the qualitative analysis

Each of the five topic maps (Appendix F) is divided into four dimensions—context, purpose, process and concept—as a common structuring mechanism. The subtopics that emerged during the analysis form sub-trees under these classifications. Statements that define relationships or associations between subtopics are modelled in the topic maps as links (relationships). During the analysis, the notes associated with each subtopic and link were used to accumulate relevant extracts from the transcripts, as well as digested summaries. These formed the basis of the emergent grounded theory assertions, which are presented in a table comprised of a cell containing a serial identifier (qualified by the chapter number), a descriptor and the description.

T< <i>n.i</i> > Descriptor.	<i>Description.</i>
-----------------------------	---------------------

The identifier (T*n.i*) allows cross-referencing between grounded theory statements in the discussion. The descriptor is a short-form summary and the description is a concise statement of the assertion. The chapter's text flows as follows—given a significant theme, a selection of the architect's statements are presented and arranged as an argument which builds to a grounded theory statement. These statements comprise the atoms of the qualitative analysis which later chapters use to express further theory, models and overall findings. The data from the interviews was voluminous and all parts of it were analysed to the point of derived assertions. Since space does not permit discussion of every assertion, priority is given to the 'design act' topic map.

## 6.2 What is 'Software Architecture'?

To commence the interviews, each architect was asked to give their own definition of 'software architecture'. A topic map of the architect's responses (and all other statements they made during the interviews that relate to this definition) is shown in Appendix F. Two broad themes—product and process—permeated the architect's definitions.

Those who expressed software architecture in product terms emphasised structural features. *Eames* talked about software architecture as 'a big picture of how it all fits together' and *Mackintosh* similarly described it as 'the big picture, the shell'. *Griffin*

emphasised ‘shape and structure’ as did *Voysey* (‘the fundamental structure, the framework’). The product theme was evident in the way the architects used or implied the building metaphor. *Stickley* described software architecture as ‘what you put in what building blocks and how you join them all together’, and *Piano* likened software architecture to ‘the framework of the building’. The product theme is also evident in the view of software architecture as facilitating composition. *Sullivan* thinks of software architecture as ‘the layout of components’, *Breuer* as ‘a composition of the general objects’ and *Le Corbusier* as ‘units of core functionality’.

Those who expressed software architecture in process terms emphasised ways that it assists the development process. *Johnson* described software architecture as ‘a plan, a design’, *Lethaby* as ‘design at a particular level, a higher level’, *Ruskin* as ‘the way of separating things, a way of breaking things up’, and *van der Rohe* in terms of ‘level of concern’. *Moore* expressed a strongly process-centric perspective on software architecture—he regards all valuable architectural characteristics as being inherently emergent and unable to be designed *a priori*.

Of the twenty-four architects interviewed, the fact that only one questioned the inherent ambiguity and over-loading of the terms ‘architecture’ and ‘architect’ suggests that the terms were familiar ones. *Moore* was the only participant to express a critical perspective. He questioned the value of the term ‘software architecture’ at face value—‘just about anything that you want that is bigger than a class can be slid under the title of architecture’, he says. His wariness points to the divergent intentions of those who use architecture as a way of promoting particular outcomes, and he suggests that the term is at times deliberately kept ambiguous to serve particular agendas. Consequently, he actively seeks clarification—‘the first thing that I do when anybody else asks me about architecture is to try to figure out what they actually mean’, he says.

Both the product and process-centric perspectives on software architecture allow for subjectivity. *Kahn* thinks that software architecture is inherently subjective, and that being objective about software architecture is difficult. He recalls situations in which he and other architects have strongly disagreed on architectures over what ultimately came down to subjective interpretations:

I can think of various people... who are just so subjective about some of their solutions... and they’re solutions that I would look at and go, ‘yuk, that’s really not helping’... by the same token, I’m quite sure they would look at some of the things I’d done and say ‘this could have been better in so many ways’. —*Kahn*

*Gropius* thinks of software architecture in terms of an overarching context in which the process plays out:

[Software architecture is] a collection of ideas, sketches of what the system should look like, a collections of rules-of-thumb or idioms that convey the vision that would let someone that looks at the architecture get a feel for how you wanted it to be built. — *Gropius*

In general, the architects were not strongly polarised to a product or process view and many made subsequent comments that implied the alternate perspective later in the interview. Their particular stance may be as much influenced by what it is they are trying to communicate at the time as by other factors.

T6.1. Architects understand ‘software architecture’ in terms of both process and product.	<i>Software architects tend to define software architecture in terms that align with a product or a process perspective. Those who express product-centric definitions talk either of structure (structure, shape and framework) or composites (objects, components). Those who express process-centric definitions talk of plans, decomposition, levelling and heuristics. The alternate views are by no means mutually exclusive.</i>
---	---

#### 6.2.1 Context and equilibrium

The architect’s descriptions of software architecture referred to the context of software design, including how context enforces constraints and the relationship between goals and requirements. The architects mentioned constraints imposed by the ‘systems panorama’, the development platform, integration requirements and technology.

T6.2. A new system changes the existing business process or technology equilibrium.	<i>Whenever a new system is deployed the equilibrium of systems and information flow will be changed. Architects consider forces in each dimension that has influence on the new system’s acceptability. This includes its development, deployment and use.</i>
---	---

#### 6.2.2 Interpreting requirements and goals

A number of the architects described interpreting—rather than simply receiving without

negotiation—their client’s stated goals and even notionally ‘hard’ requirements. In *Cook*’s definition of ‘software architecture’ the architect and client engage in a relationship to explore goals:

I have to understand what the customer wants... not necessarily down to ‘the button must be here’... but definitely in a ‘what are we trying to achieve here’ kind of way. —  
*Cook*

*Cook* asserts the architect’s right to interpret the customer’s requirements on the basis of a shared understanding of outcomes. This relationship is one in which the architect plays trusted adviser rather than a subcontracted service provider. Understanding the leeway available to negotiate requirements is an important consideration for an architect when an engagement begins. *Piano* also talked of ‘driving’ requirements from the client relationship.

T6.3. Architects may attempt to negotiate the client’s goals and requirements.	<i>Most architects recognise that they have the opportunity to negotiate some of the goals and requirements that the client brings to the engagement. This is because clients often form an idea of what the solution should be and then express their perceived needs in these terms. The architect can often open up the client to other possibilities and in so doing, re-negotiate goals and requirements.</i>
--	--

As requirements are negotiated and mutually understood, the architect forms a personal vision of the solution that serves two purposes—a vehicle for reflecting back his interpretation of requirements to the stakeholders who have been engaged in the negotiation, and a basis for the subsequent design of the solution’s architecture. Some of the architects described it as their place to ‘have the vision... of what the system is to be, and how it is to be implemented’ (*Cook*). *Gropius*’ vision includes ‘a collection of ideas, sketches of what the system should look like, a collection of rules-of-thumb or idioms.’ *Mackintosh* defines his vision in terms of distinguishing the *what* from the *how*—‘I can see how the building is going to look, but I can’t necessarily see how you are going to fasten the walls together without them falling down’. *Utzon* makes the analogy with architects of the built world who ‘don’t decide where to put the light switches’ but conceive and communicate the metaphors and the aesthetic merit. *Gropius* also turned to the built world to explain the purpose of vision:

You couldn't build a building from an architect's drawings... the architect doesn't determine what sort of foundations you should have... they know it has to have foundations, but an expert will determine what type of foundations... and provided it doesn't alter the architect's view, provided it doesn't compromise the vision in any way, there's no need for the architect to know. —*Gropius*

For *Gropius*, the vision is key to whether the system ends up being worthwhile, because it represents the best way to encapsulate the system's primary purpose whilst ensuring the sponsor's involvement and understanding. *Gropius* advocates using the vision as a vehicle for reflecting the architect's understanding of the problem back to the sponsors in their terms for validation.

T6.4. Architect's vision confirms interpreted goals.	<i>The architect forms a vision in response to his interpretation of the client's goals. This interpretation is a function of many things, including the architect's personal approach, history, bias, those of the client or key business stakeholders, and numerous non-technical forces under which the architecture is conceived. If it is shared with stakeholders, it provides a vehicle to clarify project goals and requirements.</i>
--	---

### 6.2.3 Purpose

In defining 'software architecture' the architects made statements about its purpose. Motivations for software architecture were classified in the following sub-topics—bootstrapping a team, facilitating work breakdown, return on investment, and risk mitigation. These sub-topics concern decomposition, management of complexity, and distribution of work effort. Of these, bootstrapping and facilitating work breakdown emerged as the primary purposes of architecture.

T6.5. Software architecture typically serves multiple purposes.	<i>Software architecture typically serves multiple purposes, including the structuring of the software solution, management of work across a construction team, and the pursuit of software quality. The software architect must be considerate of what purpose investment in architecture serves in any engagement. Different purposes may change the success or acceptance factors for architecture and as a consequence necessitate different design priorities.</i>
T6.6. Software architecture establishes a design capability within the team.	<i>One purpose of software architecture is to establish a design and development capability across the construction team. This capability is a product of the combined skills and abilities of the team members. The software architect is its initiator and is responsible at all times for its health and wellbeing.</i>

#### 6.2.4 Return on investment

Because software architecting employs specialist skills and consumes time, the economic notion of return-on-investment has relevance. Investment in architecture may be perceived as an overhead. Overheads are tolerated when they are understood by the key stakeholders (or investors) as indispensable—however, the software architect finds himself, at times, justifying this investment. When the architects raised minimisation of rework as a return on architectural investment, they were asked to state how object-orientation helped them to achieve this goal. Encapsulation emerged as the most important feature. Encapsulation enforces decomposition decisions in source code (class and interface) structures, reinforcing the architecture’s value proposition, *Gropius* claims. Encapsulation provides the primary mechanism for reinforcing architectural structure, enforcing decomposition and maintaining structure over a software architecture’s lifetime. *Le Corbusier* also sings its praises—‘it worked well, because things were actually well modularised... we had simple component interfaces, between each module’. *Le Corbusier*’s strongly encapsulated architectures have proven easy to change, resilient and extensible. Building on strong encapsulation, *van der Robe*, *Breuer* and *Utzon* all named frameworks as a key characteristic of object-oriented software architecture:

You want to express [software] architecture in terms of frameworks... and the driver for doing that is productivity, and de-skilling the development task... if you’ve got good

guys who have done the first bit, then it should be easier for others to come in and add functionality by doing an incremental change. —*Breuer*

Experienced architects separate the difficult implementation tasks from the simpler ones and cast solutions to difficult problems into framework code using encapsulation, leaving the simpler solution parts to derived code, or framework customisations, *Breuer* claims. The return on this investment in design is based on de-skilling the majority of the development team, potentially allowing the use of (cheaper) commodity resources.

T6.7. An economic motivation of architecture investment is de-skilling.	<i>Software architects manage complexity by partitioning the solution and encapsulating complex or stable portions from less complex or unstable portions. Object-oriented languages support this partitioning through encapsulation and frameworks, which allow developers to work independently. Strongly encapsulated architectures and framework structures provide a return on the architectural investment it takes to design them by de-skilling (and therefore lowering the overall cost of) the construction team.</i>
---	---

#### 6.2.5 The role of creativity

The architects almost universally agreed with the assertion that creativity is an essential part of software design. *Pugin* views creativity as driving lateral thinking. *Utzon* talks of creativity opening up possibilities for more elegant solutions—he sees some aspects of software architecture as a projection of designer’s personalities. ‘Every creative act is a creation of someone’s personality, or a group of people’s personalities’, he claims. He sees creativity in every act of software design, regardless of the level of abstraction. Unlike *Utzon*, *Piano* views creativity as having most influence at the architecture level. *Piano* sees a role for creativity mainly to fill gaps where proven, standard reference architectures cannot be routinely pressed into service. Creativity surfaces when the designer modifies or combines recipes by changing elements and their relationships.

Acknowledgment of creativity as integral in software design raises the question of what part of the design act is creative and what part routine. Both *Cook* and *Utzon* agree that software design is ‘all a creative process’. *Cook* regards his ‘ability to abstract things, to generalise, to move forward and make things more specific’ as creative acts. Unlike *Piano*,

*Cook* perceives creativity as an essential part of the design act regardless of scale or abstraction level. If creativity is employed equally regardless of the level of abstraction at which the designer works, then creativity is just as important in component customisation as in designing green-field architectures. *Moore* illustrates the point:

You know, people say, this will be easy to build, it's just an integration, we'll get this and this and this and put it together—I think that's crap... something will have to be created, something new, every time, every system... something will have to be invented... it mightn't be very much, or it might be a lot. —*Moore*

*Morris* defines creativity as putting something together. 'If I was in the pre-computer age', he muses, 'I would probably be a carpenter, or a craftsperson of some sort, because I want to build things with my hands and my brain'. *Sullivan* agrees with *Morris* about the distinction—'you have to create something where there was nothing before'. A software architect is a creator because 'you go into an organisation, you talk to a lot of people, they've got a lot of requirements, there isn't any structure in that (or not the one that you want)—you actually have to create it', *Sullivan* summarises.

<p>T6.8. Architecture's reliance on creativity is independent of domain or abstraction level.</p>	<p><i>The act of design puts an artefact where none previously existed—this is essentially a creative act. The ability to create is a skill possessed by some but not all people. In the case of the design of software architecture, the designer's reliance on existing assets (components, patterns, archetypes or frameworks) does not eliminate the need for creativity. Creativity drives software design regardless of the level of abstraction, and routine design activity similarly complements creative design activity at all abstraction levels.</i></p>
---	---

*Utzon* was the only participant to express with clarity the relationship between creative and routine design. Creativity, which typically opens things up, conflicts with the routine or rational act of resolving, which closes things off. Put another way, methodology attempts to impose repeatability over creative acts. *Utzon* has resolved this tension in the following way—given that 'creative acts [are] happening all the time in a software development process' a methodology must 'allow the space for those things to happen'. As an example, he cites the familiar problem of identifying classes in a business model, which, he says, is



not a rational process:

The only thing you can do is to say, if you hold in mind the purpose of the system, the human brain being what it is, suggests creatively a set of classes, if you work your way through it... then you have to apply some sort of goodness to that—which is the rational act—which is the refining and the testing... and those two things have to go in balance. —*Utzon*

*Utzon* reconciles subjective creativity with rational objectivism by separating the two and allowing each one to counter-balance the other. This is an insightful statement that serves to relate these two apparently opposite and conflicting forces in software design.

T6.9. Architects alternate between creative and routine design acts.	<i>Creative or conceptual design drives the design act that puts a software artefact where one did not previously exist. Rational or routine design verifies the artefact's fitness for purpose and fills in its detail. These two opposing modes of designing must be made to work alternately and constructively.</i>
--	---

#### 6.2.6 Planning tensions

Architects are generally not well disposed to plans. *Breuer*, for one, expressed disdain for planning. ‘I am definitely not a meticulous planner, I am very much an *ad hoc* planner’, he observes using deliberately contradictory terms. ‘I sketch the main lines, and then hope like hell that everybody else will fill them in, and not go over them’. Planning must provide a project framework that ensures visibility but does not impede the unexpected, which many architects (like *Breuer*) rely upon. ‘I think it is extremely important not to rule out the *ad hoc*’, he says.

The architects reported two recurring problems with planning. Firstly, they are frequently left out of the planning effort and their designing must adhere to the constraints of externally imposed deadlines. Secondly, a planner cannot anticipate where the design effort will go. Architects are often required to work within bounded time periods to achieve design outcomes. Experienced architects determine what decisions can be made and what structures to use to provide a degree of insulation from areas that may open up into more design work than was anticipated.

The exploratory nature of software design and development compounds the planning

problem for the architects. Because (as *Sullivan* puts it) the architect has to ‘put something there that was not there before’ the trajectory of every design is to some degree unprecedented. *Cook* likens the uncertainty in design to positivistic notions of theory—because ‘you can’t know what you don’t know’ your theory (architecture) is always at risk of being proven wrong (undermined) by new evidence. Traditional project planning cannot meaningfully impose structure over such a process. One stance, *Cook* suggests, is to take the position that architectural design is an ongoing activity and to demand a continuing architecting task through the lifetime of the system. The architects report that few projects are fortunate enough to have such enlightened management and sponsors.

T6.10. Architects defer design commitments where possible.	<i>Architects are wary of making early decisions that rule out options. They use a range of design techniques that leave extension options open and cater for the unexpected, such as defensive design techniques, extensible architectures, encapsulation and the elements of good object-oriented design.</i>
--	---

Planning tensions follow from the long-standing conflict between creativity and process. In *Pugin’s* experience, creativity is at odds with what he calls the bureaucratic agendas of ‘the process people’. A process-based development methodology, if well designed, can be adaptable, but in *Pugin’s* experience, that is usually not the case, because the cost of changing development processes in large organisations is high. *Utzon* regards the objective of methods as trying to reduce design to a rational process—‘it’s not possible to reduce everything in this world down to logical positivism... it just doesn’t work!’, he declares. He asserts that methods are ‘never going to be able to describe what is going on internally in your mind when you are confronted with that problem’. He sees process as useful for providing a planning framework for a project, but effectively useless in initiating or generating designs. ‘You can plan for creative acts, but you cannot expect them to happen’, he says.

T6.11. Project plans must impose structure over creative tasks.	<i>Software architecture is a tangible (executable) representation of an architect's conceptualisation of a solution. As such, it is volatile in the early architectural design phase and subject to shifts and re-conceptualisation. A project plan must control the allocation of people and time to the design activity during this period. The conflict that may emerge between designer and planner (manager) is not often explicitly managed.</i>
---	---

T6.9 ('Architects alternate between creative and routine design acts') summarises *Utzon's* model of how subjective and objective design actions can coexist. Going further, *Utzon* claims that a process can be defined that includes 'spaces for the things to happen' as well as exit criteria and testing. Designers working within such a process can be given freedom to work independently in proposing structures, components and solutions. After creative design has put the solution element in place, the designer relies on a rational process to test the output of creative design, validating and integrating the designer's outputs. This work of validation is objective, rational and repeatable. 'The rational cannot create anything', *Utzon* proffers, 'it can only criticise and assess'. Any attempts to define rational processes to drive that creative element of design are 'just plain stupid!', *Utzon* concludes.

T6.12. Architects alternate between creative proposing and rational evaluating.	<i>Software architects work in two modes of thought—creative and rational. Creative thinking proposes new solutions and new design conceptions. Rational thinking evaluates the creative mind's propositions. Architects are critical of methods and processes that attempt to dictate or direct the creative part of the design process. Methods, processes and plans can provide a structure within which software design occurs by leaving space for creative acts and supporting rational evaluation of what gets created.</i>
---	--

### 6.2.7 Structure and shape

'Structure' and 'shape' are two terms that the architects used frequently in describing what

they understood software architecture to be. The architects predominantly used ‘structure’ to refer to the way code-partitioning mechanisms are used (including architectural patterns, modules, components and classes). The most frequently mentioned architectural structure was layers. The term ‘shape’ was generally used as a synonym for ‘structure’.

Several of the architects described software architecture in terms of ‘dividing space’. This raises intriguing similarities with the design of buildings by drawing an analogy between physical space and the conceptual space of abstractions and code structures. *Utzon*, for example, describes architecting in software as ‘trying to put guidelines on how you divide the code space up’. That design is a mapping of requirements onto a solution space (or that design is somehow about dividing space) relies on the notion that distance between two points in this space has some kind of meaning. *Utzon* thinks of the code space in two dimensions, so that designing involves deciding ‘which bits of functionality you put where’ and ‘whether or not you put two pieces of code together’ or you keep them separated in the code space, possibly even ‘keeping them as far apart as possible’. *Kahn* also talked in terms of finding ‘the good way to partition the space so that you can, in some senses, practically achieve the project’.

*Moore* talks about structure in terms of three layers—the raw technology layer, the components and their interaction protocols, and the data-flows between components. *Moore* uses patterns as a mechanism to synthesise a design from all of the inputs. *Moore*’s depiction of structure is oriented towards the solution space. Others reported looking for, and finding, significant structure in the problem space. For example, *Mackintosh* comments that ‘the structure is usually defined by the problem space’ and ‘you want to have a solution going with the problem’.

T6.13. Architects draw structure from both problem and solution spaces.	<i>Problem space structures may be reified in object-oriented software architecture as business objects, and solution space structures may be visible in mechanisms in the architecture (such as Model-View-Controller). The architect sources, selects and converges structure from both problem and solution structures as the design proceeds. In general, problem space structure is mapped onto business objects and relationships within a layered software architecture.</i>
---	---

### 6.3 What is ‘Software Design’

The architects were asked to give their own definition of ‘software design’ in order to draw a distinction with ‘software architecture’. The architects fell into three broad categories when describing ‘software design’—those who regarded it as the decomposition of the architecture (this could be called the *product* view), those who regarded it as a process for implementing the software architecture (the *process* view) and those who do not see a distinction between architecture and design other than level of abstraction.

Those who viewed software design as decomposition of the software architecture (the product view) regard software design as distinct from software architecture by virtue of it filling in the architecture’s structural scaffolding. *Voysey* regards design as the ‘components we need to build and to integrate to talk to each other, to achieve the end... as dictated by the architecture’. According to *van der Rohe* ‘a design is a class model that’s going to attempt to achieve what you want to do’. *Mackintosh* defines design as the resolution of how the solution will execute—‘I know *how* you are going to solve the problem from the design’, he says. Design completes when ‘you have come across all the problems, you have got a solution for each of the problems’, and the designed (or elaborated) software architecture provides ‘the structure, the framework that all the other problems that we haven’t thought of yet are going to be solved as well’.

Those who view software design as a process define it in terms of methods, patterns and practices that serve to organise how the design-level components will be completed. *Moore’s* view is that architecture establishes ‘conventions and guidelines for things like specifying design’ and that design is the actual work of software creation. *Pugin* describes design as ‘a blueprint of where you are going to go to... to construct the system, so that people understand exactly what modules and what classes they’re actually going to be working on’. *Lethaby*, *Utzon* and *Morris* are strongly oriented towards the collaborative design processes that their teams follow.

A subset of the architects made no tangible distinction between architecture and design other than the level of abstraction or the audience consuming the outputs of each activity. ‘I think software design includes architecture, I think architecture is a form of design, it’s a high level form of design’, says *Utzon*. ‘I design at the component level and I architect at some higher level’, says *Cook*.

T6.14. Architects do not distinguish between the processes of designing architecture and designing elements of the architecture.	<i>In general, the architects do not describe using different design processes for designing at the architectural level and the component level. Abstraction level is the only characteristic that consistently separates the architect's definitions of 'software architecture' and 'software design'.</i>
--	---

### 6.3.1 Design as 'resolution of forces'

The architects expressed their understanding of software design in different ways, sometimes drawing on analogies or metaphors. Their choices of concept or metaphor illustrate what they consider important. *Utzon's* conceptualisation of design is one of self-similarity (or 'holons') such that the same structures and properties can be seen at every level of abstraction. *van der Rohe* views design as defined by the tools at hand, and *Gropius* explains architecture as a contract and design as fulfilment. *McLuhan* was the only one to have invented a trivet not unlike Vitruvius' three-legged stool—design is a joke, *McLuhan* declares, but he is definitely serious as he explains why he chooses this metaphor:

It's like a good joke! The best jokes I know are the ones where there are a number of elements in the joke, and all of a sudden you spring the trap, and show how they all fit together. —*McLuhan*

A joke sets a scene and introduces tension, then the punch-line springs the trap and the listener is caught off-footed. Design—like the punch-line of a well-conceived joke—reconciles all the elements of the problem at hand to arrive at some solution, and in the best instances, a design's solution resolves multiple aspects of the problem with simplicity and elegance. The joke—like a good design—is minimally structured such that the punch-line resolves each and every element of the story with no forces left unresolved. The metaphor derives extra appeal from the suggestion that the joke's resolution is often not what the listener was expecting. Good design provides the sound, simple, elegant resolution of the forces evident in a problem—a definition that *McLuhan* acknowledges has much in common with patterns.

T6.15. Software design resolves forces that are left unresolved by the architecture.	<i>Software design differs from software architecture in that it resolves forces that arise from (and remain unresolved by) the software architecture. This explains the relationship between software architecture and design—design problems exist as a direct consequence of the chosen software architecture and their solution completes or resolves the architecture. Good software design perceives the relevant forces and delivers mechanisms that resolve these minimally, efficiently and with appeal.</i>
--	---

### 6.3.2 Emergence

A number of the architects raised the issue of whether software design is, by nature, forward-engineered or emergent. Emergent design gained popular exposure with the publication of Beck's (2000) 'extreme programming' manifesto. Subscribers to the theory of emergent design believe that meaningful and useful design is indistinguishable from implementation, that design insights come primarily from implementing, and that design in isolation of implementation (ie. coding) is futile. Critics of emergent design (*Howard, Griffin*) affirm *a priori* design effort—far from being a dated waterfall relic, it can only be dispensed with when the problem being solved borders on trivial, they claim.

T6.16. Architects adopt a preference for explicit or emergent design.	<i>Software architects exhibit a personal preference for how much effort they are prepared to invest in indirect manifestations or representations of design. At the explicit end of the scale, a software designer would elaborate a set of architectural models with design-level detail before committing them to code. At the emergent end, a designer would go straight to code without any external models or representations. However, emergent designers acknowledge thinking about the design, sometimes in a semi-formal sense, before and during coding. The drivers of this personal preference are unknown, but may include the individual's visual orientation or their ability to abstract from flat (textual) representations (such as code) into models or structures.</i>
---	---

## 6.4 The ‘Software Architect’ Role

The architects made many statements about the role of the software architect. This section presents some analysis of selected sub-topics that were interesting because they were in some way pragmatic or unexpected.

### 6.4.1 Architect as salesperson

When *Morris* defined ‘software architecture’ he stated that ‘everybody [who] wants to push their own barrow is happy to push it under the architecture banner’. *Mackintosh* agrees that ‘architecture is not always decided upon by technical or rational reasons’. *Kahn* observes that ‘our industry is somewhat prone to people... who are in power plays... and being an architect is one of them now’. The software architect is at times caught in a compromising space between sales and delivery. *Breuer*, *Lethaby*, *Mackintosh* and *Le Corbusier* all recounted stories about how they had at one time felt pressure to force-fit a particular software technology or their employer’s product into a situation, against their best judgement. Some stories from these participants illustrate the theme.

In the late nineties, *Mackintosh* and *Le Corbusier* were both involved in what was perceived to be a successful project using a combination of three object-oriented technologies (ObjectStore, OpenUI and C++). At its conclusion, the client offered the sub-contractors a larger, more lucrative project on the condition that they use the same technologies and associated development processes. *Mackintosh* claims that at the time, he saw a significant mismatch between the follow-up engagement’s problem type and the mandated technology set. He believed that a relational database—not an object oriented database—would be the best solution technology. ‘It would have been a quarter of the complexity, ten times as fast’ if it had been built in Oracle, he claims. But the client associated the previous project’s success with the technology set and would not entertain change. In this case, the client associated project success with the combination of people and technologies and was not prepared to change either, even though the problem was entirely different. *Mackintosh* subjugated what his technical judgement was telling him to his desire to win the follow-on work.

*Breuer* recalls a project where a high-value engagement (and probably his job) depended on a client choosing to use the company’s software product, against *Breuer’s* professional judgement:



So I was put in a position of a stimulus in that they [the client representative] were saying, okay, this product that I was representing, I was a consultant on, they were saying would not do the job... so they [Breuer's management] said, well okay, we have got to make this product do the job... when he [Breuer's manager] put it like that he encapsulated and put precisely what the requirement was. —Breuer

*Lethaby* recalled a project that failed despite his attempts to design the architecture to get around the problem. The solution architecture had been designed to rely on a database server at several dozen hosts, meaning that the incumbent database vendor stood to make substantial license-based revenue. *Lethaby* independently designed the architecture to allow for other options for persistence and data exchange, but the project's management, under pressure from the vendor, could not deal with the architectural re-configuration and terminated the project:

So if you are in the business of selling labour, or selling things that sit in the architecture, then you can use the architecture to beef up your sales if you think you can get away with it. —*Lethaby*

T6.17. Architect's designs may be forced or constrained for commercial reasons.	<i>Architects may inadvertently become players in sales or commercial negotiations. The architect may experience pressure to recommend a particular technology, to select one design option over another or to suppress a preferable design option for commercial rather than technical or aesthetic reasons. These situations can present significant ethical and professional dilemmas.</i>
---	---

#### 6.4.2 Impact of team capability

Given that one of the purposes of software architecture is the management of work across a construction team (T6.6) it should come as no a surprise that architects take the capability of their implementation team into account when designing the architecture. This goes beyond selecting the team's developers for their product and technology skills to accounting for changes to the structure and shape of the architecture. *Kahn* describes selecting decomposition and structuring patterns that deliver an architecture that maps to the team's implementation skills. Designing in this fashion requires a high degree of awareness on the part of the architect, and to what extent *Kahn* or others actually change

the architecture to map onto their team's skills profile is difficult to assess. *Mackintosh* hints at where the line is drawn—'you consciously limit the complexity of the design because you realise who is going to build it and who is going to maintain it', he claims.

T6.18. Team capability constrains architecture.	<i>The architect's perception of the capability of the delivery team constitutes a constraint on software architecture. The architect may modify aspects of his design—the basis of decomposition, the mechanisms used, the overall complexity of the solution—to match his perception of the team's ability to realise the architecture. Architects may use framework structures to isolate complexity and expose simplified abstractions to parts of the team.</i>
---	--

#### 6.4.3 Illusion of progress

A number of the architects talked about the need to manage stakeholder perceptions of design progress. Architecture and design phases on large projects can last for many months, and during this period external observers who are not technically versed may perceive much activity but little apparent progress. 'If management don't see anything visible, then they are not happy, they think that nothing is actually being done', *Eames* says.

T6.19. The architect manages the visibility of architectural design progress.	<i>The architects report that (in general) managers and business stakeholders do not understand the process of software design. This is partly due to the invisibility of conceptual design work and also the radical nature of software design (particularly the propensity designers have for throwing away designs and starting again). The opportunity for architects to educate managers is often not present nor appropriate. Therefore, architects must ensure that stakeholders can see tangible progress during periods of architectural design and that selected aspects of the design process are transparent.</i>
---	---

## 6.4.4 Influence of prevailing culture

Most of the architects described how they had felt compromised at some point by unreasonable pressure to deliver software in short timeframes. Some attributed this to business culture. *Gropius* and *Voysey* traced the influence back further to societal culture. They opined that Australian business and industry does not have a culture of investing in architecture in the way that you might find elsewhere in the world.

T6.20. Prevailing culture directs attitudes to investment in software architecture.	<i>The prevailing contextual, business and even societal culture can affect attitudes to investment in software architecture. Culture shapes the context in which the software architect designs and may constrain what the architect can achieve. If the prevailing culture does not value long term investment, or investment in infrastructure, then the argument for investment in quality software architecture becomes harder to make.</i>
---	--

## 6.4.5 Career investment and subversion

Many of the architects mentioned the effects of career and motive on their experience of architecting software systems and working with other architects and developers. *Pugin* laments that ‘most places that I’ve ever worked are places where people are just trying to push themselves up the career ladder, not actually do anything properly’. *Utzon* is unconcerned about exposing his own career motives—‘I’m not altruistic, there’s nothing particularly spiritual about building a computer system’, he says:

The system itself is not the most important thing with most people, right? If you actually scratch the surface they’re not really there to build that system... they’re there to get what they can out of the process of building the system. —*Utzon*

Many of the participants related stories of how colleagues have chosen particular implementation technologies to freshen up their resume rather than on the basis of suitability or project risk. *Mackintosh*, *Le Corbusier*, *Voysey*, *Moore* and *Eames* all talked of how the use of a current software technology is an attractor for skilled professionals. Use of an inappropriate technology, chosen for the wrong reasons, significantly increases project risk.

T6.21. Market demand for skills represents a form of technology bias.	<i>Software professionals are sensitive to the rate at which their skills in a particular technology date. As a result, current market demand for skills in a new technology may inappropriately bias technology selection or use. The architect must manage his team's (and his own) desire to inappropriately use a project or engagement as a means to gain exposure to a new or 'desirable' technology.</i>
---	---

## 6.5 Methodology

The architects made many statements about how they use methods and methodology in their designing.

### 6.5.1 Use, misuse and dissatisfaction

Because methods and methodology address a broad range of solution and system development concerns, the architects were specifically asked about the role of methods or methodology in supporting the design of software architecture. In response, a number of the architects expressed dissatisfaction with methods, methodology and process. In general, their comments reflect a perception of methodology as a mechanism for inappropriate forms of project control. They reveal a suspicion that methods de-skill or unseat the expert practitioner. *Kahn* summarises the sentiment:

I guess I would say that I'm still struggling to find a methodology that works, that encompasses everything I need to do in the timeframe I need to do it in... I've seen glimmers of things that help, might work, and at the same time I've been presented with stuff where I've outright been able to say 'this doesn't help, I know this is not right'. —*Kahn*

*van der Rohe* moderates his dissatisfaction with an acknowledgment that he has not seen methods used diligently:

I have never been in a project were I have seen them contribute any positive benefit to the project—ever... I haven't been in many projects were they have been assiduously used either. —*van der Rohe*

*Kahn's* 'glimmers of things that help' suggests that he does perceive some value in methods. Other architects confirm that methods contain useful techniques and themes worthy of

appropriation:

I have never followed a single methodology—but have used the tools that each methodology provides that are good. —*Kahn*

You pull out the bits you think are good, you justify the bits you don't, and that's the way I tend to use them. —*Mackintosh*

*Gropius*, like most of the architects, denies having 'any good experience with any methods'. But he also acknowledges lifting tools and techniques from methods:

What I've found is there are a small set of tools and techniques that are scalable... interaction diagrams, sequence diagrams... they're a very scalable tool, and you can use them at all different levels... I think if you reuse the same tools and techniques, if you are able to propagate them through different levels of the process, then that becomes powerful. —*Gropius*

Sequence diagrams, *Gropius* claims, are 'scalable'—they are independent of programming language or even paradigm and can be used to model interactions between any objects or parties at any level of abstraction. Their relative simplicity allows them to be used to aid communication with both business stakeholders and developers. These characteristics make them convenient and ready at hand. 'I'm a big supporter of tools... not much of a supporter of methodology', *Gropius* concludes.

<p>T6.22. Architects hand-pick useful techniques and themes from methods.</p>	<p><i>Architects do not rigidly follow methods when designing. They do not appear to champion or otherwise advocate the use of methods to assist design on their projects. Instead, architects use methods and methodology as a source of design techniques. Examples include the selective use of UML and RUP models (use cases, class, sequence, state and package models were explicitly mentioned). Architects assemble their own 'toolbox' of design techniques that they have found useful over many projects, and with time, their knowledge of how to use these tools becomes tacit. This know-how constitutes one of their most important design assets.</i></p>
---	---

## 6.5.2 Risk mitigation and transfer

Some of the architects described methods and methodology as a mechanism for mitigating or transferring project risk. Risk mitigation is sensible project management practice, whereas risk transfer implies avoidance of responsibility. Although not an advocate of methods, *Mackintosh* suggests that they do mitigate risk by providing a checklist to ensure that things do not get overlooked. *van der Rohe* is more condemning of the motivations of those who would promote methods and views them as only providing risk transfer:

What have they [methods] contributed? Arse covering! Risk transfer by the 'keeper of decisions'—what are the things, the criticisms that might be able to be held against me later? And how can I prevent those criticisms being asked? —*van der Rohe*

Methodology, *van der Rohe* claims, is used as a form of insurance. Methodology has become sufficiently visible to business sponsors that it is now 'something that someone could be criticised for *not* doing'. Methods, like project plans, are an artefact of the professional project management culture—by creating activity (some of it nugatory) they justify bureaucracy and ancillary management and administration roles on projects, *van der Rohe* claims.

T6.23. Architects regard methods as mechanisms for transferring project risk.	<i>Methods may be mandated in certain circumstances as a mechanism to transfer risk away from project sponsors and stakeholders. In such cases, the architects will likely mistrust the method and adhere only minimally to it.</i>
---	---

## 6.5.3 Technology churn invalidates method detail

Some of the architects commented that the value of methods is reduced by the constant changes to the base of software technologies. Software technology changes date methods and design techniques, reducing their relevance with time. *Pugin* illustrates:

Every time I get to point of feeling happy about how to use a particular technology I'll never use that technology again, the next project uses something completely different—you have to reinvent yourself, or take what was meta-methodology and take that across. —*Pugin*

*Pugin's* comment on taking 'meta-methodology' across the generational technology divide exemplifies one way that architects cherry-pick techniques and themes (T6.22). Several of

the architects criticised software development methods for avoiding a close dependency on software technologies by evading coverage of the software design part of the overall development process. *Utzon* also describes meta-methodology (as ‘architectural precepts’), or the separation of changing from unchanging elements in the presence of technology churn:

So the architectural precepts that I think I come in with are pretty much applicable every time I go to do an information technology application... and they don’t stay rigid because as each new technology comes in it informs something about the architecture and the architecture’s revising. —*Utzon*

<p>T6.24. Architects take ‘meta-methodology’ forward across technologies and projects.</p>	<p><i>Over time, technology churn dates and eventually invalidates method detail. Architects adopt techniques and themes from software design methods and take them forward, adapting and evolving them as they are applied to new technologies and design situations. These are a digested and personal (ie. tacit) form of knowledge—the architect may describe them as ‘precepts’, ‘principles’ or ‘meta-methodology’ rather than methods or techniques.</i></p>
--	---

#### 6.5.4 Relationship between experience and method

Some of the architects described, albeit obliquely, how they understand the relationship between experience and method. While they claim not to need methods, they do use methods as sources of techniques that they may then choose to appropriate. They allude to a tension between experience and method that may derive from control of the design process or from their desire to protect the perceived value of their expertise. This tension comes from the recognition that methods claim to commodify architecture and design capability via processes, templates and recipes. This is a tension that rises whenever an expert’s tacit knowledge is externalised into a form that can be shared or followed by lesser skilled professionals.

When the architects reject methodology in favour of personal experience as a primary driver of design, they must account for how they draw on their experience to replace the stated goals of the displaced method. In other words, they should be able to justify how they ensure design quality and consistency across a development team and how their ‘box

of tools’ helps them achieve these design goals. There is evidence that suggests the architect’s ability to achieve these goals is highly dependent on their personal skills and leadership capability, and in the absence of a skilled architect’s leadership, other factors become influential.

*Stickley’s* company used Mentor (Edwards 2006) over a five year period, which he describes as more of a process than a design methodology. For software architecture and design, they used UML and Rational ROSE, which resulted in a common representation for all models. *Stickley’s* account of their use of methodology highlights the difficulty of enforcing adoption, especially amongst groups of experienced professionals:

But we don’t really, I don’t think we have a methodology *for design* in this organisation, I think different people do it different ways... the thing that ties it together is probably that it is all object oriented... it still is, we haven’t regressed to functional design. —  
*Stickley*

*Stickley’s* realisation that consistent use of modelling tools and a consistent design methodology are two different things illustrates a subtlety of methods that escapes many software professionals. In *Stickley’s* company, architectural consistency and software quality are notionally enforced by the use of common modelling tools but in reality, the product’s design is influenced more by the individual designers working across the teams than any established quality system, process or method. Elsewhere in the interview he bemoans the inconsistent architectural quality across the product’s extensive code-base. *Stickley’s* employers would claim adherence to a proven software development method, but as *Stickley* reveals, the architecture is littered with the signature styles of a dozen or more designers past and present.

In observing that ‘the thing that ties it all together is probably that it is all object-oriented’ *Stickley* identifies a kind of lowest common denominator design quality phenomenon. In the absence of both a methodology for software design and strong technical enforcement, the product’s architectural consistency was never raised above the level enforced by the object-oriented languages and tools in which it was built.



T6.25. Tool support defines the ‘lowest common denominator’ in architectural consistency.	<i>In the perceived absence of viable software design methods and processes, the architects rely on their own experience. However, architects find it difficult to enforce their vision consistently, especially across large projects or teams. In general, the lowest common denominator in design quality and consistency that will be achieved by a team will be that which is enforced by the software development tools being used.</i>
---	---

In the end, *Kahn* suggests, a handful of experienced people must step up and take the lead, mentoring the less experienced in the fashion of vernacular knowledge transfer. *Kahn*, like *Stickley*, has reached this point after having rejected methods as viable software design drivers.

T6.26. The architects reject methodology as a universal and viable driver of software architecture.	<i>The architects reject established design methods or design processes as a primary driver of software design. That is, the architects claim to own the design process, often exclusively, rather than mechanistically following an external best-practice process or method.</i>
---	--

## 6.6 The design act

In the light of their rejection of methodology as a driver of design activity (T6.26), the participants were invited to reflect on the personal act of design at length. This section describes the main themes. To commence, participants were asked to describe their personal approach to an architectural design problem. *Le Corbusier’s* description reads like a tutorial in object-oriented analysis—he talks of abstracting (‘call[-ing] up the real world’), ascribing behaviour to the structure, modelling the interactions, then getting the abstractions to ‘converse to each other so you can say that it is going to provide a solution... not [at] the object level, but messages between components’. Others were less sure of the exact actions, steps or sequences of their personal design *modus operandi*. Some architects were unable to describe the detail of their personal design process, but trust their ability to perform the act, or more specifically, to know a good design or abstraction when

they see one. *Pugin*, for example, described his approach as being initially *ad hoc*, but likely to converge to an *in situ* process over a relatively short period of time:

In some ways it seemed fairly *ad hoc*... I actually ended up with handwritten pages of things that needed to be done... I would get to a particular point and things would start to suggest themselves to me. —*Pugin*

In describing his personal process as one of ‘design by doing’ but then ‘quickly converging to a process you intend to use’ he infers two phases—a first where he opens up the design process and allows discovery and emergent themes to suggest a process, and a second in which he converges and optimises that process as he executes it. At no time does he explicitly document or externalise his process. To the observer, such a process would appear largely *ad hoc*.

A problem with intuitive skills like *Pugin* describes is that they are not amenable to management. Not even the designers themselves can always be sure of being able to perform the design act under all conditions. Design act ‘performers’ can get stage-fright, referred to by the architects as writer’s block. ‘You look at something and you think ‘so how do I get started on that?’ *Pugin* recalls. *Kahn* agrees—‘the blank page is the most dangerous thing’, he says. Their antidote is the same as is recommended by literary writers—‘you’ve just got to get on and do something... go for a walk in the park, let the ideas go round in your head until you get something, and you see what you should be doing’, *Pugin* recommends. *Kahn* uses the same strategy. ‘As soon as you start putting things on paper then you realise that you’ve got ideas there, and you can shuffle them around, and try and find the best fit of everything’. The key to getting started is the core user requirements for the system:

If I’m having difficulty getting started, I’ll try and go back to some fairly fundamental user-driving aspect... and that draws my first box... and in order to satisfy what it needs I’ve got to have something fit in behind that. —*Kahn*

<p>T6.27. Architects assemble their personal design process early in design engagement.</p>	<p><i>In general, when the architects approach a design task, they do not have a common way of starting or an externalised process to invoke. The architects initially appear to search for a path forward—trialing ideas, sketching and forming concepts, in an apparently ad hoc fashion. As well as trialing early design elements or options, they are also trialing a personal design process for the design situation they find themselves in.</i></p>
---	--

### 6.6.1 Problem space considerations

The architects were asked how they consciously worked to understand the problem space, and if so, what form this activity took. The architects recognise that different ‘types’ of problems exist. Most described ‘recognising’ problems, and several described attracting work specifically because they had solved ‘similar’ problems before. *Cook* states that ‘the most important thing you have to know’ as an architect is ‘how to adapt... learn and... understand problems’. He refers to an ability to isolate and recognise problems (‘to generalise problems from one form into another’) and a related ability to associate a current problem ‘to fit something else that you understand’. This problem-matching and mapping skill is significant because it suggests the existence of a small set of problem types which experienced architects can ‘see’ when confronted by what appears on the surface to be a new problem description.

Architects work in both the problem space and the solution space when designing. The problem space is the source of business constraints and may include ontology, stakeholder views, business processes and rules. The solution space is the source of technology constraints and is comprised of the stuff of architectures—solution concepts, solution ontology, solution models, solution archetypes, components, connectors, code structures and patterns. When the architect designs software architecture, he bridges these spaces with a structured solution that maps requirements and significant problem space elements to elements of software architecture. *Cook* describes a scenario where he used this kind of ‘problem mapping’ with success:

If I think back to the project that I think was my most successful... one of the great things that we managed to do... was to continually take problems, and re-evaluate them in the context of what we’d already done, and say ‘well OK, this is really just a

special case of that’... ‘why doesn’t [the architecture] already handle that?’ ‘Oh look, if we re-factored it a certain way it would still do all the stuff it was doing before and it would handle the one special case... great! Fantastic!’ —Cook

Not all of the architects regard the problems they are confronted with as immutable—many, even most, are negotiable. This suggests that problem space constraints are not purely objective. *Kahn* notes that the architect is often in the position where he can ‘drive the changing of some of the constraints’:

My experience with most customers, although they’ll come to you saying ‘this is what I want’, it’s more a statement of ‘this is what I believe I want’, and if you go back to them and say ‘well that’s nice, but do you realize it would work better if we did it this way’... you have to be aware that there’s a flexibility there—and if you can come up with a better approach for them and sell it to them (ideally in such a way that they think they came up with it themselves), then yeah, there’s variability there. —*Kahn*

T6.28. Architects negotiate problems to match known problem types.	<i>At times, the architects may direct their negotiation of requirements (T6.3) in order to make a presenting problem look more like one which they have had prior experience of.</i>
--	---

### 6.6.2 Solution-based constraints

Architects may manipulate constraints during the design process. A number of architects described ways that they consciously emphasise or de-emphasise constraints relative to each other in the early phases of design. For example, *Kahn* stated that he may choose to ‘ignore or relax some constraints’ during his ‘first level of structuring’, after which he might ‘start to tighten up the constraints or introduce additional constraints that you know about’ during ‘a second iteration or a refinement to that first cut’. His strategy of being selective in the use of constraints to form a scope and a consequent architectural structure avoids having to deal with all known constraints in one sitting, which can lead to ‘analysis paralysis’ where ‘you’re sitting there worried about whether or not you’ve satisfied every constraint and never actually running the potential scenario’. Constraints change, and so trying to satisfy every constraint ‘with your first idea straight out of the gate is really not going to work’, he claims. Constraints and the solutions they prompt are inter-dependent—the designer’s response to the first subset of constraints has the potential to

change the next subset, and so on.

At certain times when designing, software architects deliberately select and impose constraints on the solution space in order to manage complexity and move the design process and the solution forward. For example, a designer may choose to reverse a decision to use Layering in an architecture in order to avoid inter-layer communication or parameter-passing overhead. *Moore* describes designing in terms of successively interpreting and applying increasingly fine-grained constraints:

You start with this huge kind of big blob, and you start hacking into it, hacking pieces away, constraining it, by the time you get to the very end and you're just touching it up... I quite literally imagine myself sculpting. —*Moore*

An important source of constraints on software architecture is the target platform. The adoption of a technology platform such as Sun's Java 2 Enterprise Edition (J2EE) brings a raft of architectural constraints that significantly changes the task of doing architectural design. The J2EE architect is focussed more on detecting and designing for differences (from the J2EE reference architecture) than designing from scratch or for a less prescriptive technology platform. In selecting a technology platform, the architect must balance contradictory requirements such as simplicity with non-functional requirements such as scalability and reliability.

T6.29. Architects alternatively relax and tighten solution space constraints when designing.	<i>Some of the architects report alternately relaxing and tightening selected constraints in the solution space in order to separate design iterations and to simplify the number of constraints in any given design iteration. For example, an architect may 'selectively ignore' functional or non-functional constraints or requirements (such as performance, security or scalability) then consider the effect of these constraints in subsequent design iterations. The choice to selectively ignore a set of constraints is not made randomly, but with the knowledge that parking them will not invalidate the design or divert its trajectory.</i>
--	---

### 6.6.3 Candidate solutions

Most of the architects report that they routinely consider different solution options, but

most do not normally invest effort in the evaluation of different candidate solutions. *Pugin* cites delivery pressure as the main reason why he limits option evaluation—‘it’s not so much an I-don’t-care attitude as a pragmatic thing of, hey, you’ve got to get a system out’. The cost of evaluating different solution options depends on how complete the architecture is at the time. An architect who wants to generate several alternative conceptual models early in the design process need invest only ‘think time’, not coding time. An architect’s propensity to extend ‘think time’ is a form of design discipline. This discipline must not be prescriptive—*Pugin* says he will often ‘try to think about what the options are’ but he does not discipline himself to always generate three options if only two obvious ones exist.

The architects appear to rely upon their past experience and (at times) intuition when choosing between alternatives under time pressure. Each decision is assessed for suitability and risk which may be introduced. The experienced architect can demarcate the situations in which short-cuts can be exploited, and must equally understand associated risks of a limited evaluation. *Piano* claims that he has learned over time which short-cuts he can safely make. He does not, for example, ‘short-cut’ understanding or clarifying requirements.

*Griffin* cites timeframes and finances as the two biggest dampers on generating options—‘there’s no way known that you can build two systems... you say, this one’s the best one, let’s take you’. *Piano* comments that ‘as you become more experienced you eliminate the options more quickly’, because ‘you know that a certain path... will be not appropriate in this case because you’ve met it before... and it didn’t work’. *Piano* depicts his ability to select a viable option from amongst several as one of his marketable skills. ‘You have your experience and that’s what you sell to your clients... if you can eliminate a solution from the start within 20 minutes of considering that solution, then you’re doing your clients a good service’. *Voysey* agrees—‘you’ve got a history, you’ve got knowledge... people are hired for... what you already know, not [for] what your potential is to learn new things’.

T6.30. Architects rely on personal judgement to select a solution option.

*The architects invest think-time but generally little design effort in evaluating options. Experience significantly expedites the generation, evaluation and selection of options, and provides higher confidence in the chosen option.*

## 6.6.4 Complexity vs simplicity

In all forms of design, simplicity is desirable but often difficult to attain. Complexity in software architecture is costly in terms of extra effort to implement and maintain. The more complex elements of a software architecture do not always survive over the system's lifecycle. *van der Robe* equates simplicity with the maintainability of a software artefact—'if your design is simple enough then you are going to be able to change it'. However, complexity in software architecture is frequently necessary because software architectures must solve complex problems.

The ability to balance simplicity and complexity in software architecture is a key skill for software architects, and appropriate handling of complexity is critical for the longevity of a software system. There is little doubt that the architects prefer (and strive for) simplicity in their designs—*McLuhan's* joke metaphor hinges on simplicity by emphasising the minimal structure required to function. The degree of complexity in software architecture is sometimes (but not always) an indicator of the architect's design skill. Several of the architects recounted how and why their earlier work exhibited the absence of these skills. *Mackintosh*, for example, described an architecture that he helped design as 'unbelievably complex, a brilliant piece of work'. But its complexity made it difficult to implement, and eventually the project was shelved. He reflects on this as a case of an inexperienced architect's ego running wild—'it was important to be smart, and do it the best possible way', he admits. Simplicity in an architecture that addresses complexity in the problem space is an indicator of an architect's design ability. *Le Corbusier* agrees that better designs are more minimal designs and that maintainability follows from simplicity:

<p>T6.31. An architect's design skill is partly revealed by how well their design addresses complexity.</p>	<p><i>The architects must design for complexity in the problem domain whilst striving for simplicity. Some architectural complexity ('necessary complexity') is unavoidable. Architects may inadvertently introduce complexity by choosing unnecessarily complex design options ('introduced complexity') or by missing recurring problem or solution space patterns. Skilled architects consistently minimise introduced complexity as a result of their experience and their knowledge of solution archetypes, patterns and practices.</i></p>
---	--

Several of the architects report aiming to reduce rather than expand their architectures as they move through the design process. One sign of the emergence of a maturing software architecture is reduction in complexity and size. *Ruskin* states that he ‘feel[s] pleased whenever I can throw away a few classes, it’s a sense of relief because I knew I was heading in the wrong direction’. *Breuer* describes his reaction upon being presented with a voluminous specification and object model for a portion of a large business application in a financial services project. ‘When I saw the [Insurance] Commissions package specification... I walked out of the meeting... it was one hundred and twenty-four pages... I ended up saying—please turn it into 15’, he remembers. He interpreted the document’s size as a sign that the analysts had not identified the structures inherent in the problem domain.

T6.32. Architectural reduction frequently signifies design progress.	<i>The architects confirm that a maturing object-oriented architecture reduces in scale through the discovery and elimination of duplication as well as the discovery of better-fitting structures.</i>
--	---

#### 6.6.5 Ontology

Although the word ‘ontology’ was uttered by only a few of the architects, all mentioned concepts, conceptualisation and abstractions. A conceptualisation is an abstract, simplified view of the world comprised of objects, concepts, and other entities that are assumed to exist in some area of interest, as well as the relationships that hold amongst them (Genesereth & Nilsson, 1987). Every knowledge-based system relies on some conceptualisation, explicitly or implicitly. Ontology is a description of the concepts and relationships that can exist for a system. The architects implicitly create and define ontology in their models. The concepts in their ontology include domains, data, classes, objects and relationships. What is important with respect to ontology is what it is used for and how the architects use it when designing. *Morris* describes his understanding of ontology in software design:

If we talk about architecture it should give me some words that I can use to describe things to you at a high level, that give us a further basis for talking about the system... architecture is just some set of terms that encapsulate broad sub systems, and allow us to communicate about how they work with one another, without talking about the



details of a particular class. —*Morris*

A solid foundation of solution concepts is critical for software architecture. *Sullivan* describes strong concepts as being like ‘an architectural principle’ or ‘how we do things around here’. To illustrate, he describes a top level object (a broker interface in one of his architectures called `App_Central`) which he describes as a kind of trading post—‘if you want to get access to something then you can go and ask [`App_Central`] for it’. `App_Central` exemplifies how architects and teams invent a shared ontology which then gets anchored in software artefacts such as components and classes.

Building the ontology involves defining the objects and their relationships. The architect’s expertise in abstraction and level-of-detail setting comes into play in object and relationship discovery. Architecting a new object-oriented application can be thought of as language-authoring. ‘Object-oriented coding is about writing a new language every time’, claims *van der Rohe*. He advises that a good (language) design will follow from a well-chosen and managed namespace, and is an indicator of a healthy team design capability. While the architect can initiate the language creation, the team must foster language evolution. Teams ingratiate their own cultures with their own internal languages, and a team’s design dialect evolves within its culture. *Sullivan*, in discussing collective ownership of a solution architecture, makes the point that like the English language, the architecture should always be evolving.

<p>T6.33. The process of software design naturally expresses a language of the solution.</p>	<p><i>Objects directly define the solution’s ontology. The ontology may include classes, relationships and patterns of collaborating objects—all of those software-structural elements that can be named. Architects build narratives around their object ontology that enunciates or confirms their vision (T6.4). Like elements of any language, objects and their meaning in a solution ontology must be expected to evolve. Parts of a solution’s ontology form elements of the meta-methodology that architects take forward from project to project (T6.24). The emergence and health of a design language is an indicator of the team’s collective understanding.</i></p>
--	--

### 6.6.6 Conceptual view

Four distinct sub-categories emerged from the architect's statements about how they use views—conceptual, static, dynamic and historical. Views are a staple component of software design methods and software engineering process. UML (Kruchten 2000) offers different model types for representing (amongst others) static, dynamic and deployment views of a software system. That software architects routinely choose to work with both static and dynamic views and models during design is unremarkable. More interesting is which views they prefer in the architectural design phase, to what degree they use both static and dynamic views, and how and when they move between the two.

Gero makes a distinction between conceptual and routine design (Gero 1996). Conceptual design is the design of a solution using concepts—an abstract machine of sorts. Routine design addresses concerns about the implementation of the elemental components of the conceptual machine. When this distinction was discussed with the architects, most did not consider the distinction valuable, and some did not even perceive the difference. *Utzon* and *Moore* illustrate the extremes. *Utzon* values conceptual design, but he does not see it as being separate from any other kind of design. ‘The architect needs to move through, flexibly move up and down... I don’t see the boundary, I see very few boundaries’, he summarises. At the opposite extreme, *Moore* recognises the distinction between conceptual and routine designing—‘I always need to have a reasonable conceptual understanding of what it is I’m doing’, he says. His motivation for doing a distinct conceptual design is management of complexity and scope. ‘I never like working at a low level of detail where I don’t have a broad understanding of what this thing is, and what it’s trying to achieve at a more abstract level’, he says. When developers do not appreciate the value of a good conceptual understanding of the solution ‘it can lead to some strange pieces of code that just don’t seem to make sense when they’re viewed from a conceptual point of view’, he claims. ‘They solve a specific low level problem but they do not respect the conceptual model, and may compromise the system’s conceptual structure and integrity.

### 6.6.7 Static view

The architects who indicated a preference for a static view of architecture generally rely on functional decomposition to organise scope and complexity. *Howard* always develops his static model first, but quickly finds himself validating each of its candidate classes, which involves thinking about relationships and dynamic behaviours. *Breuer* claims to keep a static model of the entire solution in his head. He limits the complexity of the model by

employing functional decomposition to hide the detail of each element until needed. He can explode or decompose each element in this model when appropriate, but prefers to think of the architecture in its abstract entirety, just as *Howard* does with his context model:

It is not of the detail of the whole thing—but essentially, it is a small collection of about 5 or 6 different key areas... but once you open up one of those areas, it then subdivides into its component parts... I think I am dealing with a kind of hierarchical tree, that I can pull bits out of and then track down a path. —*Breuer*

*Breuer* talks about associating certain key words, names or terms with elements of the top-level structure, so that when these names are used ‘it has a special key meaning to me—it means a huge thing that would take literally 3 days to explain’. *van der Rohe* also uses associative ‘keys’ to preserve and unlock complexity in architecture, but being an active implementer, he reifies his ‘keys’ as class names. He has a practice of always prefixing a candidate class name in a software architecture with the word ‘simple’ to indicate that the class is a first cut. By disciplining the namespace in this way, he encourages simplicity in the top-level class model:

My way... is to find really good atoms, building blocks to build out of... you [then] make complexity by plugging different implementations... but [if] you plug simple implementations it still works. —*van der Rohe*

*van der Rohe*’s simplicity heuristic works in two ways. Firstly, it ensures that a generalised, simple, disciplined set of classes emerge to anchor the architecture. Secondly, it ensures that complexity is realised in the implementations of the key abstractions, rather than in the abstraction’s definitions (interfaces) or the relationships between them. Complexity in the solution structure is tamed and corralled by ensuring a simple top-level object structure. Necessary complexity is introduced only in the implementation of these simple objects.

*Gropius* (who has twenty year’s experience in designing financial trading systems) designs software architectures out of components that rely upon middleware infrastructure. His is a hybrid approach that relies on components (business objects) and message abstractions. He relies on dynamic models because he organises components around message exchanges. Curiously, *Morris*, the architect most committed to the concept of emergent architecture, talks of how he prefers to ‘build a static model in my head, but I don’t build the dynamics’. He reflects on why this might be so:

One of the first things I got taught in a rigorous way was database design, and how to

do normalisation... and I used to build entity relationship models in my head, and then when I transitioned to objects... really, I still build entity models in my head, and then I tack on the dynamics later. —*Morris*

*Morris* thinks of designing architecture as building a structure rather than a machine. His account of how he builds entity models ‘in his head’ counters his earlier argument for ‘emergent design’ where he claimed that all worthwhile software design occurs as a consequence of emergent structure during the code-polishing process. His statement reveals how much conceptualisation he is able to comprehend ‘in his head’ before committing the resultant structural design to code, and is perhaps a telling insight into the abilities of the emergent designers to design *a priori* without relying on externalised design phases, activities or representations.

#### 6.6.8 Dynamic view

A different subset of the architects expressed a preference for a dynamic view of software architecture. They talk of architecture as representing a kind of conceptual machine that they imagine in their mind’s eye, and of ‘executing’ the architecture’s mechanism in their minds. Dynamic visualisations draw on an analogy between objects exchanging messages and the parts of a mechanical artefact. Many people can imagine a running car engine, even those with little mechanical aptitude, but only a minority of the architects describe using analogous visualisations of the dynamic dimensions of their designs.

Those who use a dynamic view talk of knowing how the system will work before actually committing anything to whiteboard, paper or code. For example, in a design using a publish-subscribe mechanism, the architect might imagine objects subscribing and being notified of a publication. Some of the architects can keep bigger, more complex machines—including the mechanisms of those machines in some detail—in their minds. The dynamic view is useful in designing systems or sub-systems that must handle complex temporal sequences. For example, *McLuban* described ‘thinking about the objects that are in play’ in a scenario where he needed to construct and then gracefully tear down a complex network of interdependent objects in a particular sequence. *Stickleley* also works with a dynamic view. ‘For me, the machine’s always there’, he says. ‘That’s how I try to keep a clean design of the system’. He uses the evolving dynamic model to formulate responses and clarifications in his own mind, in terms of the machine. His conceptual machine is an ephemeral tool—it cannot be easily shared, and as *Stickleley* admits, ‘the definition in your head isn’t really complete... [so] it tends to be fairly fluid’. Once the architect is happy that his conceptual machine has been sufficiently explored in this

personal cognitive space, it must be serialised onto a sharable medium and communicated.

#### 6.6.9 Historical view

Finally, *Cook* talked about the importance of leaving a design rationale trail for those who follow. ‘There’s a whole line of reasoning’ about why a particular design decision was made, he says, and ‘understanding how you arrived at certain things is very important’ on a software architecture team. He describes his experience of joining a project late. Although the project’s source might be of relatively high quality, the code does not define these design justifications, without which the new architect is disoriented. A documented narrative is needed to capture the actual narrative that was lived out by the designers at the time the architecture was designed.

T6.34. The architects use conceptual, static, dynamic and historical views interchangeably.	<p><i>Most architects report using conceptual, static, dynamic and historical views interchangeably and at times concurrently.</i></p> <p><i>Some express a clear preference for one view over another, particularly when doing conceptual design, with static (class) and dynamic (object collaboration) views being dominant.</i></p> <p><i>Their stated preference appears to be independent of problem type or domain.</i></p>
---	--

#### 6.6.10 Bias, perspective and perspective-shifting

Bias is implicit in all designers and in all acts of design. The architects reported different kinds of bias—technology bias (which results in selecting known technologies over unknown ones), perspective and paradigm bias (which motivates the architect to adopt a preferred perspective or paradigm) and personality bias (which follows inescapably from the individual’s personality).

Personality bias is described by *Stickle*. He confirms that ‘architects often have their own prejudices’, their ‘particular style or way of doing things, or tools that they like to use’. He is starkly honest in admitting that ‘the last two projects I have worked on have had oddly similar architectures... and they were quite different systems’. Personal prejudice is a risk mitigation mechanism, because architects return to ‘what you’ve learned or what you’ve been using most recently’, they ‘tend to stick to the things that they know’ because of their

investment in prior solutions.

Architects also bring a particular perspective to a design that has the potential to bias their designing. Not many of the architects explicitly recognised this. Perspectives appear to be strongly influenced by historical factors such as when the architect was educated or entered the profession. The strongest perspective bias observed in the analysis of the architect's discussions is paradigm bias. Paradigm biases detected amongst the participants include data, behaviour and functional biases. *Howard* and *Morris* acknowledged their data-centric perspectives, and *Le Corbusier* his behaviourist perspective. *Howard* has a thirty year history of working with database design and recognises that he tends to perceive things from a data perspective. He also recognises that his long professional history attracts data-oriented work.

*Mackintosh's* story about the design of a telecommunications exchange monitoring system illustrates a misconception in the design of the software architecture that resulted from both perspective and paradigm biases. The original object-oriented architecture was designed with a data bias (paradigm bias) which resulted in a model centred on an EXCHANGE class. The key modelling assumption appeared sound but proved to be flawed when the system's real requirements emerged after the EXCHANGE-based model had been committed to a substantial code base. What was needed was a concept of a measurement class. The perspective that biased the architecture was that of anchoring its business object model on a tangible object (the telecommunications exchange) rather than an intangible unit of measurement. This fault in the object model was symptomatic of larger project problems. The full story of this system and its architects is told in the next chapter.

T6.35. Experienced architects are aware of their own perspectives and how they influence their designs.	<i>Experienced architects are conscious of their perspective bias. They may adopt the perspectives of different stakeholders, users, or different key classes in a model, with the objective of exposing design options and fully exploring requirements.</i>
---	---

While a number of the architects acknowledged paradigm and perspective bias, none described explicit paradigm-shifting during the conceptual or exploratory modelling activity. There is no reason why the experienced software architect cannot explicitly adopt

alternative paradigms, sketching abstract models and internally debating the pros and cons of the models that each alternative paradigm or perspective gives rise to, leading to an informed choice of modelling perspective. Some of the architects appear to be able to recognise limitations of their preferred perspective, even if they do not report actively exploiting this awareness.

Paradigm perspectives are not limited to behaviour versus data. *Breuer* repeatedly returns to a hierarchical view of the world in his stories of designing. He takes it ‘almost [as] an article of faith’ that everything can be viewed as a tree. *Breuer’s* conception of hierarchy is based on functional decomposition. In the top ‘layers’ he attributes architectural blocks on the basis of function. These embody key solution abstractions and behaviour, but rely upon services provided at layer boundaries, thereby forming an invocation or delegation hierarchy. Each layer embodies its own services and internal model, and complexity is dealt with through appropriate assignment of behaviours and state to the layers. ‘The crucial part about the architecture is to do that layering process first’, he claims. *Breuer’s* commitment to the functional decomposition paradigm and the resultant hierarchies it produces is a clear case of paradigm-bias.

T6.36. Architects are subject to paradigm bias.	<i>Architects reveal a propensity to be locked into a single design or decomposition paradigm. Architects should be capable of disbanding their paradigmatic perspective and adopting another, even if only as a mechanism for validating their preferred paradigm.</i>
---	---

#### 6.6.11 Abstraction

The architects regard abstraction as one of the most basic and important skills in software design. ‘If you can’t abstract—forget it’, says *Pugin*. ‘If you’re not trying to abstract then you’re not going to succeed at object-oriented’, claims *McLuhman*. ‘I don’t think you can develop any non-trivial system without good abstraction skills, good conceptual models, and understanding the difference between the abstract and the concrete’, claims *Moore*. *Piano* agrees about abstraction’s pre-eminent place in the architect’s skill set. ‘If I was to pick one developer out of an organisation of 120 that had abstraction skills and nothing else, he would be my architect’. The architect must have the ability to avoid getting

overwhelmed—in the midst of the sea of detail, the successful architect can ‘see the big picture’ and ensure that ‘it all fits together’.

Abstraction is fundamental to any discussion of software architecture, and all forms of architecture rely on abstracted views in which the designer selects particular features to represent over others for particular reasons. Abstraction in software is pervasive—it is found everywhere from intrinsic types in programming languages to architectural patterns and archetypes. *Morris*’ definition of software architecture relies on abstraction—‘it [software architecture] is constructing software systems that have layers of some sort, where interactions are between a layer and its neighbours rather than between a layer and anything it can find, so that there is some sense of localization of change, not just at the class level, but at some further metaphorical level’. Abstraction is related to metaphor in that memorable abstractions (such as Layers) have commonplace or ‘real world’ analogues. Abstractions improve comprehension, encapsulate complexity, and therefore preserve structure through a system’s lifecycle.

A key issue when using abstraction in any form of design is the selection of the level of detail. As we have seen (T6.18) the architect changes his design based on his perception of the capability of the architecture’s consumers. One way the architect does this is by choosing an appropriate abstraction level. *Mackintosh* selects his detail according to the high-level problem he wants his architecture to solve, and how much detail he thinks the implementers will need—‘you can solve a problem without a lot of definition... and it gives everybody a footing to take off’. *Piano*’s architectures are comprised of ‘implementable but undefined blocks’ such that the architecture is ‘not ultimately one hundred percent prescriptive’. Drawing on the built world to illustrate his point, architecture is ‘the broad brushstrokes of what the entire building will do, its capacities, its functions’ and his architectures provide ‘the framework of the building, the layout of the rooms in terms of measurements, in terms of flooring, in terms of bearers and supports’.

T6.37. Architects select abstraction levels for the architecture’s purpose and its consumers.

*One of the ways that architects consider the consumers of their designs is by selecting an appropriate level of abstraction for representing the architecture. The chosen level is not applied universally—rather, some areas of the architecture will be detailed while others are left abstract, unfinished, or high-level.*



For *Ruskin*, forming the right abstractions primarily depends upon knowing what detail to leave out. He describes deliberately deferring some detail while promoting or dealing with mapping other detail. ‘It’s all about modelling... and modelling is all about what you are going to leave out’, he says. *Breuer* agrees—‘we are going to leave some things out... we are going to have to include other things’. He relies on the business requirements to tell him what to leave out and what to include.

Architects do not report working at one level of abstraction then moving up to a higher level, or dropping down to a lower level. Rather, they move through different levels of abstraction constantly. *Utzon* describes ‘the ability to work your way up and down the levels of abstraction’ as being ‘extremely important’. Even the idea of there being abstraction boundaries appears foreign to him—‘I don’t see the boundary, I see very few boundaries’. *Pugin* describes abstracting ‘all the time’ when designing software.

T6.38. Architects move flexibly between levels of abstraction when designing.	<i>Architects move between levels of abstraction constantly. Some architects exhibit a preference for a particular level of abstraction to which they naturally gravitate when thinking about software architecture. Some architects prefer to spend most of their time at the highest level of abstraction, while others prefer the lowest levels (ie. the code). They choose different strategies for excursions from their preferred place to other levels in the architecture’s abstraction hierarchy. The architect’s preferred level broadly follows the two categories that emerged from the analysis of approaches to conceptualisation—top-down and emergent.</i>
---	--

*Cook* describes abstraction skills as the key enabler for fitting known structures to a problem, and he admits to deliberately re-conceiving or even ignoring certain mis-fitting parts of a problem in order to fit an architectural structure he is familiar with to a presenting problem. ‘When you can abstract something and generalise it’, he suggests, ‘you can make things look the same, and when you can make things look the same, they become easier to understand’:

It’s much easier to make something look like something you understand if even that requires discounting certain aspects of it... you may say OK, look, I’m getting caught up

in this bit here... if I just ignore that as being a special case, I just arbitrarily or artificially ignore that, then, the rest of it looks exactly like this thing over there. —Cook

T6.39. Architects use abstraction when fitting known solution structures to presenting problems.	<i>At times, the architects use their abstraction skills to make presenting problems look like problems for which they have known architectural patterns or system archetypes. Fitting known solution structures to a presenting problem is related to requirements negotiation (T6.3).</i>
--	---

Validating abstractions is important for *Howard*—‘the single biggest thing in success of a model is to take it to the detail’, he says. He describes verifying his models by constructing prototype databases and populating them with sample data from a scenario. ‘The designer of the model [must be] able to grapple with hugely generic things like ‘asset’ and ‘location’ and ‘arrangement’ and be ‘able to go right down and populate attributes’ as well. Well-chosen abstractions are self-explanatory. *Le Corbusier* agrees that simplicity is an indicator that the abstraction is well-formed and that the abstraction process has terminated:

I just find the easiest way to do it is to work your way down to behaviour and association... if you can’t describe the component or an object as a behaviour—this does this for my system—if you require a paragraph to describe something, you have made a mistake. —*Le Corbusier*

The experienced architect’s abstraction skill is not limited to creating and finessing architecture models and classes. *Pugin*, *Cook*, *Le Corbusier* and *Gropius* all concurred that the skilful architect abstracts elements of their personal process as well, in order to reuse approaches, techniques, and ways of acting that have proved useful and fruitful in the past. This illustrates the architect’s abstraction skill as one of seeing structures, sequences and shapes through the detail to selectively use or reuse the essence of a structure, pattern or process.

T6.40. Architects abstract equally in software and non-software dimensions.	<i>Experienced software architects employ the same basic abstraction skill in the business domain, software solution domain, and process dimensions.</i>
---	--

The architects use abstraction to exploit similarities in the interests of achieving simplicity. The ability to abstract may be the architect's most important personal skill. 'When you can abstract something and generalise it you can make things look the same, and when you can make things look the same, they become easier to understand', claims *Cook*.

T6.41. Abstraction exploits similarities to achieve simplicity.	<i>Abstraction discovers the underlying structures in the problem and solution spaces. When these structures converge, the architecture is simplified. Reducing, collapsing, converging, simplifying structure are all signs that good abstractions are being found and that the design process is making progress.</i>
---	---

#### 6.6.12 Abstraction discovery

The architects were asked about how they discover abstractions. Although most described their personal approach to abstraction discovery as incorporating both bottom-up and top-down approaches, some stated an explicit preference. *Johnson* typifies those who prefer bottom-up abstraction discovery when he describes discovering objects in an 'as-needed' fashion. He starts with one or more classes and uses the process of elaboration to drive discovery of new classes. New classes are 'borne out of necessity' as opposed to some sort of 'arbitrary, high-level ponderings', he says. By contrast, the architects who prefer to work top-down are able to deal with complex domains and fuzzy boundaries. They must also be comfortable deferring detail. Several described personal techniques for managing scope and complexity. For example, *Howard* has developed a pragmatic way of scoping a domain model:

I don't feel happy that I've got my mind around the problem until I can come up with a model that represents the super types, the classes on a single sheet of preferably A4... until I can see the entire the business at a high level, I think I haven't seen the patterns. —*Howard*

*Howard* claims that he is yet to come across a business, no matter how complex, where 'with a bit of effort' he could not reduce the complexity to a single-A4 class model. Not all the architects fall into the top-down or bottom-up categories. *Gropius* prefers to design at a component level of abstraction because his experience is mostly in designing middleware-

based architectures. He starts architectural design with the set of business objects and then instruments their interactions and those with the application's infrastructure.

Associated with bottom-up abstraction discovery is the notion of emergence. *Morris* actively discourages people from doing design work up front. 'People's first step is to say, well we need architecture, and I say well no you don't... it will appear':

So I guess I have a inverse approach to architecture—architects (in most people's minds) worry about the big picture and let the fine grain picture take care of itself... my approach is to take care of the fine grain details and let the big picture take care of itself. —*Morris*

He encapsulates his bottom-up, emergent design approach in a bunch of heuristics. Firstly, he re-factors out duplication—'whenever there is [sic] two pieces of code that look the same... the software is telling you that a concept is missing'. He migrates the duplicate behaviour upwards using an abstract super-class, or he introduces aggregation by introducing a third class that provides a service. Removal of duplication results in reusable classes. This process of repeatedly and rigorously driving out redundant code results in smaller objects that are useable in more contexts. 'We never aim for reuse first, we aim for use, and it just turns out that those ones also happen to be reusable a lot of the time as well', he declares. The architectural principles of good cohesion, lightly coupled objects and the removal of duplication, with the safety net of comprehensive and accurate unit tests, account for *Morris*' personal process for emergent architecture. 'And we do all of that stuff rigorously, and the big picture takes care of itself', he concludes.

T6.42. Architects follow notionally divergent paths when abstracting.

*Architects do not universally practice traditional step-wise refinement or decomposition to discover abstractions. Architects may arrive at an initial design by progressing from the abstract to the specific, or from the specific to the abstract, or they may follow a combination of these two trajectories. Some architects report working both trajectories more or less simultaneously, both on different and the same parts of the solution space.*

T6.30 stated that an architect's experience expedites option selection. The same is true for the formation of viable, enduring abstractions. *Moore* agrees that 'the more experienced you are as an architect and designer the quicker you're able to hone in on the critical bits'.

This supports the emphasis that *Piano* places on the affect of time pressure on architects. *Kahn* relies heavily on his experience with particular candidate designs during the process of selecting or creating a new design. ‘I think probably the biggest thing that I rely on is what I’ve seen before’. His approach can suffer from a form of architectural myopia—‘whatever I build is highly likely to be an evolution of something I’ve built in the past’. *Kahn* is sufficiently self-aware to ‘discipline myself to go outside of that in some cases’.

#### 6.6.13 Generators

In design theory literature, generators are ideas, concepts or metaphors that serve to initiate, inform and shape design (Beyer and Holtzblatt 1994; Lawson 1997; Lovgren 1994). Stories from the accounts of built-world architecture typify generators as metaphors that remain recognisable in the final form of the building—the sails of the First Fleet ships on Port Jackson as metaphorical inspiration for the Sydney Opera House, by way of famous example. In software, the idea translates to the notion of an idea which seeds a universal structural or detailed recurring pattern in the architecture. In the software fabric, the meaning of, and usefulness of a generator is debatable—is Model-View-Controller a generator? Or in a business context, is the class model that describes a Customer, a Supplier and an Order (and how these classes relate) a generator? What value can be had from promoting the generator concept in software design? The concept of generators was raised with the architects to investigate these questions, and to determine how committed the architects are to the design principle of a handful of structural patterns or design drivers that may be applied across multiple layers of architecture, or are in some other way unifying.

The architects were generally unfamiliar with generators but in most cases their thoughts turned to principles. One interpretation is to consider generators as the things in the project’s conceptual space that impose principles and order when the project is in what *Lethaby* refers to as its ‘chaotic ramp-up phase’. Designs start out as a rough set of ideas and principles, he explains. Explicitly stating these ‘ideas and principles’ during this phase is important. ‘You spend time in front of the whiteboard with a group of people going through the ideas, thrashing them out, revising them, and fixing the leaks’. Some of the architects talked about discovering or formulating ‘architectural principles’—general assertions about a problem or solution mechanism. Where these translate directly to recurring architectural structures, they could be said to constitute a kind of generator. *Breuer* equates architectural principles (which he calls ‘first principles’) with the conceptual

‘footholds’ that he struggles to attain when commencing design:

The model in my head... comes out in speech as a sequence... it will come out in terms of definitions... I am at first principles—it goes back to my mathematics training—it is even unlocked by the key phrase ‘first principle’. —*Breuer*

*Breuer’s* first principles have certain characteristics by which they may be recognised. He uses words and phrases like ‘simple’, ‘elegant’, ‘pretty’, ‘easily memorisable’, ‘a guiding rule’. *Breuer* thinks of a first principle as a ‘key’ that unlocks the power of a concept:

My example is the discounted cash flow equation—and the first principle of amortisation is the sum of discounted cash flows is always zero.... once I discovered that principle, I knew that I understood everything there was to know about amortisation, I didn’t need to go any further... that was the first principle... understand that and you can derive everything else about cash flow. —*Breuer*

This particular principle unlocks a problem domain for *Breuer*, and may or may not turn out to be a generator in the designed solution. The distinction between generators and perspective is important. Generators are foundational structuring principles, whereas perspective defines the viewer’s chosen observational point and paradigm. Perspective influences what generators the architect may see.

<p>T6.43. Architects rely on key concepts to guide the design.</p>	<p><i>In some cases, the architects seek to seed their designs with primitive structures. They describe these as ‘architectural principles’ and they may source them from the problem domain or from a solution archetype (an amalgam of architectural patterns). They provide a recurring structural pattern that helps the architect to establish the architecture during its conceptualisation. Generators are principles that lead the architect to select patterns or archetypes. Examples of generators include a common object lifecycle, the consistent use of views or filters on an object or collection, or the use of indirection between objects and their clients such as might be provided by certain design patterns.</i></p>
--	---

## 6.6.14 Crystallisation

We saw in T6.10 that the architects deal with different kinds of commitments at different phases of the design process. The architects report that documents are a particularly inflexible medium for committing design decisions. ‘Crystallisation’ is *van der Rohe*’s term for a phenomenon of stagnation in software design due to heavy-handed documentation and process overheads. *van der Rohe* has witnessed bureaucratic development processes weigh down the design process when organisations invest in specifications that make revisiting an earlier decision or redirecting a mis-directed project difficult or impossible. ‘You can’t aggressively re-factor a specification that’s sixty pages long and cost you half a millions dollars to write—and there’s your problem’. *van der Rohe*’s solution is twofold—he (radically) suggests not writing specifications, and less controversially, keeping the high-level structure of the system clearly represented in abstract classes.

In *van der Rohe*’s view, the existence of a voluminous specification is conventionally interpreted as progress. ‘Most people aren’t prepared to change their minds’, he suggests, ‘because they see it as too much investment’. *van der Rohe* concludes that it is human nature to correlate sunk cost with value. An architect’s decision to terminate a flawed candidate design is often made more difficult if the effort invested in that option is both non-trivial and highly visible to his sponsors. Sunk cost therefore increases the likelihood of crystallisation.

Crystallisation can be avoided if the architect correctly makes and manages commitments in the architecting process, and documents those commitments in appropriate ways. Finding a balance between making progress and avoiding crystallisation can be achieved by knowing which decisions can (and should) be made now, and which decisions can be deferred (T6.10). Architects must direct development teams—‘you have got to make some commitments... you have got to give enough to the development team to be able to start developing’, *Stickleley* says. They need boundaries (such as the choice of tools and techniques) in order to initiate team progress, but the architects must avoid committing too many design decisions too early.

T6.44. Sunk cost increases architectural ‘crystallisation’ or rigidity.	<i>The architects are aware that it is common for sponsors and stakeholders to associate investment in effort (time and therefore cost) with progress, that volume of documentation is additionally roughly associated with progress, but that quantity of design documentation does not imply completeness or quality of design. They report that projects sometimes reach a point of investment in artefacts that makes rework or re-architecting difficult or impossible, even though flaws in the architecture are evident and acknowledged.</i>
---	--

#### 6.6.15 Archetypes

Experienced architects evolve a small number of preferred solution archetypes over multiple projects. A solution archetype is comprised of the essential structure, concepts and principles that are commonly applicable to a class of problems. An archetype can be thought of as a frame or schema (a structural template for organising solution elements and design know-how). Archetypes are loose bundles of related ‘soft’ knowledge, and may or may not include design or code patterns or fragments—as such, they may be thought of as the knowledge required to design a software artefact (such as an object-oriented framework). They are frequently domain-specific, or contain some elements that are unique to a domain and recur with regularity. The emergence of archetypes in the qualitative analysis is significant because they provide a handle on a kind of knowledge structure employed by the architects.

All experienced architects bring their catalogue of archetypes with them, whether they be implicit or explicit. ‘There’s no such thing as green-fields’, *Mackintosh* states, ‘you’re bringing along your own legacy system, which just happens to be in your head’. Architects draw on this ‘legacy system’ as a way of artificially constraining the field of possible solutions, and to give them some initial structure when approaching or commencing a new software architecture engagement. Experience leads architects to develop a broader base of archetypes, and to develop confidence in selecting the appropriate one (or part thereof) in a given design situation.

*Lethaby* describes his ‘pluggable architecture’ as something which emerged over time from several projects. ‘The architecture ideas... evolve from one job to the next’, he states, and



as a consequence, ‘you are continually revising your body of ideas’. The concept of pluggable components in *Lethaby*’s archetype is neither new nor original. ‘I guess the word ‘pluggable’ definitely came from Smalltalk... I guess they took it from, you know, electronics, or they took it from manufacturing’, he contemplates. Although the idea of ‘pluggable’ is no breakthrough, what is significant is *Lethaby*’s preservation, evolution and use of the concept over a series of projects and how he uses his ‘pluggable’ archetype in conceptualising a new architecture.

Invention and archetype-fitting occur simultaneously. Some architects describe their personal process of arriving at a design as one of inventing abstractions and relationships. Others (and in a few cases the same architects) describe their personal process of arriving at a candidate design as one of fitting known solution archetypes to the problem at hand. These two approaches are not mutually exclusive but are instead performed together to some degree. Only some of the architects have externalised their ‘legacy systems’ into archetypes that they could cogently discuss, reason about, and relate repeated experiences of. Architects may have solution archetypes that they can freely discuss and describe for which no published pattern exists. Their descriptions of their archetypes may appear unstructured and even simplistic. *Stickley* describes the architecture of his employer’s product in the following terms:

What is the system doing? In terms of activation and the unified message system... it is almost just a data flow thing—you’ve got data coming in one end, it gets all sorts of transformations and things done to it, it gets stored and delayed and then it pops out the other end... and in the case of activation it can actually go back through again... so it goes from the management system, down through to the device, and it comes back up... so you get an architecture that (sort of) goes down, in a ‘U’, to the bottom, down to the device at the bottom, then back up to the thing at the top... so that actually defines the architecture. —*Stickley*

This cameo reveals how *Stickley* is comfortable to discuss what is a large and complex software architecture in remarkably simple terms. His ‘U’ archetype is probably the result of a large amount of selective abstracting on his part. Also worthy of note is the fact that his depiction, while apparently specific to the system at hand, is an amalgam of known patterns, including Layers (he refers to messages going ‘down... and... back up’) and various message encapsulation and storage patterns. In putting his architecture into a simple narrative *Stickley* has constructed a system archetype for the interview situation. His earlier comments on architect’s personal ‘prejudice’ would suggest that this particular archetype is an amalgam of a number of architectures he has designed or worked with.

The architect's archetypes appear to be derived from classes of systems they have designed or had exposure to. These constitute patterns at the highest level of abstraction. They assume a paradigm and imply a particular structure. For example, *Mackintosh* talks about how he toyed with a common archetype for a network management system that he designed for an Australian telecommunications company. 'It's an Order Entry system', he argued at the time. What complicated the situation was the fact that the system was intended to automate a telecommunications network problem resolution workflow. There were no orders, no stock, no inventory, and no fulfilment processes. But *Mackintosh* could see the essential structural similarities—a 'network event' is essentially an 'order', and the network transmission model is essentially an inventory, he believed:

As much as there was a lot of data to capture, it was just names and addresses... the fact that the addresses were [multiplexed transmission] links [and not delivery addresses] really didn't matter. —*Mackintosh*

<p>T6.45. Architects abstract simple archetype representations from complex software architectures.</p>	<p><i>Some of the architects were able to narrate large and complex system architectures in remarkably simple terms in a story-telling fashion. Their depictions might involve basic message traces and paths, holonic self-similarities, metaphors, or a combination of these elements. Their narratives are abstract and span or imply many patterns or combinations of patterns. Archetypes are similar to knowledge schemas or frames, and can be thought of as the 'know-how' required to design object-oriented frameworks. They generally do not include code artefacts but the architects may retrieve the source code or related design detail of architecturally significant components from previous systems when selecting and fitting an archetype. Archetypes are an efficient and minimal form of knowledge template—the architects retain only the key characteristics they need to instantiate or fit the archetype into the new situation.</i></p>
---	--

Architects who work repeatedly in particular business or technology domains develop domain-specific interpretations of a generalised pattern and treat these as archetypes. For example, *Johnson* describes a variant of Model-View-Controller that he has repeatedly used

in game software. The archetype comprises a queue of pending orders (commands) that have been issued to all of the active game objects. The orders dictate how game objects move to coordinates, interact, and so on. The controller implements various coalescing algorithms over the queues of commands to implement behaviours that depend upon game state (such as the stage of the game or the player's history). The archetype provides a conceptual, structural and code-level reference for *Johnson's* game architectures. Outside game domains, its usefulness without extensive modification is dubious. As noted in T6.45, archetypes like this primarily provide value as a vehicle for an architect to package solution design knowledge.

Architects choose to use archetypes for the same reasons that they choose to re-use familiar technologies—risk reduction. The most pressing risk to be reduced for the newly engaged architect is that of not producing a viable architectural solution. Using known archetypes helps the architect to package know-how and leads to confidence in the proposed solution. Another reason to use archetypes is to facilitate a stable and reliable division and transition of work. System archetypes, such as *Mackintosh's* Order-Entry archetype, are comprised of well-known modules and components. Architects accumulate experience in handing off, supervising and delivering some of these components. Archetypes bring not only architectural structure but also a micro-process for partitioning work and organising delivery responsibilities and tasks.

The existence of solution archetypes is well grounded in the architect's accounts of design, but the processes that they use to select and fit archetypes to problems are less clear. *Cook's* reflective description of how he fits solution archetypes to problems reveals a significant degree of interpretation of the problem:

OK, it's different on the surface—well that's the first alarm bell... I haven't seen this before... therefore, it can't be right... I guess that tells me that that, for a start, I'm looking at the problem in the wrong way... I must be able to make this look like something that I already understand... and once I can do that... maybe it requires breaking the problem down into smaller chunks, such that those chunks conform to something that I already understand, and then it's just a matter of putting those chunks together again. —Cook

*Cook's* archetype-fitting process is 'continually trying to look at a problem and re-evaluate it in some well-understood context'. As we have seen (T6.3), many problems, or parts thereof, are negotiable. Where no fit is obvious, architects may go beyond perspective (T6.35) and deliberately attempt to alter their stakeholder's perception of a problem in order to achieve an acceptable fit with an existing archetype. *Cook* describes 'discounting

certain aspects’ of a problem in order to achieve a fit with a known solution archetype:

If I just arbitrarily or artificially ignore that, then, the rest of it looks exactly like this thing over there... so what is it about this special case that makes it suddenly look the same? —Cook

T6.46. Architects interpret problems in search of a fitting archetype.	<i>Just as architects interpret problems to fit known solutions (T6.28) and to fit known abstractions (T6.39), they engage in a similar form of interpretation at the solution level. The archetypes (system-level structures) they manipulate are an amalgam of structure, metaphor, conceptual machine and the associated heuristic knowledge gained from past experience of implementation.</i>
--	--

#### 6.6.16 Personal patterns

The architects are most comfortable with the concept and use of patterns. They talked about using both commonly recognised architectural and design patterns (such as Layers, Model-View-Controller, Proxy, Command, or Composite) and their own informal ‘patterns’—concepts, design fragments, classes, and schemas drawn from their experiences of design in projects. The architects used the term ‘pattern’ informally. For example, Moore distinguished between patterns and ‘architectural fragments’:

I’ll always think in terms of, not necessarily patterns, I don’t want to say patterns, but things that I’ve... design fragments or architectural fragments that I’ve seen work before, which look like they might fit the current problem. —Moore

Cook similarly talks about patterns as being ‘any approach that I have reused and refined over a period... whether that be architecture, design, the way I go about installing software’. Cook’s patterns include personal or group process conventions. ‘When something conforms to a certain picture or pattern, it’s a pattern, and then you come up with a tried-and-tested solution... a way of solving it’. Given sufficient time and breadth of experience, the architects talked of discovering the essential software-architectural patterns themselves, just as a by-product of good design and modelling practice. ‘I think if you have been doing it from the outset, you would have found them yourself already’, Le Corbusier notes. Software architects are prepared to discover or mine these ‘personal

patterns’ from sample code. The resultant ‘patterns’ may be ephemeral in that they have immediate use but will not necessarily be written down, or remembered in detail, beyond the motivating design episode. *Cook* describes his practice of doing this:

[I will] just have a look through people’s source code to see the way other people have approached it... and I will try and get, you know, 3 or 4 or 5 or 6 of these examples, in the hope that I will see some common thread through these, and I will take that to be a... if not a standard way of doing it, a defacto standard way of doing it. —*Cook*

The fact that these ‘*ad hoc* patterns’ may reproduce the solutions of well-known patterns does not concern *Cook*. He is pragmatically interested in getting a result that is grounded in other people’s experience. This self-motivated code-research is a relatively recent phenomenon enabled by ubiquitous internet and open source products and libraries. *Moore* is equally unconcerned about rediscovering known patterns for himself:

And you may well be using them, but you don’t know their name, and you don’t know their formal structure, and you’re not particularly conscious of the way the classes involved in a particular pattern are described. —*Moore*

T6.47. Architects rely on ‘personal patterns’ when designing.	<i>Most architects can recite a number of ‘Gang of Four’ patterns, but regularly use an unknown number of ‘small-p’ patterns—heuristics, rules of thumb, small code and process structures. These are drawn from the architect’s personal recollections and reconstructions of published patterns, design fragments, idioms, and known solution fragments. The ‘personal patterns’ that experienced architects accumulate over time overlap with and duplicate known architectural and design patterns. The architects appear not to be concerned by this rediscovery.</i>
---	--

Architects expand their personal catalogue of patterns with each new project. Architects do not generalise their patterns during or after each implementation—rather, they remember the experience of using a pattern on a project and generalise or re-contextualise from this memory when and if the same pattern is needed in the future. *Sullivan* illustrates how he regards the maturity and utility of his AppCentral concept:

I’m not absolutely certain about my ‘AppCentral’ approach—so I’m sort of adding to that... I’ve been through a few cycles, so that’s my starting point, I don’t have to create

that idea any more... they're sort of like resources you have at your disposal, the ideas that are already established. —Sullivan

T6.48. Architects remember patterns in an associative fashion.	<i>Architects generally do not remember the detail of their personal patterns—instead, they remember the value they associate with a particular pattern in a situation. They will call up previous code samples or other examples when evaluating or reifying the pattern.</i>
--	--

#### 6.6.17 Intuitive leap

The architects were asked to discuss what they regarded as their most successful software architectures. Many described satisfaction as deriving from having made an unexpected breakthrough or ‘intuitive leap’. Not surprisingly, most of the architects had difficulty in being objective about the intuitive leap. *Stickley*, for instance, is ‘not conscious of it’, and believes that ‘it’s subtly different each time... and it evolves’. Learning to design is like learning ‘the essence of calculus, it’s not possible for you to forget it, it’s not a bit of knowledge you’ve got, it’s a bit of understanding you’ve got, it’s a part of your person’. Like ‘the ability to ride a bike’, the ability to design ‘kind of happens in an instant’:

When you’re attacking a software design problem, I think there’s a point—and we talk about an intuitive leap—there’s a point at which you can actually see the shape of it.  
—*Stickley*

*Stickley* associates the intuitive leap with translations from one medium into another—from use case models and requirements depictions into domain or analysis models, or from analysis models into code, for example. Intuitive leaps also sometimes occur when otherwise disparate parts of the same solution are brought together. The software architect (as information and knowledge hub) is in a unique position within the project team to perform this act of combination. *Breuer* says that his ability to make an intuitive leap seems to depend on ‘people coming to me with questions’. When questioned, he listens and then reflects ‘on an intuitive level’ about what it is the team member is really trying to achieve. This gives him distance from the immediate, pressing problems and allows him to continually reconceptualise the evolving architecture.

## 6.6.18 Breakdowns

In the design literature, intuitive leaps are associated with breakdowns (Schon 1983). A ‘breakdown’ is an unanticipated event that breaks the designer’s path or concentration in some way. Reflective designers are sensitive to breakdowns and use them to inform and alter their personal design actions. The analogy of driving a nail into a block of wood (discussed in 3.2.3) was raised with some of the architects in an attempt to get them to identify the metaphor with their own experience. In the analogy, the breakdown event occurs when the hammer glances off the nail head and dents the wood. The event breaks the craftsman’s concentration and creates an instantaneous opportunity for reflection. In the act of designing software architecture, breakdowns can come in many forms—what is of interest is whether, and how, the architects are conscious of them and then use them.

In *McLuhan’s* experience, most breakdowns have their source in badly formed abstractions around the domain’s ontology. He strives to uncover well-formed abstractions and structures, primarily in the data. He comments that ‘the whole purpose of identifying the underlying data structure’ is to ensure that things (the software structure) fit together properly. A breakdown, he says, is therefore a consequence of an undiscovered structural flaw and a prompt to reconsider the architecture’s data structures in light of the design activity that was being attempted when the breakdown occurred.

*Lethaby* commented that there are weaker kinds of breakdowns which, although not as destructive as denting the wood, are dangerous to the designer’s or team’s productivity. ‘Maybe your arm is getting sore, because you are holding the hammer slightly the wrong way’, *Lethaby* surmises. And the soreness is a prompt for the reflective designer to ask, ‘I am just hammering—why is my arm getting sore?’ *Lethaby’s* observation has similarities with *Morris’* attitude to duplication when doing emergent design—in both stories a sense of tedium, repetition, or dullness hints at an underlying structural problem. It is difficult to put quantitative measures on these kinds of sensory ‘feelings’, they claim. *Morris* is content with ‘listening to the code’ but such notions are considered unacceptable in the software engineering milieu. *Lethaby* does not think that quantifying the phenomenon is necessary or important:

I think generally there are enough programmers who have a good sense of those things... fortunately, in environments that I have worked in at any rate, there have been always plenty of programmers who have had a very good sense of ‘nice code versus nasty code’—and it shows that I am optimistic that this is a general condition, that is a human condition where you naturally respond to some kind of elegance in the code we are writing, and the machines we are making. —*Lethaby*

T6.49. The architects identify both hard and soft exceptions as breakdown events.	<i>The architects associate both hard and soft software design and development exceptions as kinds of breakdown events. A breakdown event effectively breaks a period of commitment to the current design task or option, and in some cases, may constitute a trigger for an intuitive leap. A ‘hard’ exception or failure must be addressed by conscious re-design or refactoring. A ‘soft’ exception is typically less obvious and may go undetected for some time, but when detected, has the same result. Inexperienced or insensitive architects may miss ‘soft’ exceptions.</i>
T6.50. A breakdown event may initiate perspective and/or paradigm-shifting.	<i>A breakdown draws attention to some aspect of an inadequate or flawed design. This forces the designer to re-think that aspect of the solution, and may lead to a change of perspective. As a result, a breakdown can initiate a perspective-shift or a paradigm-shift in an architect’s personal design process.</i>
T6.51. Breakdowns demarcate distinct design episodes in the design trajectory.	<i>The designers report the personal experience of designing as one of moving forward in ‘fits and starts’. This suggests a series of distinct design ‘episodes’, separated in time, each representing a period of commitment to the design as it stands, and demarcated by breakdown events. Each ‘design episode’ represents the currently preferred design option, and is stable until proven flawed.</i>

Some of the architects (*Breuer, Morris, Sullivan, Johnson, McLuhan, Piano, Utzon*) talked of feeling uneasy, stressed, or tense when faced by a troubling design problem, and a corresponding sense of relief when a solution is found. *McLuhan* voiced this phenomenon when he described ‘design as a joke’ (see 6.3.1). *Morris* talked about removal of code duplication as a critical factor in relieving this kind of collective team stress. ‘When we finally get around to doing it [refactoring] you can see the release of stress’, he says. Stress also derives from fighting the solution paradigm or the implementation language. *Johnson* described reflecting on any kind of difficulty, when and where it occurs, as an important behaviour of the experienced architect:



I try to do something and it is hard—then I give in and do it a different way and suddenly everything starts to make sense... and the language ... brings me to that—I love that. —*Johnson*

T6.52. Architects experience a cycle of ‘tension and release’ when designing.	<i>There is a fundamental cyclical pattern of the personal design process that is alluded to by the tension that a designer feels about an unresolved problem, and the sense of release that resolution brings. It is informed by the designer's sense of aesthetic. It can also be depicted as states of equilibrium in the design trajectory.</i>
---	---

#### 6.6.19 Aesthetic

In design in the tangible word, aesthetics forms one leg of Vitruvius’ tripartite canon (commodity, firmness and *delight*). The architects repeatedly mentioned software aesthetics and notions of elegance, a perceived quality that guides them in assessing candidate structures, solutions and designs. The part played by aesthetics in the work of software architects is an intriguing theme. Although the existence of the phenomenon in software design is hard to deny, going beyond recognising ‘software aesthetic’ to useable definitions is difficult. ‘It’s very important but extremely hard to define’, suggests *Utzon*. ‘It is essentially intuitive’, says *Cook*, who reports being able to ‘look at something and either have an intuitive feel that this is an elegant solution or it’s not’.

The software aesthetic appears to be both subjective and experiential. ‘It is very subjective’, claims *Utzon*, who relates how he has received widely divergent responses to ‘the same architectural principles’ from different clients. His explanation is that different audiences have different degrees of readiness for investment in software quality, and software architecture. When *McLuhan* talks about how ‘in an elegant solution things fit together well’ he is talking about his model of how the objects are interacting in the executing solution. Primarily, he looks for ‘a degree of elegance associated with use and usefulness of it’. For *Howard*, an ‘elegant’ domain model is generic, abstract, and as a result of this abstraction, flexible and adaptive. But his abstractions must be grounded in actual examples. Elegance may be correlated with clean, simple mappings between domain concepts and solution model concepts. *Johnson* describes ‘elegance’ in software architecture as being implied by a transparent mapping—‘pretty much everything in the

UML and in the classes had a nice clean correlation in reality’. *Morris* was clear on his understanding and use of ‘software aesthetics’—he describes ‘listening to the code’ as an intrinsic part of doing emergent design:

Doing emergent design is doing as much design as I am confident of, and then trying it out, and then modifying what happens based on the feedback that I get from the code.  
—*Morris*

*Morris* uses his aesthetic sense to regulate the rate that he modifies his source base. ‘Things then get a little bit new-age-y’, he says, ‘because you start to talk about ‘listening to the code’ and listening to what the system is telling you’. *Morris* explains what he means by ‘listening to the code’ in terms of his model of learning. The model has three different phases which he transitions through when he learns something new. First, he learns something to a point where he can reproduce what he was told. Next, he achieves sufficient understanding to be able to reproduce the task without anyone’s guidance. At this stage, he cannot necessarily explain what it is he is doing—he may ‘just do it’. At this level of maturity, he describes being able to ‘listen to the code’. Finally, he reaches a level where he is aware of his own internal processes and reasoning. At this level, he can explain it in a way that he can educate others.

Elegance may be associated with usefulness and universality of application. For example, *Cook* typifies ‘elegance’ in software as a characteristic of a system that allows it to fit a range of related problems, with minimal change or adjustment. The factors that led to one of his most successful projects included the ability to ‘continually take problems and re-evaluate them in the context of what we’d already done’. The software architecture was lenient in the face of new uses—it ‘accepted’ new applications with minimal change. ‘To me, that’s an indication of elegant software... with very little change, you can incorporate new concepts or solutions to new problems’, he summarises.

#### 6.6.20 Habitation and aesthetic

Some of the architects associated habitation and the software aesthetic. There is an analogy between occupying a physical designed space (such as an architect’s building) and occupying a designed conceptual space (a software architect’s system architecture). *Utzon* draws on this analogy strongly:

I think that, when people walk into a building that really works, they know... and I think people know that about software as well... when they start walking around a software design, they know. —*Utzon*

*Johnson* describes a tangible sense of occupancy of software designs and code. ‘I have come up with designs that I don’t want to ever go back to, I don’t want to touch the code, I am scared to go there’, he admits, ‘I don’t want to occupy that space’. By contrast, his ‘good’ designs yield a positive sense of habitation. ‘I love that space, I love going back’, he says, ‘I can go there and I can read it, I can make changes and I know what I am doing’. *Johnson’s* aesthetic sense appears largely motivated by readability, conceptual and structural clarity, and tidy source.

The experience of habitation of software and built-world structures are similar but not identical, and it is likely that some people have one but not the other. ‘We all have an innate sense of what makes us comfortable in a living space... we do not all have that sense of what is comfortable and livable inside a piece of software’, claims *Morris*, drawing from his extensive mentoring experience. He draws the conclusions that ‘some people aren’t suited to software development’ because ‘they don’t have that sense of what I call ‘software aesthetics’.

#### 6.6.21 Application of aesthetic

*Morris* reports that software aesthetic becomes useful in the context of a team:

I try to teach people to make technical decisions and listen to the code and appreciate the software aesthetics, but that is what they can do as individuals... but I try to place as much emphasis on teaching them to interact positively with the rest of the team, as I do on learning to appreciate the software aesthetics, because one is important in the solo environment and the other is critical for the team environment. —*Morris*

Although the architects talk about eliminating duplication and striving for simplicity, elegance is not the same as simplicity. ‘I look at something’, *Cook* states, ‘I either have an intuitive feel that this is an elegant solution or it’s not... and when it’s elegant... whether it’s considered simple by one person or not, it’s the right solution’, he proffers. The ability to share concepts, and to jointly see concepts in a design or code with others, is one important signature of architectural elegance. The preservation of an essential theme in a structure or architecture, or the absence of its corruption, is another signifier of elegance. ‘The less the designers and the programmers corrupt the theme... the more successful your architecture is’, suggests *Utzon*.

Some of the architects correlate software aesthetic with a sense of ease experienced by the architect in the design act. For *Cook*, the experience of not being able to distil a problem into a set of statements or principles suggests that he does not understand the problem

sufficiently well. The architect's sense of 'elegance' in a design may also be inversely proportional to the amount of perceived compromise that he has made in realising the design. *Cook* observes that, while his and another architect's definitions of what seems like an elegant solution may not be all that different, their willingness to compromise on certain issues may be very different. 'I may consider some things to be more important than others, and that will bias my decisions', he says. For *McLuban*, elegance is 'very important' because 'that's the part of making everything fit properly... in an elegant solution, things fit better'. He regards elegance and 'things fitting together' in a dynamic sense—in the conceptual machine described earlier.

Several of the architects talked about symmetry contributing aesthetic qualities to architecture, and of it being an indicator of both elegance and simplicity. 'Symmetry is a key issue in the elegance of the solution', offers *Breuer*. Many problems have an underlying structure, and the skilled software architect strives to uncover and exploit the natural symmetries in the structures of both problems and solutions. Symmetry is important in software structures because it typically reduces code volume. *Breuer* describes how he has been able, at times, to take a view from a particular perspective of the business domain or the problem that resulted in a solution that collapsed down into a grammatically simple structure. For *Breuer*, the fact that it does collapse into something so elegant 'means that you have to be right... this has to be the way to go'. Elegance, or in this case symmetry, confirms correctness. *Breuer* goes further than the other architects on this point—'symmetry is, I suppose, one of my guiding lines... if something is not symmetrical then I have probably got something wrong'.

T6.53. The architects describe being guided by an aesthetic sense.	<i>The architects describe being guided by what they refer to as an aesthetic sense. This relies on a combination of inputs on design elegance and quality. Software aesthetic appears to be correlated with symmetry. It is important in the architect's personal design process because it drives them toward design decisions and interventions.</i>
--	---

## 6.7 Conclusion

To commence the interviews, the architects were asked to define their understandings of software architecture and design, their role, and their attitude to methodology. Although each architect told a personal story, all of the themes reported here were voiced by several or many. Some of the findings are surprising—for what they say about the practice of software-architectural design in Australia, or for the certainty with which they were stated.

### 6.7.1 Definitions and Context

The first three topic's subject matter overlaps considerably. For example, discussion of the context of designing occurs in the 'What is software architecture?' map, the 'What is software design?' map and the 'Architect's role' map (Appendix F). In most cases, the themes in each of the three topic maps provide a different perspective on the same aspect or phenomena, so the maps reinforce each other. In general, the analysis does not contradict the conventional view of the software architect but enhances it with detailed and rich descriptions of activity in context. The architects think about 'software architecture' in process or product terms interchangeably. They regard architecture as motivated by many concerns over and above the technical, including the preservation or disruption of the existing systems equilibrium and requirements negotiation. They see its purpose as going beyond software structure and quality to include team boot-strapping, distribution of work, selective de-skilling of the construction team, and transfer of project risk. They regard software architecture as fundamentally creative work, independent of the domain or level of abstraction. They use software architecture as a vehicle to both commit and defer project and design decisions, and they report the tension between rational methods or processes, conventional planning regimes, and the creative activities of design.

In contrast to 'software architecture' they think about 'software design' in terms of filling out the architecture, or as the resolution of forces presented by the unresolved architecture. They do not distinguish between how they approach software design and architecture, citing abstraction skills and previous exposure to design solutions as the key enablers of a design capability. Individual architects generally express a preference for either doing design 'up front' or for allowing design to emerge as they code—however, those who prefer emergent design admit to conceptualising designs in their minds (sometimes in a semi-formal grammar or model) ahead of coding.

They see the software architect's role primarily as a designer, but also as a team leader, facilitator, salesperson, and mentor. They see the architect as having a responsibility to

design for the construction team they have at their disposal rather than to pursue perfection in design. They admit a number of ways that their personal biases can influence their designing, including career investment and acquiring new skills.

Almost universally, the architects expressed disappointment and disillusionment with methods and methodology. They do, however, cherry-pick techniques from methods and use these, often appropriating them as their own. The most useful techniques are those that can be applied across business domains, technologies, levels of abstraction, and changing technologies. Figure 13 presents a depiction (in the style of Gasson's model) of the architect's context as they related it.

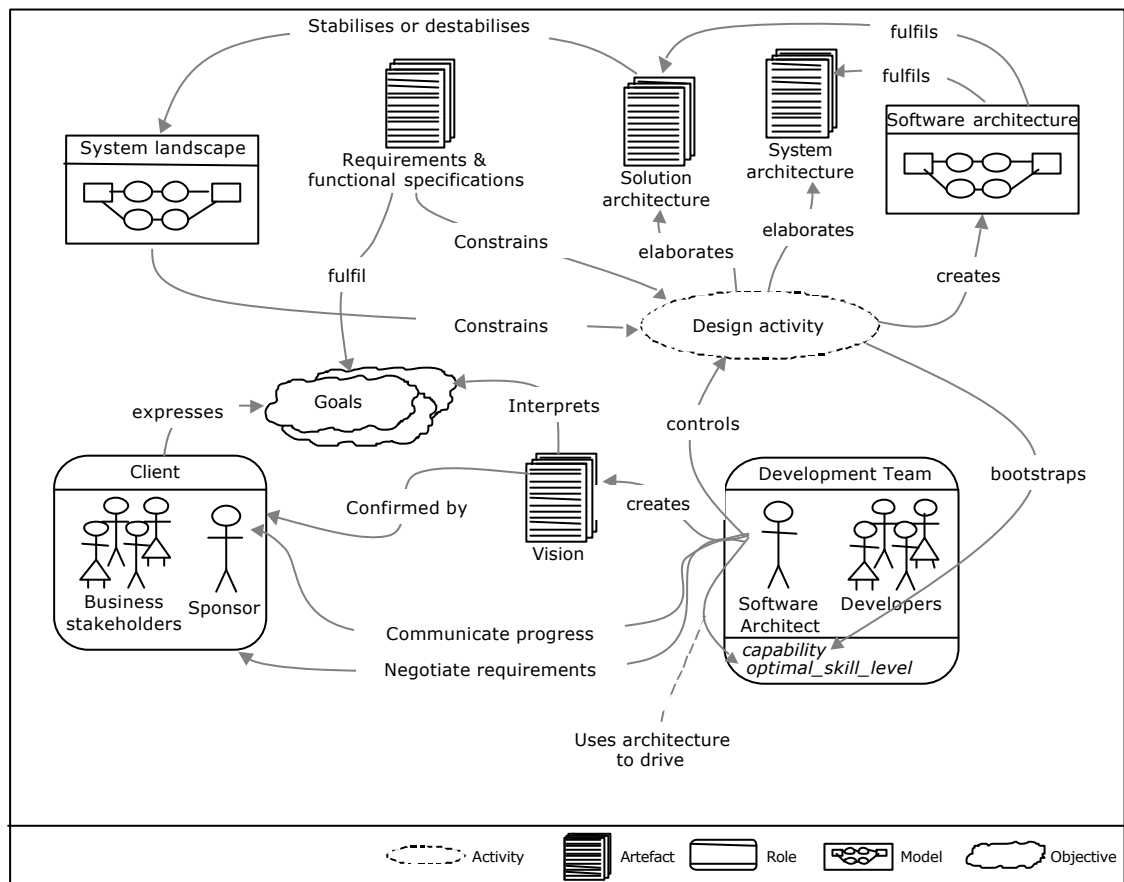


Figure 13: The architect's design context as described by the architects.

### 6.7.2 Design act

During the interviews, the architects were asked to describe how they approach the design of software architecture, their personal design process, and their most successful software architectures. In describing their approach to design, the architects talked mostly about abstractions and abstraction skills, patterns and archetypes, dealing with complexity, the relationship between problem and solution spaces, and various forms of bias.

Starting with analysis of the problem space, the architects again showed a propensity to negotiate problems to match known problem types, reconceiving problem definitions where possible to match their experience. During this problem negotiation period the architects also assemble their personal design process based on situational factors and forces. Their opinions on the importance of problem-solution transparency diverge. As they move quickly into solution considerations, the architects adopt various ways of managing solution complexity, including relaxing and tightening solution space constraints, and employing system archetypes from their repository of experience. Some agreed that the architect's design skill is partly revealed by how well complexity is addressed.

The process of software design naturally expresses a language of the solution in the form of an ontology—patterns, abstractions and models. As the design proceeds, architectural reduction often signifies progress as redundant parts of the architecture are re-factored or eliminated, and like parts are converged. The architects use a range of views on architecture (conceptual, static, dynamic, historical) with some expressing a personal preference for static or dynamic views. The architects seldom invest in evaluating options but instead rely on personal judgement to select a solution option quickly.

Abstraction skills are paramount in the architect's reflections on designing. Architects consciously select abstraction levels for their architecture's purpose and its consumers. Architects abstract with equal skill in business, software and process dimensions, moving flexibly between levels of abstraction and again relying on their personal experience when evaluating candidate abstractions as well as abstraction quality. Architects use abstraction to exploit similarities to achieve simplicity.

The architects were well aware of bias, and named perspective and paradigm as two sources. There is little evidence of architects capitalising on bias by consciously considering or shifting perspectives to identify options and evaluate requirements.

Some of the architects were able to abstract simple archetype representations from complex software architectures, and to use these in a kind of narrative fashion for description and communication of their architectures. These archetypes constitute a frame or schema for organising knowledge around solution design. Several architects were aware of a small number of archetypes and admitted reusing them frequently across engagements and business domains. As well as interpreting problems, some of the architects described actively interpreting problems in search of a fitting archetype. Archetypes are strongly related to the underlying technology platforms.

The architects talked of relying on 'personal patterns' (solution fragments and design

know-how) when designing, which turn out to be orthogonal to published design and architectural patterns. The architects remember these ‘personal patterns’ in an associative fashion (that is, for their benefit in design situations past) rather than via any kind of formal or informal taxonomy. There is evidence that the architect needs strong commitment to the model or architecture to promote its adoption. When designing, the decision-making model is coupled with the notion of breakdowns and the intuitive leap. The architects identify both hard and soft exceptions as breakdown events. A breakdown may initiate a shift in perspective and/or paradigm that leads to a design re-conceptualisation or discovery. Breakdowns also demarcate episodes in the design’s trajectory or stable intermediate states in the design’s maturing. Some of the architects report experiencing a cycle of ‘tension and release’ when designing—tension builds to a breakdown event which precipitates an alternate option, view or concept with an accompanying sense of resolution.

The architects are almost universally guided in their decision-making by a sense of software aesthetic. They talk of ‘listening to the code’ and ‘knowing’ when a design is right or not yet right. The architects found it difficult to objectify their notion of software aesthetic, but symmetry appears to be one important ingredient. Figure 14 presents a depiction (in the style of Gasson’s model) of the design act as the architects related it.



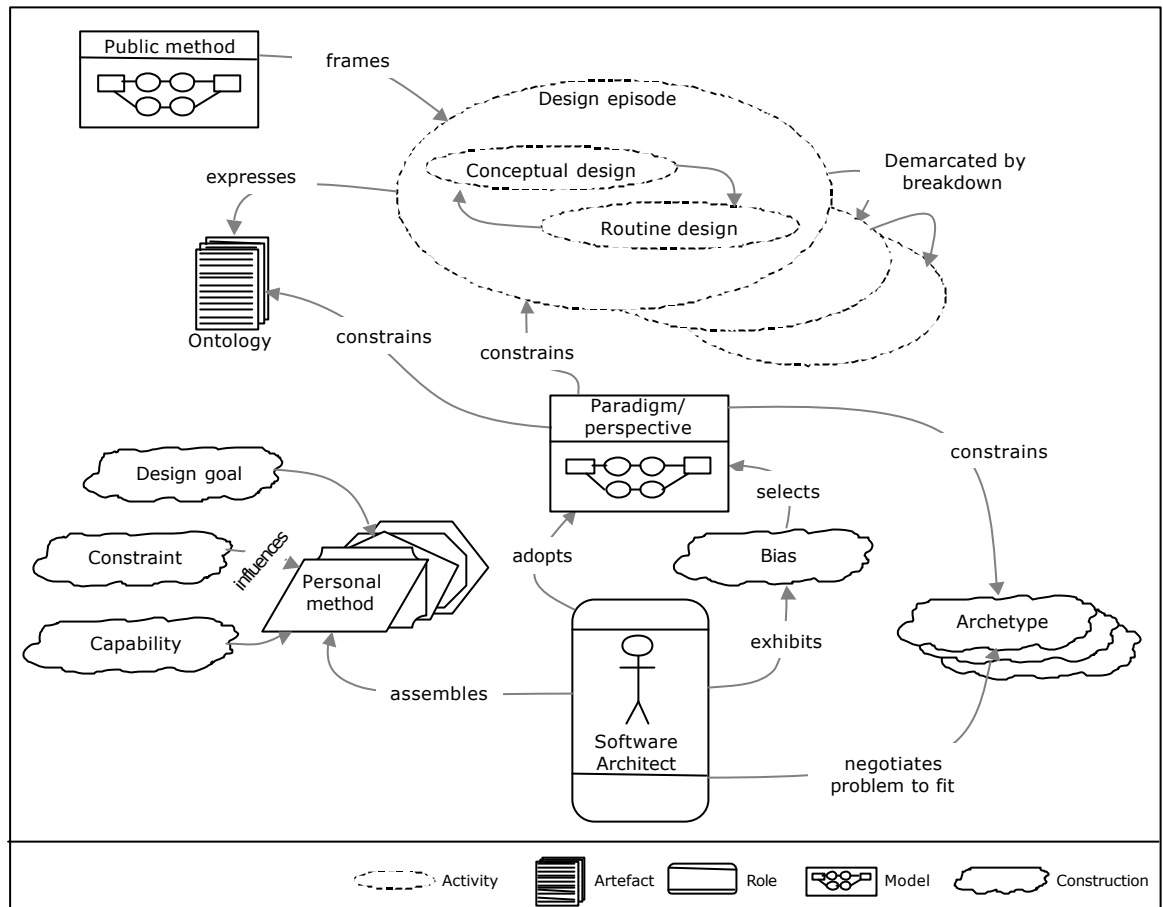


Figure 14: The 'design act' as described by the architects.

This completes the thesis' grounded theory profile of the practice of software design. The next chapter builds on this by exploring two significant case studies.

## Chapter 7: Case Studies in Situated Software Design

If you want to build a factory, or fix a motorcycle, or set a nation right without getting stuck, then classical, structured, dualistic subject-object knowledge although necessary, isn't enough. You have to have some feeling for the quality of the work. You have to have a sense of what's good. That's what carries you forward. This sense isn't just something you're born with, although you are born with it. It's also something you can develop. It's not just 'intuition', not just unexplainable 'skill' or 'talent'. It's the direct result of contact with basic reality. Quality. Which dualistic reason has in the past tended to conceal. (Pirsig 1974, p. 277)

### 7.1 Introduction

This chapter presents two case studies to further illustrate the nature of situated software design. Each case study is drawn from the accounts of one or more of the interview participants, and describes a project conducted in Australian private and government organisations between 1997 and 2004. The case study is a form of documented observation that allows the researcher to describe and explain complex social phenomena. Case studies yield holistic and meaningful characterisations of real-life events (such as individual life-cycles, or organisational and managerial processes) and are well suited to research questions that ask 'what' and 'why' particular phenomena occur (Yin 1994). Case studies should primarily report grounded, practical, factual results, as distinct from the interpretations of the results. The validity of generalising from one case study to theory is supported in some cases (Shanks et al. 1993; Yin 1994). Yin (1994) recommends that a well-designed case study should have five components—a motivating question, its propositions, a declaration of the unit of analysis, a way of linking the logic of the case study to the propositions, and criteria for interpreting the findings. These elements are stated for each of the case studies.

## 7.2 Case 1: The Naissance of the Decision Tree

The first case study examines the origins and design of a software-architectural component in a large business system development project. The invention and introduction of this component made a substantial contribution to both the architecture and the overall viability of the project. The designer of this component was the participant *Breuer*. He cites several of the project's senior designers as having acknowledged that the introduction of this component saved the project from impending termination by eliminating a large amount of user interface, application code and relational database structures. The case study highlights how the design came about, what factors impacted the component's design, and how the interplay between the senior designers—each working from markedly different perspectives—influenced the designer and directly shaped the component's design. That these design episodes occurred independently of the project's prescribed methodology and design process is a significant finding in itself that serves to illustrate the disconnection of project-level method or process from the events that initiate design episodes. The adoption of the design necessitated a technological and cultural shift in the direction of the project, and the story of how the component was adopted across the project illustrates paradigmatic shifts in project cultures at times of impending crisis. *Breuer's* reflective account also serves to highlight the importance of being prepared to work in different modes of thought and communication in achieving acceptance and adoption of his design.

The following declarations about the design of this case study satisfy Yin's (1994) guidelines for case study structure, and the preconditions for generalisation of the results. The motivating question for this case study is simple—do the defining features of the case study confirm or refute the characteristics of situated design as presented in the previous chapter and also in Chapter Five's design assessment framework, particularly those in the planning, generators, collaboration, control, process and method dimensions? The proposition is that the case study confirms the relevance and validity of the grounded theory assertions and the framework. The unit of analysis is a group of designers working collaboratively.

### 7.2.1 Actors and Roles

The case study narrative is related by *Breuer* who acted in the role of the software architect for the enterprise services component of a large financial services project. *Breuer* was a consultant supplied to the bank from the vendor whose software product was being used

for the development of these services. Three other senior designers appear in the account. As they were not interviewed for the main part of this study they do not already have pseudonyms, so they will be referred to as *Chen*, *Nygaard* and *Goldberg*. At the time of the case study, *Chen* was an author and consultant data modeller of international repute assigned to the project to oversee the database design. *Nygaard* was a senior consultant and trainer with fifteen years of experience in object-oriented system development. *Goldberg* was a member of the organisation's staff with twenty-five years of systems development experience. As a result of their individual and collective experience, the foursome constituted a *de facto* design team that made most modelling and design decisions immediately prior to implementation.

### 7.2.2 The Business Context

The project was charged with designing, developing and delivering a large asset finance system for a leading Australian bank in the period from 2000 to 2003. The project had the ambitious agenda of redeveloping an entire suite of legacy applications for which experienced programmers were beginning to become scarce. Another motivation was the perceived need to be able to deploy browser-based applications to both the external sales agents and internal branch and head office staff. The desire to replace the mainframe with relational database and application server technologies was also strong.

The project had a budget of approximately \$A30M and peaked (in terms of headcount) in February 2000 at nearly one hundred people. Teams were organised around business analysis and domain modelling, the design and implementation of HTML and Java Server Pages, application services, and database and transaction design. A significant commitment was made to the project-wide use of a commercial object-oriented development methodology (Mentor (Edwards 2006)) and to the use of the Unified Modelling Language (Rumbaugh et al. 2004) as the common modelling notation. The project applied UML structuring concepts by dividing the business domain into about ten functionally-bounded packages. The packages defined teams as well as functional scope, and the vertical division between teams was strongly enforced by project management processes.

Like most contemporary business systems, a requirement existed for the system to be highly customisable and responsive to the rapidly changing financial services marketplace. This flexibility was regarded as critical to the project's success, particularly in the areas of products, contracts and commissions. A 'product' is a financial service offered to clients

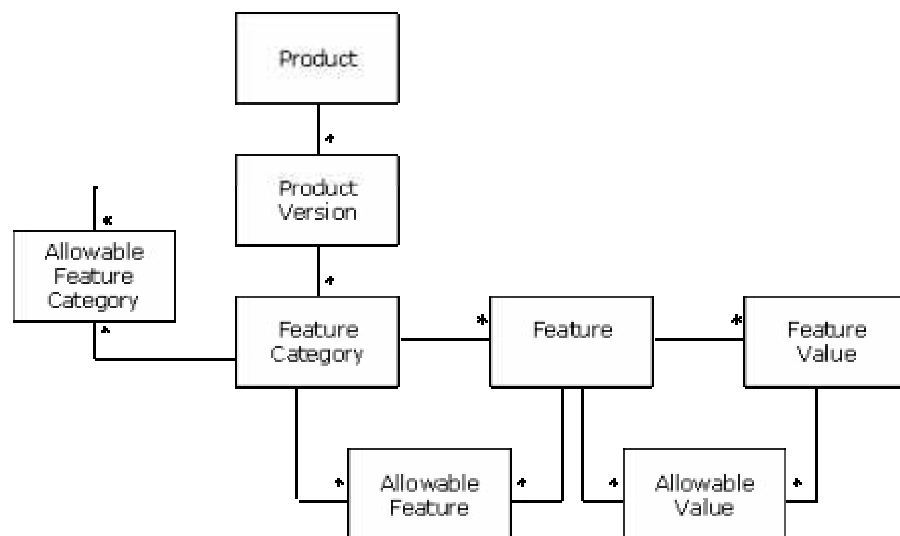
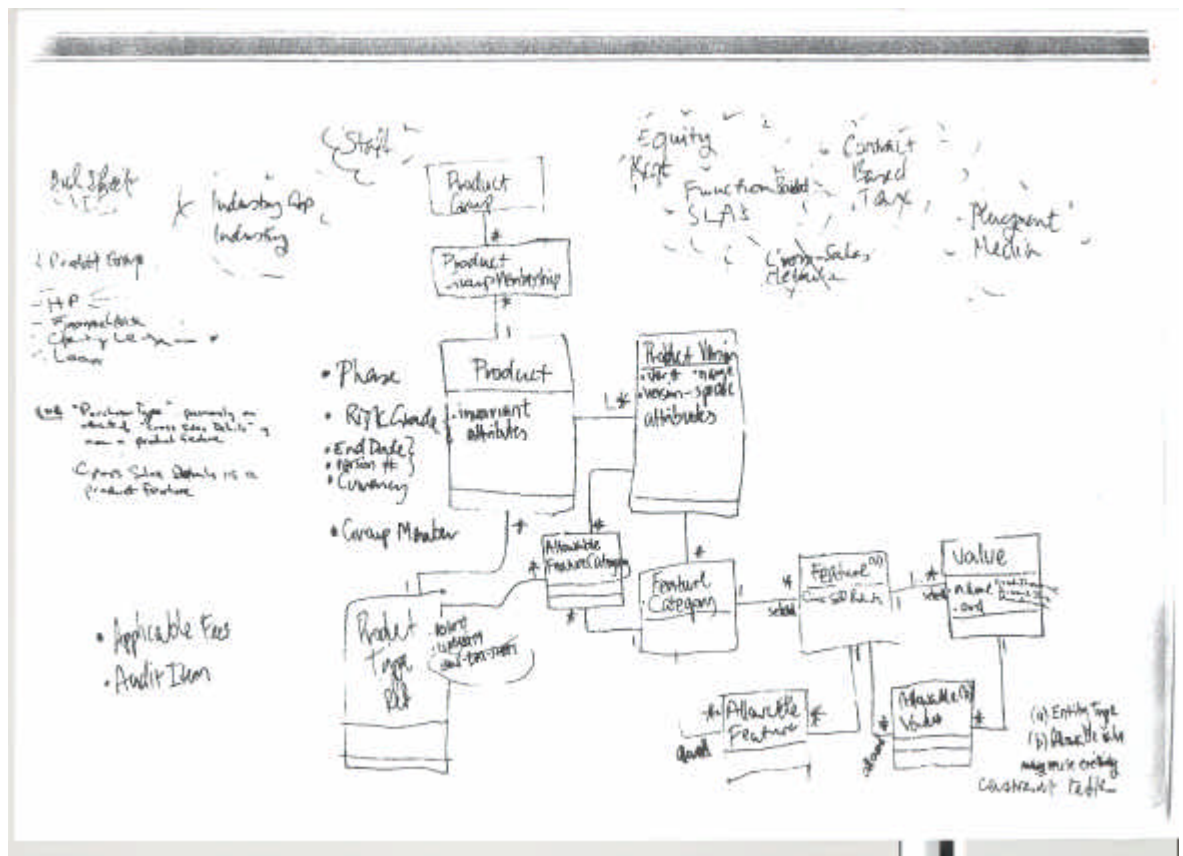
via agents or dealers. At the time of the project's initiation, the bank offered several hundred separate products. Many products were related and many shared attributes. There were some products for which successive versions differed only by the addition or deletion of an attribute. A 'contract' is the binding agreement between the bank and a customer that defines the terms and conditions of a product's sale. Agents typically strike contracts with new or returning customers by starting with a generic contract and modifying its clauses or conditions during the negotiation process. The result is that very few actual contracts are identical. 'Commissions' are arrangements put in place between the bank and each agent that define the amount of commission the bank pays to the agent upon sale of each product. Like contracts, individual commissions are struck by modifying the attributes of a template commission during the negotiation process between the bank and an agent.

The case study concerns the design of the Product package, a key component of the application for two reasons. Firstly, the business administered several hundred products at any time and the product configurations controlled how a product could be sold and what revenue followed. So an accurate and unambiguous representation of all products was critical to the successful operation of the business. Secondly, products had to be very flexible—a product might be in use for anywhere from three weeks to ten years, during which time it might undergo weekly changes. The design of the Product package had to support flexible creation, editing and management of products within this volatile business environment.

The designers determined that a product should be defined by instantiating from a template product, and that each product would need to be a composite of parts. Conceptually, products are a kind of assembly—each is a composite of parts, and each part can be configured in a number of different ways. Non-composite parts of a product are its attributes—examples include applicable sales regions, sales restrictions, applicable asset classes, finance limits, security types, draw-down and redraw options, and the usual identifying attributes. Defining a new product involves more than simply configuring its attributes and composing parts, because the allowable set of parts that may be used in the same assembly is governed by business rules (or dependencies) and constraints that exist between both the parts and composites. A conceptual model of product would therefore need to support cloning specific products from generic product templates, the assembly of products from attributes and parts, the version history of a product family, and the dependencies between parts, product parts, and products.

[illegible]

Following the unsuccessful review the Product package was picked up by *Nygaard* and *Goldberg*, who spent half the following day working from scratch on a new model. After several hours they had a candidate model (Figure 2) which was unfinished.



**Figure 16: Product model redrawn (second design episode).**

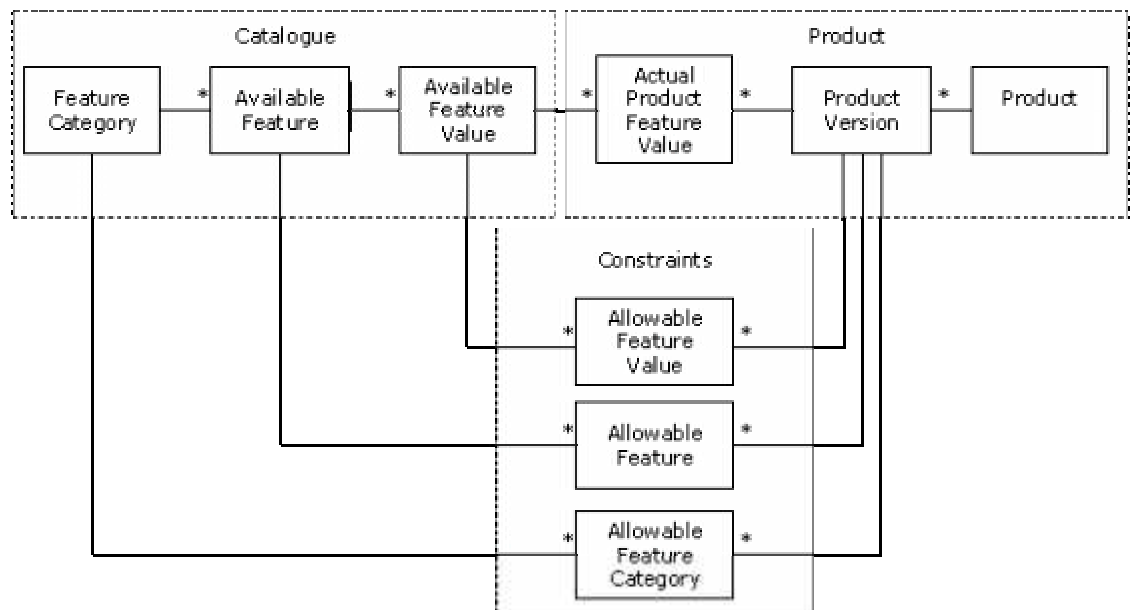
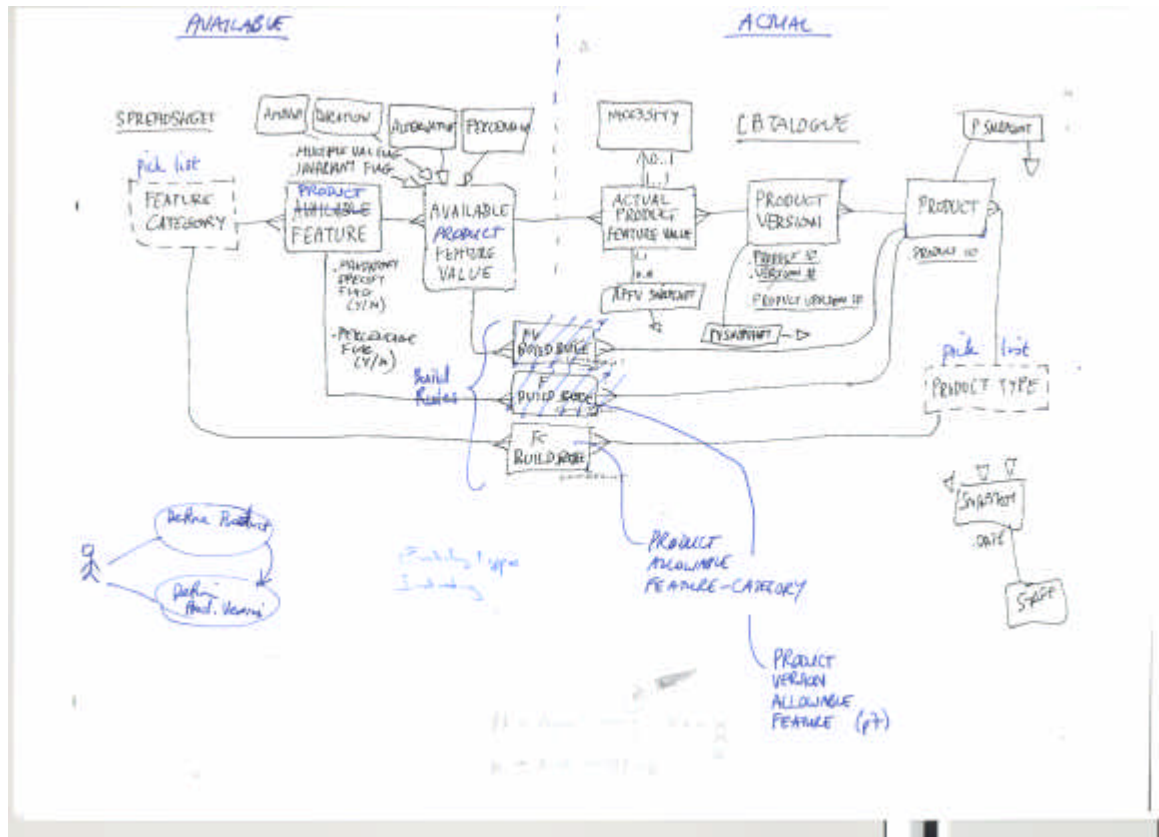
The model exhibited one important additional abstraction—the introduction of a class (`Feature`) that replaced all domain-specific feature classes in the first-cut model. This meant that each feature would be represented by data, allowing the required degree of flexibility of features. The model also accounted for multiple versions of products

(‘Product\_Version’ class) but it did not model a convincing representation of the constraints between features and other features, or features and products. This version of the model addressed the most obvious shortcomings of the first version but *Nygaard* and *Goldberg’s* modelling effort slowed when no obvious solution could be found to these remaining problems.

#### 7.2.4 Design episode 3: A data-oriented alternative

The next day, *Chen* joined *Nygaard* and *Goldberg* in an informal meeting to talk through the progress on the revised model. Before *Nygaard* or *Goldberg* had had a chance to explain their model and its shortcomings, *Chen* (who had not seen their model) interjected with an offer to propose an alternative model. He walked up to the whiteboard and immediately drew up the essence of a new model, based on a central idea that each specific product should be thought of as a set of pointers or references into a catalogue of product features. His alternate model (Figure 17 depicts both his original sketch and a re-drawn model) clearly demarcated between the product features in the catalogue available to be used in a product (AvailableProductFeatureValue) and a second class which modelled the inclusion of a particular AvailableProductFeatureValue in a product (ActualProductFeatureValue). This model variant allowed independent control over the catalogue of product features and their values that the *Nygaard-Goldberg* model did not. It was enthusiastically received and the design team agreed to adopt *Chen’s* model.



Figure 17: *Product* design—data-oriented version (third design episode).

Breuer recalls that the other designers were impressed by *Chen's* seemingly instantaneous modelling effort. Later, *Chen* revealed that the essence of his model (which he referred to as the 'catalogue-order' model) was a pattern that he had successfully used in data models several times before. When asked whether he had read this pattern somewhere or

discovered or developed it himself, *Chen* felt strongly that it was one of his designs. *Chen* was able to recount in detail its reification in a data model he had designed several years earlier for a government transport organisation. This familiarity enabled him to reproduce it with impressive speed.

One problem remained with *Chen's* catalogue-order pattern, however. The constraints that served to limit which feature category, feature, or feature value could legitimately be used in any given product still needed explicit representation in the model. *Chen* decided to model these as entities that resolved the many-to-many relationships between feature types and a product version as per data modelling convention. These entities took the prefix 'Allowable' to indicate that each instance represented one allowable combination of a product and a feature category, type or value. Although this clearly worked, *Chen*, *Nygaard* and *Goldberg* went away thinking that a complete solution had yet to be determined, and further work would be needed to find a satisfactory representation of the constraints.

#### 7.2.5 Design episode 4: A rule-oriented solution

While this model appeared highly suitable at a business modelling level of abstraction, *Breuer* was not convinced that he wanted to implement these constraint classes in the application services and the database. *Breuer* could not see how an implementation of these constraints could be achieved with the same degree of elegance exhibited by the model's solution to the catalogue-order part of the problem. *Chen's* model appeared to *Breuer* to have been created from a data perspective and it gave the developers no assistance in knowing how to implement the constraints.

*Breuer* argued back and forth with *Chen* over how to best represent the constraints whilst still preserving the catalogue-order pattern in the model. The next revision and the shift in perspective that allowed its conception involved *Breuer* in isolation. He recalls the instant at which the seed of an alternate design formed in his mind:

Chen got upset in a meeting, and threw his hands in the air and he uttered a key phrase... and that key phrase was—'all I need to be able to do is to transport a set of business rules from the back end to the front end and have them execute'... that was the key content of what he threw out during that meeting. —*Breuer*

*Breuer* recalls that in the first instance he took *Chen's* comment as a criticism of his company's product (which was being used to develop the enterprise services). Even before *Chen's* comment, *Breuer* had been well aware that the issue of how business logic should be allocated across the application's tiers had proven troublesome—he believed that it

remained an unresolved and architecturally dangerous problem. ‘When he put it like that he encapsulated and put precisely what the requirement was... and he was right, we had to put rules in somewhere, and make those rules execute on the front end’, he recalls. To that point in time, *Breuer* claimed, the models had been produced from a data perspective and this had meant that the importance of the business rules (the constraints between features) had been overlooked. ‘It wasn’t a data relationship problem’, *Breuer* claimed, ‘it was a business rule representation problem’. Next, *Breuer* produced his own model which addressed this perceived imbalance. He recalls what happened next:

I went home and thought about it at every level... I don’t recall *not* watching television... yes I do. I think I just fiddled with some papers, played piano, did something, let it lie fallow for a bit, but it kept on re-echoing... I know I probably didn’t go to sleep very early, it was probably more like 3 o’clock in the morning... *Chen* had encapsulated the problem exactly, and I was predisposed to thinking about representing business rules in a hierarchy... it kind of seemed just obvious from experience. —*Breuer*

*Breuer* decided that if the abstract concept was one of decisions, any business rule must be conceptualised as a hierarchical collection of decisions. The resulting architectural component that he designed was named the ‘Decision Tree’. Its class model (Figure 18) illustrates the concept. The tree stores feature values and each feature’s sub-tree of dependent features. A single self-referencing class (`ProductRuleElement`) implements the recursive concept of a decision leading to further decisions. The Decision Tree is best explained with an example. A loan product might be represented with a tree of about twenty nodes. The root node is the first feature that needs to be resolved in navigating the loan product tree and each allowable value forms a sub-tree below the root. If the root feature is the type of loan, then the allowable loan types (say, ‘interest-only’ and ‘interest and principle’) each form sub-trees. The structure is executed by traversing the tree from the root to a leaf, and presenting the question at each node to the user. Interrogating a product represented as a Decision Tree involves a single traversal of the tree from its root to one of its leaf nodes, based upon actual arguments or selections made by the user. Each completed traversal results in a vector of values that define the user’s selection of all mandatory and optional features at each node in the trajectory.

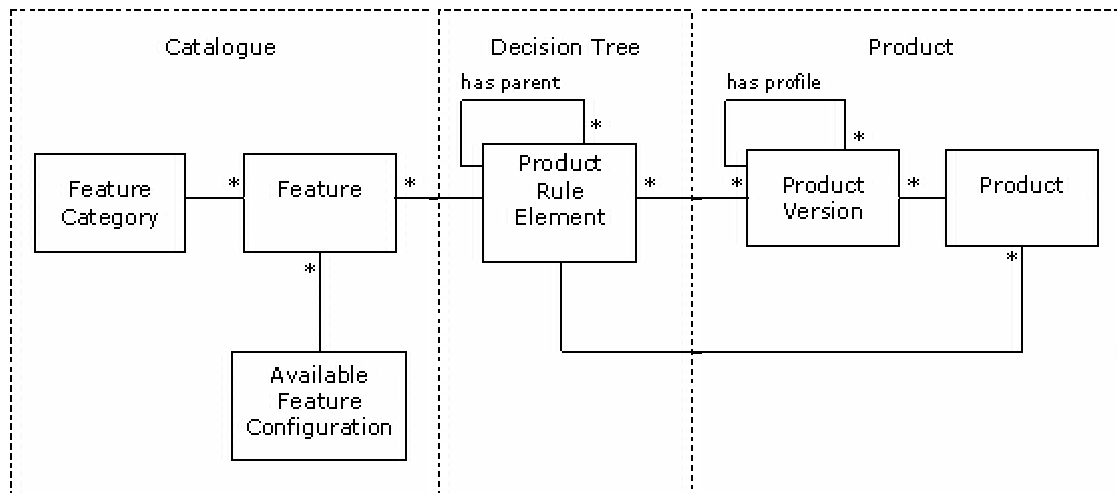


Figure 18: Product design—rule-based version (design episode four).

The design team collectively felt a penny drop as the implications of the Decision Tree began to dawn. ‘Everyone who I talked to (who actually understood the issues) took it as if the sun had just come out’, *Breuer* remembers. Firstly, use of a tree structure solved the problem of enforcing dependencies between features with a simple tree traversal. The solution preserved all the benefits of its predecessors. Like the *Nygaard-Goldberg* revision, its features, categories and values were all stored as data rather than as domain-specific classes. And like *Chen’s* catalogue-order pattern, the set of available product features was kept distinct from the actual (used) ones.

Further advantages of the Decision Tree were realised in its implementation. Firstly, *Breuer* proposed to store the Decision Tree in a single database table and to write code that generated the tree as an XML document, which would be transformed into HTML for transmission to the user’s browser, where it would handle the user-entered or selected results. Thus the user interface required to present a product could be automatically generated. Up until this point, the project had planned significant effort to have this part of the user interface hand-coded. The use of the Decision Tree meant that all of the product’s attributes, and all web forms that presented and prompted the user for values or selections could be entirely data-driven. To make the Decision Tree even more compelling, *Nygaard* and *Goldberg* started finding additional applications of the Decision Tree in other functional areas of the system beyond products—agreements and commissions quickly became candidates.

## 7.2.6 Designer's reflections on the design process

*Breuer* talked about the origins of his idea. He had perceived a propensity amidst the project teams to slavishly adhere to the conventional two or three-tier business application architecture (a relational database with business logic in the client or application tier). He had to work hard to undo the implicit biases of his developers toward this orthodoxy:

I think there are a lot of myths... I had to undo a lot of the developer's myths too... [the myth that] the whole of the system should have been built into a very simple set of database tables... because it is not data-intensive, it is business-rule intensive.' — *Breuer*

In the history of the project there had never been an architectural design phase in which some of the fundamental technology issues had been considered from an architectural perspective. As well, parallelism of the package-based development process unnecessarily exerted pressure on the implementation and database teams and did not allow for their discoveries to feed back to the rest of the project. The widely varying capabilities of the large number of developers additionally confounded any efforts from the design team members to effect change. *Breuer* was convinced that to have proceeded with the entrenched development process would have lead to inevitable project collapse:

This can't be right, it's too complex... we'll never finish implementing, because as soon as somebody changes something it will be hopeless to undo, because we don't understand all the interconnections... so what is the simplifying principle? That simplifying first principle was—'business rules are a tree'... that was the elegant, simplifying concept. —*Breuer*

His conviction that the Decision Tree was the right solution was largely based on a personal belief in his notions of structure and correctness:

The fact that it did collapse into something so simple and elegant meant that it *had* to be right, this had to be the way to go... one of my guiding lines here is that if something is not symmetrical then I've probably got something wrong. —*Breuer*

Adoption of the Decision Tree on a project-wide basis had the potential to remove a sizeable component of analysis, domain modelling and screen prototype work in progress. The decision had to be made quickly, if at all. *Breuer* referred to this project-wide sense of momentum with the current solution approach as 'a number of locomotives bearing down the tracks at us'. *Breuer* worked long hours for three days to rapidly prototype the first Decision Tree. There was no time to do an exhaustive examination of the Decision Tree concept, as the project was heavily staffed with a number of implementation teams that had been under-utilised for some weeks. This had focussed attention on the design team,

which was now perceived as a significant project bottleneck. *Breuer's* initial prototype demonstrated the storage of a representative product tree, identified the types of nodes required (enumerated types, display-only, boolean, unformatted and formatted data entry), automatically generated HTML and handled the user's selections and responses in an XML document. A project meeting was held a week later and a decision was made to adopt the Decision Tree as a fundamental architectural building block.

It is interesting to note that until this point in time, *Breuer* had received no direct instruction from management and was acting entirely in a bottom-up fashion. 'Someone needed to stand up at a point', *Breuer* claims, 'and state that the project's architectural approach was wrong and had to be changed'. He saw the Decision Tree as the mechanism to achieve this redirection. *Breuer* regarded that the project would almost certainly fail if left to a conventional data-oriented architecture and design approach, because the necessary continuous maintenance of the data model and business logic would be unachievable. He believed that the Decision Tree was the project's only saviour and that staking his reputation on a significant architectural decision was a worthwhile gamble. On reflection, *Breuer* assessed that his attitude toward risk was 'fifty percent self-confidence and fifty percent knowledge', and that the degree to which a designer is risk-averse is largely a function of personality:

It's hard-wired into my personality that if there's an opportunity to rip up the tracks and lay them down somewhere else—I will do it... I wouldn't call it bravery... although a lot of people do. —*Breuer*

*Breuer's* risk-taking paid off with the decision to adopt his component. The Decision Tree was fully implemented and spread as predicted into other functional areas of the system. The project team shrunk markedly in the months that followed and *Breuer* moved into the role of lead application architect.

### 7.2.7 Relationship to Grounded Theory

*Breuer's* story of how the Decision Tree design came about illustrates four distinct design episodes alternately led by each of the collaborating designers in a self-selecting network of roles. The key breakdown event is evidenced graphically by *Chen's* meeting outburst, which stimulated *Breuer's* conception of the generator that drove the conceptualisation of the Decision Tree. In each distinct episode, the breakdown initiated a perspective or paradigm shift (T6.50), from a naive perspective, to the object paradigm, then to the data

paradigm, and finally to a functional paradigm.

Each episode illustrates how the architects relaxed different constraints in order to achieve a model (T6.29). The first model ignored too many constraints to be viable but constituted a model nonetheless. The *Nygaard/Goldberg* model over-simplified the distinction between allocated and available product parts, and *Chen's* model over-simplified constraints. In all three cases, relaxing a constraint allowed creation of a model but the subsequent attempt to tighten the relaxed constraint resulted in a breakdown event which marked the end of the episode and created an opening for the next designer.

*Chen's* ability to contribute the third model variant followed from his experience with using it previously (T6.46). That he had a name for his pattern (the 'catalogue-order' model) and believed he had derived it from his own modelling experience reinforces it as an archetype (T6.45). Perhaps *Chen's* biggest frustration was that he was not afforded an environment in which to negotiate the user's requirements on constraints to fit his predisposed solution (T6.19, T6.46). *Breuer* explicitly recognised that *Chen's* (data) paradigm bias blinded him to finding a solution to the constraints representation part of the problem (T6.36) because there was no workable solution in a data model.

An aesthetic sense appears to have been a significant factor in the assessment of each option (T6.53). While *Chen's* model reduced the earlier model's introduced complexity, *Breuer's* model addressed the 'constraints problem' and, most significantly, collapsed complexity across the entire project (T6.33). Much of its appeal could be attributed to the symmetry (T6.53) of its primary generator (T6.43)—a tree—and a recognition that the project had been dealing with a number of tree-like structures all along. Viewed in this way, *Breuer* discovered the structure of the problem, and when found, the right abstraction revealed previously obscured similarities (T6.39) across the architecture and consequently dissolved complexity. The solution's simplicity ensured its adoption amongst the architecture team and revealed its designer's skill (T6.31). The fact that *Breuer's* design had profound economic consequences for the entire project (by eliminating a substantial quantity of development work) ensured its subsequent adoption across the project (T6.7).

The way the architects alternated between creative proposing and rational assessing (T6.12) can be seen in how they changed roles. In each episode the creative designing was done by the model proposer whilst the other architects adopted the role of rational assessors. What is so illuminating in *Breuer's* story is how each episode delivered such a markedly different model variant and corresponded precisely with an exchange of roles, all without any externally imposed plan, management, method, process or facilitation (T6.27).

The story further illustrates paradigm and perspective shifting on the part of the design collective rather than any one individual designer. Despite the obvious skills and experience of *Nygaard*, *Goldberg*, *Chen* and *Breuer*, none of these individuals demonstrated an ability to deal with their own perspective (T6.35) whilst operating in the creative, proposing design mode. All were able to recognise other designer's preferred perspectives when assessing that designer's proposed solution.

### 7.3 Case 2: Perspective-bias in a Telecommunications Architecture

The second case study examines two periods in the lifecycle of the software architecture of a telecommunications system. This case study illustrates how the original architect's approach was initially judged to be successful, but later unsuccessful, as a result of a change in non-functional requirements. The story illustrates the criticality of the designer's choice of perspective and the consequences of an inappropriate choice in the presence of inaccurate or missing requirements. The story also illustrates how stakeholder perceptions of the success of a system are dependent on the situation in which it is embedded and what can happen when even seemingly small contextual shifts occur.

The motivating question for this case study does not change from that of the previous one—to see whether the defining features of the case confirm or refute the proposed characteristics of situated design as identified in the framework and the previous chapter's grounded theory assertions. In addition to this goal, this case study clearly illustrates the nature of perspective bias. The unit of analysis is the solo architect and the object-oriented software architecture he produced.

#### 7.3.1 Actors and Roles

The case study involves two of the interviewed participants—*Le Corbusier* and *Mackintosh*. At the time of the project's architecture phase, *Le Corbusier* was the principal software architect and *Mackintosh* his second-in-command. Working as consultants to the client, both were engaged to lead a team on a development project that would trial object-oriented development technologies and techniques. The technology set was comprised of C++, an object-oriented database (ObjectStore), a platform-independent user interface builder (OpenUI), object-oriented analysis and design methods (Booch and Rumbaugh), and model-driven architecture (Software thru Pictures). Each of these elements had been previously used in the client's organisation with varying success, but this project



represented the first time that they were consciously selected, integrated and applied as a standard development platform and toolkit. The client also identified skills transfer and mentoring as key project objectives and composed a team in which *Le Corbusier* and *Mackintosh* were significantly more experienced than the other members. *Le Corbusier* commenced the architectural design and *Mackintosh* (who joined half way through this phase) led the stakeholder consultation. *Le Corbusier* controlled the initial design and saw it through development and into production before leaving the project. *Mackintosh* stayed on with the project and led the second phase of the architecture, which involved a substantial redesign. Both architects recounted their own personal recollections and experiences of the project in their separate interviews.

### 7.3.2 The Business Context

The case study concerns the design of the architecture of a telecommunications exchange monitoring system for a leading Australian telecommunications company in the period from 1997 to 2002. The project had a total budget of approximately \$A10M and peaked (in headcount terms) in 1998 at about twenty people. The business requirement appeared straightforward—periodically poll up to one hundred exchanges of identical type and retrieve accumulated call handling performance measures. Under some circumstances, the application would send commands back to the polled exchange. A number of related real-time control and monitoring functions formed the core of the functional requirements. Non-functional requirements such as high availability, performance and reliability were specified in quantitative terms. A handful of business sponsors were allocated to the project. The most useful and engaged of these turned out to be a couple of late-career telecommunications engineers who had worked with the exchanges over several decades. *Mackintosh* described how these engineers treated the exchanges like ‘a much-loved favourite toy’. The fact that delivery pressure on the project from the business was steady but not extreme allowed for their frequent and at times open-ended involvement with the project team.

### 7.3.3 Design episode 1: Pursuit of perfection

Known dependencies and apparently stable requirements, secure funding, dedicated and knowledgeable subject matter experts collectively set the scene for *Le Corbusier* to deliver one of his finest designs. He confidently assumed the role of lead architect and proceeded to design—largely in isolation from the other team members—a text-book object-oriented software architecture. He delegated the mechanistic and detailed exchange interface

design work and tackled the big canvas of the object model. He worked quickly, unimpeded by uncertainty or the overhead of collaboration to produce his model. By the time the solution design and the software architecture were being presented back to the engineering stakeholders, *Mackintosh* had joined the team. ‘This was probably the cleanest architecture I’ve seen’ recalls *Mackintosh*. The engineers perceived its transparency as elegance and hailed *Le Corbusier’s* design:

The architecture was elegant, simple, worked really well, it matched perfectly how the users spoke about it, we presented it back to the users, they all went ‘yeah, you understand’. —*Mackintosh*

The team transitioned into development and made rapid progress implementing the model in C++ classes, both persistent and transient. The clarity of the object model and the use of an object-oriented database (which eliminated the need for typically messy object-relational mapping code) made the transition from design to code look almost trivial:

[*Le Corbusier*] came up with a really nice structure and it worked really well, and it was one of the nicest architectures I’ve ever worked on... we mapped the problem really well, and it was implemented, class by class, just the way it was meant to be... it was why we got the next business... it worked really well. —*Mackintosh*

*Le Corbusier*, the master architect, could not put a foot wrong as his model, the development process he directed, and his team of constructors collectively ran like a well-oiled machine. Some of the model’s appeal could be attributed to its simple, uncluttered representation of objects in the domain. In effect, *Le Corbusier’s* object model laid out a map of the engineer’s domain knowledge in a way that they had not previously seen. His most crucial decision in achieving this ‘elegance’ was to centre the model on the exchange. *Mackintosh* recounts:

And that’s what’s obvious—you come from a data modelling background, which I did, you think of the things you can pick up and kick first—exchange! We’re gathering exchange data, that’s going to be the object in the middle... because I can go out and I can kick it... and we did, we went out to Footscray and we saw the exchange. —*Mackintosh*

A model that exhibited simplicity and clarity—and prominently placed the exchange in pride of place at the centre with all the other business objects, relationships and concerns radiating out from it—attracted the engineer’s admiration. During the development phase *Le Corbusier’s* object model changed little, and every aspect of how the project delivered its first version into production with impressive:

Everything seemed perfect, and the final result was... ‘you guys are brilliant—under

budget, under time, a solution that works well', its object-oriented, this is the way to go... solves all of the world's ills. —*Mackintosh*

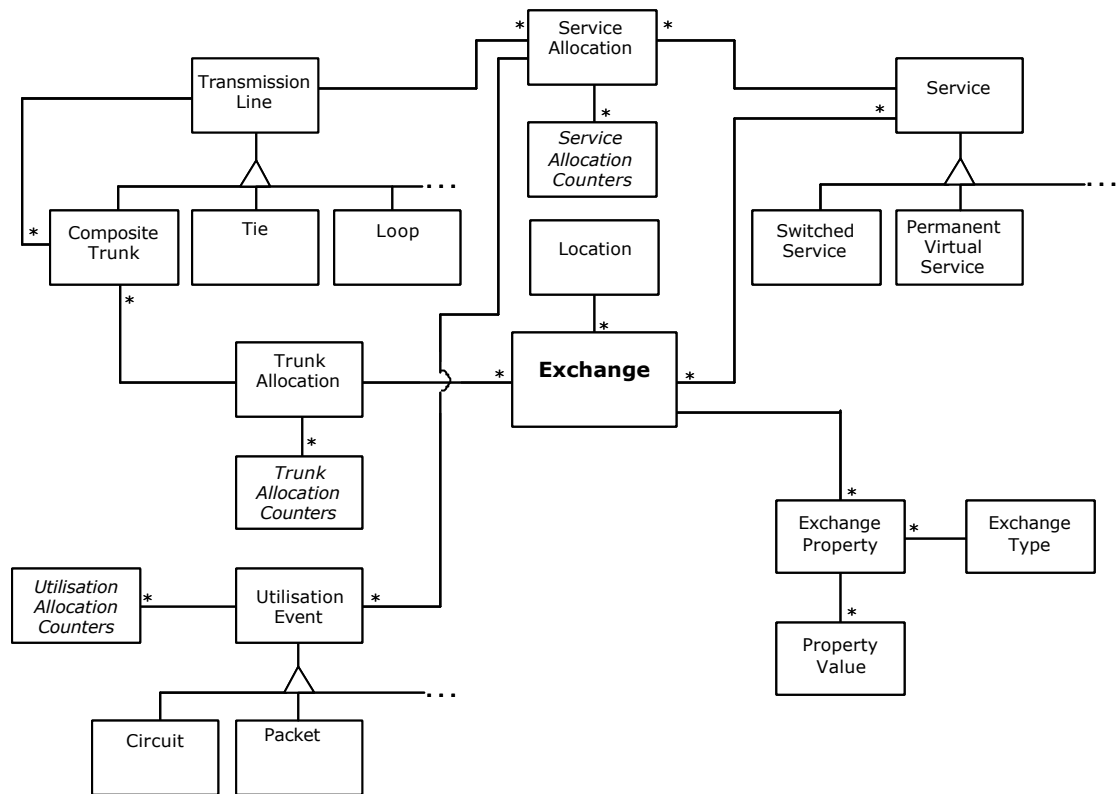


Figure 19: Conceptual sketch of Le Corbusier's exchange-centric model.

#### 7.3.4 Design episode 2: Collapse and re-conceptualisation

In the following months, the system's successful deployment drew attention to its existence. Before long, management requested the development team to add support for new types of exchange. *Le Corbusier* had moved on and the new requirement fell to *Mackintosh*. Adding support for the new exchange type appeared trivial but the implications for application performance from the increase in the number of exchanges were profound. Initial tests and predictions showed that up to ten times the amount of data coming in from the (much larger) number of exchanges was a problem, and *Le Corbusier's* architecture could not create and persist objects quickly enough. 'We couldn't have anticipated that', *Mackintosh* recalls, 'but we could have got the structure right so that the problems wouldn't have occurred'. To *Mackintosh*, *Le Corbusier's* architecture appeared 'back to front... the structure was completely inside out' in the light of the new requirement. *Le Corbusier* had assumed the luxury of perfecting an object model without explicit regard for extension of

the non-functional performance requirements.

The system had been designed to meet a requirement to store ‘per exchange’ readings sampled every hour from each of about fifty exchanges. A better solution, *Mackintosh* proposed, would be to store this hour’s readings—the approach of going to each exchange to get one hour’s measurements caused the system to do fifty queries instead of one. Until the change in requirements, the problem never showed itself because the period between samples was long. But the prospect of doing five-minute readings against the AXE exchanges (which numbered seven thousand) was daunting to say the least:

And that’s the nature of architecture—you build a house that’s a house that turns into a Bed and Breakfast, and all of a sudden it doesn’t work so well. —*Mackintosh*

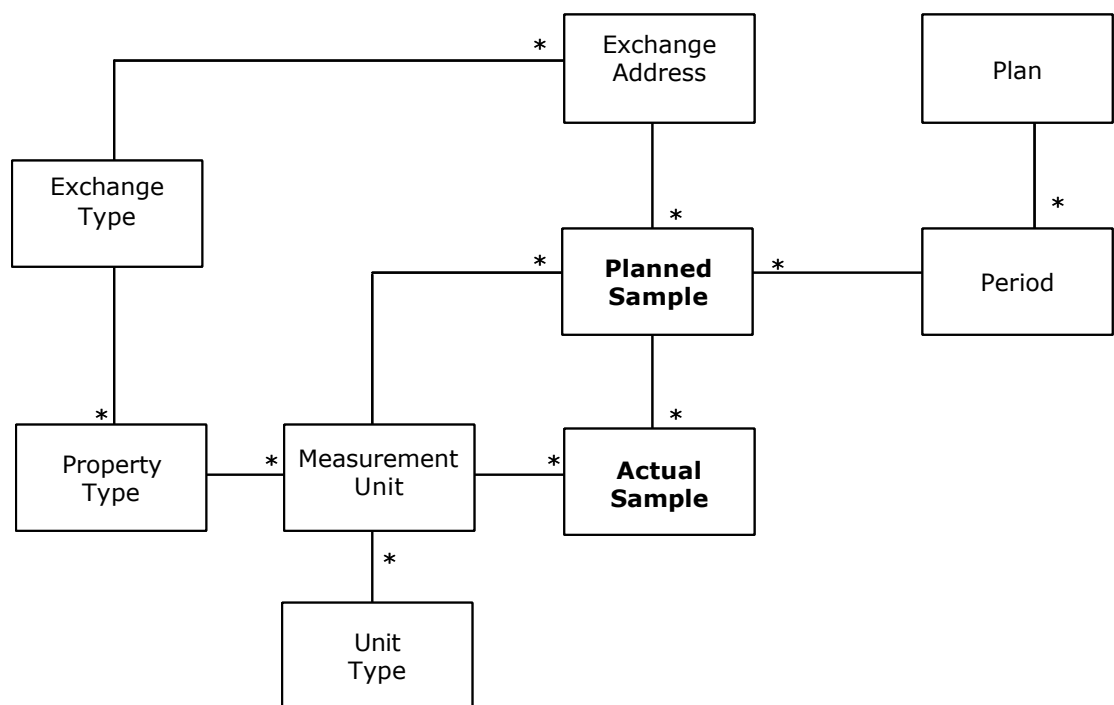


Figure 20: Conceptual sketch of Mackintosh’s alternative model.

According to *Mackintosh*, the original object-oriented architecture was designed with a data bias which resulted in a model centred on an exchange class. Although exchanges anchored the problem domain and were central to the subject matter expert’s descriptions, the system they had been tasked to design was not primarily about exchanges—it was about measurements. Mackintosh commenced a re-architecting phase which significantly changed the object model, the software architecture and the implementation. A major redevelopment phase of six months commenced with an accompanying unanticipated rise in headcount and project cost.

## 7.3.5 Designer's reflections on the design process

*Le Corbusier's* apparent desire to satisfy the most visible of the stakeholders (the subject matter experts) appears to have distracted his attention from the system's main purpose—exchange monitoring. It could be concluded that he lost perspective and instead focussed on the stakeholders and the resulting gratification. Although the project was tasked with exemplifying object-oriented development and object modelling, this objective was always secondary to the primary one of delivering a working exchange-monitoring solution. In his pursuit of perfection, *Le Corbusier* may have seduced himself—'*Le Corbusier* drew a model that didn't have any crossed lines', *Mackintosh* recalls.

*Le Corbusier's* object model was flawed as a physical (or implementation) model. This distinction between the idealised logical view and its more pragmatic physical realisation has been accepted for decades in both database and program design. Equally accepted is the existence of a mapping from logical to physical views. *Le Corbusier* appeared to either ignore or deny the relevance of this distinction to his project. His reasons are not clear. One possibility is that he took the promise of object-oriented databases—to persist business objects transparently—at face value. He may also have been prepared to rely on ObjectStore's capacity to allow control and re-configuration of physical page mapping to 'tune out' any possible performance problems.

*Mackintosh* was critical of *Le Corbusier's* approach, expressing in strong terms that the basis of the model was wrong. 'In effect, we didn't give a Tinker's Cuss about an exchange', *Mackintosh* surmised, 'what we were about was measurements'. He blames the emphasis (inherent in object modelling) to select and model objects that have tangible correspondences in the problem domain. He recognised a system archetype based on measurements as being a better fit. 'The downstream systems were just taking measurements... they could have been measurements from heart monitors—we didn't care'. In his view of the problem, 'measurement' was the key class, and basing the model on a generator such as 'measurement' (with time intervals, sources, and the like) would have yielded a superior model. In his alternative model, the processes did not depend upon physical objects, but on transient abstractions. But the abstraction 'measurement' was intangible when compared to 'exchange'. Further, the concept of measurement did not surface from the discussions with the subject matter experts. 'To the people we were talking to... they've got their spanners out, they're working on the exchanges, so to them it's the key thing that they're dealing with', *Mackintosh* recalled.

*Mackintosh* also raised and contrasted their respective personality biases—'*...Le Corbusier* is

a perfectionist, you can see his personality come through in his approach to architecture—he loves to perfect, [to] polish the model’:

I tend to be far more, fly by the seat of my pants, and that affects the definitions I use... it’s much more of a personality thing... I tend to be far more pragmatic... I’ll do the 80% every time... that makes architecture thinner, design a little thicker. —*Mackintosh*

### 7.3.6 Relationship to Grounded Theory

*Le Corbusier’s* design was impressive in the way it achieved transparency between problem domain (exchanges and their associated equipment) and solution structures (exchange classes and collaborators) (T6.13, T6.39). Because *Le Corbusier* and *Mackintosh* were skilled communicators, *Le Corbusier’s* object model effectively formed a vision—one that happened to be presented in an object notation (T6.4, T6.13). His design clearly derived much of its appeal due to the use of a central generator (exchange) (T6.43) and it was the shortcomings of this generator that ultimately flawed his design.

Some of the appeal of *Le Corbusier’s* design could be attributed to its simple, uncluttered representation of objects in the domain and the way he was able to use object technology and tools to avoid introducing another dimension of solution-driven complexity (T6.41). *Mackintosh* blames *Le Corbusier* for falling to his underlying data-based paradigm bias (T6.36) despite *Le Corbusier* clearly being an object technology protagonist. The case illuminates the subtleties of paradigm or perspective bias—even though *Le Corbusier* was immersed in object modelling, he could not see past abstractions founded primarily on state rather than identity or behaviour.

*Le Corbusier* knew his team’s capabilities to a man and adjusted both his design approach and the level of detail based on team capability (T6.18). He may well have argued that his assessment of team capability justified his heroic, isolationist approach. The architect’s drive to use object-oriented technologies end-to-end implies a degree of resume-building (T6.17) and it is not clear how *Le Corbusier* or *Mackintosh* influenced the client in this choice. When *Mackintosh* says ‘it was why we got the next business... it worked really well’ he hints at another motive (T6.5) for *Le Corbusier’s* desire to make the engagement with the project’s business and engineering stakeholders successful—the prospect of follow-on work. Everything about *Le Corbusier’s* approach seems to have been directed at these stakeholders, down to his choice to iconise exchange in his architecture. Since he did not state them in his interview, we can only guess at his conscious motives.

In adopting the master designer persona, *Le Corbusier* effectively relegated methodology and

process as playing only subservient roles in the project’s design activity (T6.26). It is worth noting that a methodology product was not included in the set of object technologies. *Le Corbusier* and his team were expected to define their own methodology, customised to the selected products, and to mentor the client’s permanent employees in its substance and execution.

It must also be noted that since some of the hallmarks of situated design are absent, this case study yields equal evidence for a rational design process as for a situated one. For example, *Le Corbusier* discovered and sought to objectify (rather than negotiate) the system’s requirements (T6.3). He forward-engineered the model rather than fitting known archetypes (T6.45) or problem types (T6.28). He discovered new abstractions rather than reusing pre-existing abstractions (T6.39). He adopted and staunchly defended an objective perspective by choosing to discover and model tangible or physical objects. The design’s ultimate failure should not be interpreted as condemnation of this approach. The only discernible fault that could be attributed to *Le Corbusier* is his choice of perspective.

## 7.4 Conclusion

Overall, these two case studies highlight the characteristics of healthy and unhealthy collaboration. Good collaboration is typified by reinforcement of the ‘design episode’ (breakdowns, creative versus evaluative parts of the design episode), paradigm and perspective awareness, and the ability for a designer to assemble a personal design process that capitalises on the capabilities of collaborators.

### 7.4.1 ‘Design episode’ structures collaboration

*Breuer’s* story illustrates how designer collaboration should work, and in particular, how the contextual arrangement of forces—both business and technical—can drive interactions between self-selected actors to trigger distinct design episodes. While *Breuer* single-handedly championed his design contribution, he owed its existence to his fellow designers. Both the conceptual design and its realisation are exclusively and unmistakably his, but his isolated act of design resulted from a period of close collaboration. *Breuer* took important contextual drivers and constraints from his fellow designers rather than from other sources (such as theory, or materials).

Also noteworthy is the pivotal role that a primary generator played in *Breuer’s* design act. In *Breuer’s* account, a breakdown event drives the recognition of a single primary generator, from which the Decision Tree design formed. The generator—that business rules are most

naturally modelled as a tree—was seeded by a remark from another designer (*Chen*) amidst the breakdown of his personal design perspective. *Chen's* breakdown event effectively transferred design momentum to *Breuer* who re-conceptualised *Chen's* model from his own perspective. *Breuer's* success can be interpreted as resulting from the alignment of his perspective (hierarchical decomposition) with the requirements.

Above all other possible interpretations, the case illustrates the unmitigated failure of process and method in directing architecture design. Almost no aspect of the Decision Tree's design can be traced to the application of a separate and independent design method or overarching process. If the process used on the project was repeated, or if a different group of designers had been engaged, there is no evidence to suggest that a Decision Tree pattern would have been recognised. The designer's perspective also played a significant part in the design outcome, as the real designers emerged, negotiated their roles between each other, and self-selected underneath the organisational radar of the project's management team. The self-appointed design leaders (*Chen*, *Nygaard*, *Goldberg* and *Breuer*) flexibly changed roles during the design phase under their own control and impetus, and in the story as told these changes correlated strongly with breakdown events experienced by each designer. *Breuer* admitted a propensity to seeing everything as a tree, and in this particular context (perhaps circumstantially) a tree fitted the problem perfectly. It is significant that it took a handful of different designers working from different perspectives to achieve this recognition.

*Breuer* used his own personal notions of elegance—symmetry, simplicity, conformance—with a standard structure as guiding principles in knowing when his design was satisfactory, well-conceived, and complete. He interpreted his solution's elimination of complexity as a sign of correctness and completeness.

#### 7.4.2 Collaboration serves evaluation

The second case study can be summarised in three words—perspective, perfectionism, and personality. All of these are types of designer bias. *Le Corbusier's* perspective was one of an modelling purist, and as a result of his perspective-fixation, he heard the wrong emphases from stakeholders and believed the kudos from his engineers. He allowed his perfectionist personality to rule over pragmatism for a critical period in the design process.

*Mackintosh* leaves little doubt that *Le Corbusier* owned the design, performed the conceptual design almost exclusively in isolation, collaborated for inputs only before embarking on his design act, and transferred responsibility for the architecture only when he left the project.



*Le Corbusier's* approach to designing the architecture fits easily within a conservative planning regime. He worked within a distinct design phase with set tasks for conceptual and routine designing. The design process followed a regular and predictable trajectory that fitted comfortably within these planned task boundaries.

*Le Corbusier's* architecture hinged on a single primary generator—the centrality of the exchange—in both the problem and solution spaces. An evaluation of its appropriateness depends upon the timeframe which the observer considers, because the primary generator could only be considered to have failed when the operational requirements of the system changed. The overall assessment of *Le Corbusier's* design therefore hinges on whether the observer considers that a designer should adopt the responsibility to anticipate new or changed uses of the software system they are charged with designing. Regardless of interpretation, the 'exchange' generator did not lead to an extensible architecture and must be considered a dubious choice.

Both stories confirm the nature of collaboration in situated design. *Breuer's* story exemplifies how the design episode serves to structure collaboration. *Le Corbusier's* story reveals how the absence of collaboration can lead to myopia and perspective bias. An important finding is the demarcation of collaboration and how it affects design action. In both cases, the creative design act was performed by a single designer, in isolation. In *Breuer's* story, the collaborating designers contributed evaluations from alternative perspectives but at no time did any of *Breuer's* designers work together to produce a design. This suggests that the act of design is an individual one and that collaboration serves rational evaluation.

## Chapter 8: Findings

If our ambitions are higher, if we seek not just competence but quality in our designs, then we must recognise that such a difficult and intangible attribute as quality is not just determined by the clear exposition of constraints, by explicit rules, by rational analytic procedures. All such things are vital and necessary, but they do not approach the heart of the matter. Thus a designer develops a sense of a target-object. This is not an abstract goal, but a concrete tangible artefact which will embody all goals, including the subtle ones that the designer is unable to articulate and may have only half-suspected as appropriate to his design task. This sense of a target-object can be accompanied by the feeling that the thing is already in existence, much as a sculptor feels about uncovering the status in the stone. This may be an illusion, but it can be a powerful motivator in the process of design. (Walker and Cross 1976, p. 42)

### 8.1 Introduction

This chapter reconciles the results of the qualitative analysis (Chapter Six) and those of the case studies (Chapter Seven) into a set of succinct, focussed narratives on the expert practice of software design. These findings represent the final refinement and endpoint of the analysis. Collectively, they communicate the findings of the research and constitute an answer the thesis' research aim—to build a rich description (from the data of practitioner's accounts) of how the practicing software architect approaches design.

### 8.2 Findings

The remainder of the chapter synthesises the research's findings in eight distinct narratives. Each may be read in isolation, or collectively, when they form a comprehensive account of software design.

### 8.2.1 Architect in context

The contemporary software architect recognises that the context in which they design has technological, business, political, social and cultural dimensions. Four characterisations emerged—the architect as a change agent, as an arbiter and controller of the software design investment, as a facilitator of the design’s implementation, and as an actor in a cultural context. The architects described ways that each of these roles directly or indirectly affect or constrain their designs and design outcomes.

Firstly, design in any established context implies change, and change may mean upsetting an existing equilibrium of systems. The software architect is a change agent, able to choose from options that have varying implications for the *status quo*. The architect must consider the totality of his solution rather than just its structure and integration in a technology dimension (T6.2). This finding is entirely consistent with design in non-software fabrics, and is substantially similar to Mayall’s (1979) principle of totality. Few of the participants, however, were able to describe how consideration of the systemic (or wider) implications of their work beyond the confines of their teams and projects changed their design’s trajectory (*Gropius* being the notable exception).

Secondly, software architecture represents an economic investment that may in some circumstances require justification, and architects may be required to provide such justification. Architecture is far from a precise science and its benefits and return on investment are difficult to measure, if not subjective. Architects have little experience of making return-on-investment assessments and are therefore unsure of how to make this justification. The participating architects generally agreed that investment in architecture results in a simpler, more compact code base that costs less to bring to production and subsequently maintain. Simplifying design ‘breakthroughs’ (such as *Breuer’s* Decision Tree) facilitate a leap in architectural simplification and code reduction which can achieve this architectural goal. The fact that object technology inherently provides mechanisms (such as encapsulation and frameworks) to grade the skills required across an architecture, thereby de-skilling sections of the software development workforce (T6.7), was recognised by only a handful of the participants.

Thirdly, software architects differ from their built-world counterparts in that they expect (for the most part) to stay engaged beyond the conceptual design phase to realise their designs. That a software architect’s account of designing *and delivering* systems carries more weight professionally than accounts of designing only might explain this difference. The software architect acknowledges a direct and often personal responsibility for his design’s

realisation, and this results in certain design constraints that are unique to the software designer's situation. One such constraint is the architect's perception of the development team's capability. The architect may modify aspects of his design—the basis of decomposition, the mechanisms used, the overall complexity of the solution—to match his perception of the team's ability to realise the architecture (T6.18). For example, architects may use encapsulation or framework structures to isolate complexity and expose simplified abstractions to parts of the development team. Most participants acknowledged varying the amount of elaboration in their designs on the basis of perceived development skill (T6.18). Physically distributed teams and out-sourcing can similarly shape the software architect's choice of structure—for example, an architect may consciously design a software structure to maximise the likelihood of (off-shore) implementation partners delivering components or services.

Fourthly, architects are influenced by the prevailing culture, be it business, organisational or societal. Culture influences the attitudes of stakeholders, particularly attitudes to investment, and therefore may directly constrain what the architect can achieve. A number of the participants reported stories of how short-term business exigencies had eroded medium or long term investment in architecture. If the prevailing culture does not value long term investment, or investment in infrastructure, then the argument for investment in software architecture becomes harder to make (T6.20). It seems reasonable to expect that certain organisations and industries would be more tolerant of infrastructure investment than others, and also that a software product (with a projected commercial viability of, say, ten years) would demand such investment. However, *Eames'* account of working the architecture within a successful software *product* company suggests otherwise. Although *Eames* did not confirm the reasons for management's disinterest in resourcing architectural investment and remediation activity, sales and revenue-raising activities were clearly dominant in his account (T6.17).

### 8.2.2 Architect as professional

Software architects are providers in a services economy. As such, they are subject to a class of forces that can be attributed to what might collectively be called 'professionalism'. Recognising the software architect's engagement with the professional culture brings to light the plurality of the individual's motives—that is, types of design drivers that are primarily socio-cultural and would be unlikely to emerge from the use of research paradigms other than interpretivism or ethnography. All of the findings in this category relate to professionalism as a driver of various types of design bias.

Professional software architects distinguish between knowledge of a particular technology or product and the knowledge of how to apply, use or deploy technologies or products in working solutions. This demarcates the perceived value of hands-on experience, and as a result, can introduce a form of bias into technology and design option decisions (T6.21). *Stickleley* described an ‘infatuation with objects’ and individual’s intent on ‘resume-building’, as does *Le Corbusier’s* drive to use object-oriented technologies end-to-end in the second case study. When *Mackintosh* says ‘it was why we got the next business... it worked really well’ he hints at the same motive more bluntly—the prospect of follow-on work with the same client (T6.5). This may explain *Le Corbusier’s* desire to make the engagement with the project’s influential engineering stakeholders successful. The professional architect must manage his team’s (and his own) desire to inappropriately use a project or engagement as a means to gain exposure to a marketable technology, win further work or otherwise benefit personally in a way that might compromise design quality.

Distinction and professionalisation of the software architect’s role has heightened in recent years. When the progression from developer to architect is viewed as a career step, the wrong individuals can end up filling architecture roles. Software quality can suffer when the status associated with the architect’s role compromises the architect’s motive (T6.21). Inexperience or incompetence compromises design quality and leads to dysfunctional teams.

Professional software architects are sometimes encumbered with incentives via their employer that can influence both the design process and the architecture itself. In some circumstances, architects can inadvertently become players in sales or commercial negotiation (T6.17). An architect may experience pressure to recommend an inappropriate technology or to suppress a preferable design option for commercial rather than technical reasons. *Lethaby* described being under pressure not to propose his preferred architecture (which would have reduced database license revenue for his employer) and *Breuer* described being told to make his employer’s (inappropriate) technology work ‘any way he could’. These situations can present significant ethical and professional dilemmas for which software architects appear generally unprepared.

### 8.2.3 Architect as negotiator

A significant finding is the degree to which the software architect is an active negotiator of problems and solution options during design. Evidence for negotiation emerged in problem definition (T6.3, T6.28), stakeholder management (T6.4), planning (T6.11), and

within the development team (T6.4). The architect negotiates during the design act with regard to abstractions (T6.39), patterns (T6.47) and solution archetypes (T6.46). Negotiation can be interpreted as the establishment of a discourse for the construction of a shared sense-making environment, from which knowledge can be developed, norms established and a shared ontology developed. The architects report adopting different negotiation approaches for each of the distinct roles (stakeholder, sponsor and fellow designer/developer) they negotiate with.

### *Stakeholders*

Although generally aware of the need to inform and engage with project stakeholders outside of their development teams, software architects admit they are not always effective in doing so. The three primary areas of concern are vision, planning and progress. Some architects place significance on the formulation of a solution vision that defines (at a high level) the purpose, intended structure and contextual fit of their proposed software solution (T6.4). Visions are expressed as narratives or abstract models, communicating what the solution will be at the expense of how it will be made. The significance of a vision is that the architect uses it as a medium to clarify, validate and negotiate requirements, expectations and solution options with these external stakeholders. *Le Corbusier's* object model in the second case study effectively formed a vision—one that he chose to represent in an object-oriented modelling notation (T6.4). *Le Corbusier* clearly judged rapport with the telecommunications engineers to be important, and in the first phase of the project, this probably contributed significantly to the project's (and his) perceived success.

Conversely, when a direct connection between architect and business stakeholders cannot be made (such as when the solution is a new product and no user base exists, or in *Stickley/Eames'* case where a sales division imposes itself between engineering and the customer base) the architect may experience frustration. In the first case study, *Chen* expressed frustration (in the second design episode) that he was not afforded an option to negotiate the user's requirements on constraints to fit his predisposed product-catalogue archetype (T6.28, T6.45).

Plans and planning processes are also negotiated between software architects and business stakeholders (T6.11). This is because the architect's input into a sponsor's plan must be provided before the design trajectory of the solution is known, and also because it raises the return on investment question for software architecture discussed above. Budgeting and planning are usually performed early on in a project. The inability for the architect to

anticipate the design's trajectory stems from the number and nature of unknowns at this point, and the unpredictable nature of the creative part of the design process. Software architecture is volatile in the early design phase and subject to re-conceptualisation and significant shifts. A further complicating factor is that business stakeholders do not always understand the nature of software design and expect it to follow a predictable path. This is partly due to the invisibility of conceptual design work and also the radical nature of software design, particularly the propensity for designers to throw away parts of their designs and start again. The conflict that can emerge between designer and planner (or manager) is not often explicitly discussed and may be mismanaged. Once a plan is in place, the software architect may actively manage the visibility of architectural design progress (T6.19) by attempting to communicate tangible progress to stakeholders during periods of architectural design. The architect may also attempt to make selected aspects of the design process transparent.

#### *Problem negotiation*

The second type of negotiation is problem negotiation. It may seem surprising that architects would not take problem statements as concrete but rather attempt to change them, even to alter them to fit a personal preference for a solution archetype, architecture, pattern or structure. However, a number of the architects described doing exactly this. The extremes of problem negotiation range from the negotiation of individual requirements within the architect's design brief to negotiation of the stakeholder's overarching motivations and goals.

Experienced software architects recognise the opportunity to negotiate some of the goals and requirements that the client brings to the engagement (T6.3). The purpose of such negotiation goes beyond clarifying or detailing specific requirements—which would be expected to occur in most if not all engagements—to that of reorienting the client's goals, expectations, and high-level requirements. Such reorientation is possible because clients sometimes form an idea of what the solution should be and then express their perceived needs in inappropriate terms or as an extension of what solutions or technology they already have. The architect can often open up the client to other possibilities and in so doing, re-negotiate both overall goals and specific requirements, often at the same time. Clearly, not all clients are amenable to this kind of engagement and not all goals and requirements are negotiable. Experience helps the architect to discern which requirements may be challenged.

The architect's negotiation of the problem is driven by two motivations—firstly, the need to improve the client's understanding of the problem and secondly, the need to avoid specific requirements that the architect expects to find difficult to deliver. When negotiating, software architects may attempt to re-orient the client's conception of the problem into one for which they have a known architectural solution—in other words, experienced software architects negotiate problems to make them match known problem types (T6.28) and for which they have known solutions. Architects justify this negotiation by claiming that achieving a closer alignment between the problem as stated and a known problem type reduces risk, both personally (or professionally) and for the client. Architects who engage in such negotiation may choose to validate their interpretation of the client's goals under negotiation at any time by using a vision statement (T6.4) written specifically for particular stakeholders.

#### 8.2.4 Architect as collaborator

Internally, within a project or team, the architect is a collaborator who both shares the designing and facilitates other's designing. Such collaboration has a significant affect on design outcomes. Whether an architect collaborates on the actual design or performs the designing in isolation appears to be driven by both personality and circumstance. At one extreme, *Le Corbusier* epitomises the master-designer—the second case study illustrates how his approach leads to rapid progress but also how the super-empowered architect can be blinded to peer input and escape review.

The architect's collaboration is directed towards specific purposes, such as facilitating decision-making, organisational change, or skills transfer (T6.6). Architecture implies making certain decisions in order to selectively solidify (*van der Rohe* used the term 'crystallise') architectural structure. Architects are wary of making early decisions that rule out options (T6.10). They use a range of design techniques that leave extension options open and cater for the unexpected, such as defensive design techniques, extensible architectures, encapsulation and other elements of good object-oriented design. Collaboration with team members serves to socialise, evaluate and build consensus around these decisions, and also to establish and reinforce behavioural norms and decision-making processes amongst the team members. In establishing these shared practices, collaboration over the architecture's design establishes a design and development capability across the team (T6.6). However, both the first case study and the qualitative analysis furnish evidence that the architects predominantly perform the creative part of software design in



isolation. Collaboration serves routine evaluation activity more than creative design activity.

### 8.2.5 Architect's use of methodology

Software architects use methods and processes (methodology) in a highly selective manner. They are generally critical of both method content (what the method instructs its follower to do) and method's purpose in the context of a project (why some sponsors prefer or insist on method use). As a result, most architects treat methods as toolkits of potentially useful tools and techniques but not as controllers or arbiters of design. This may be due to a preference to retain control of the design process. The architect's collective rejection of methodology as a viable primary driver of software architecture (T6.26) may also be interpreted as a form of protectionism.

It may be argued that the architect's reliance on design method is related to project scale and that the participants in this study lack exposure to very large-scale software systems that demand more strict control of the design process (such as aerospace or real-time control systems). However, the preliminary survey results (Appendix D) reveal that 54% of participants have worked with object-oriented systems comprised of between 250 and 1,000 manually designed and coded classes and 67% have spent 2 or more years working with the same object-oriented architecture. If methodology was an important design driver for software architects, it is likely that this group of participants would have identified it.

Software architects are not entirely dismissive of methodology—they admit 'cherry-picking' useful techniques and themes from methods (T6.22), trying and applying these to design problems at hand. Architects assemble their own 'toolbox' of design techniques that they have found useful over many projects, and with time, their knowledge of how to use these tools becomes tacit. This know-how constitutes one of their most important design assets. The term 'appropriation' is an apt description of this fossicking because architects cannot always remember the source of their techniques and may even claim to have invented them. This appropriation also explains how they can practice design of large and complex architectures without explicit reliance on methodology or process.

Architects take a critical view of method use in projects. Architects regard methods as a mechanism for transferring project risk away from project sponsors and stakeholders (T6.23). Where an architect perceives method being used in this way, he may mistrust the method even more and adhere only minimally to it.

Architects also take the techniques and themes they appropriate from methods forward from one design engagement to the next, adapting and evolving them as they are applied in new design situations (T6.24). These are generally a digested and personal (ie. tacit) form of knowledge—the architect may describe them as ‘precepts’ or ‘principles’ rather than methods or techniques. The exact form that this knowledge takes is difficult to generalise, but some indicators that emerged from the research are discussed in 8.2.7 (‘Architect’s memory’).

In the absence of commitment to methods, architects sometimes find it difficult to enforce their vision consistently, especially across large projects or teams. Under such circumstances, enforcement of paradigm in the software development tools being used defines the ‘lowest common denominator’ in architectural consistency (T6.25). This observation strengthens the argument to use tool support to reinforce architectural patterns, principles and structures, particularly for large teams.

#### 8.2.6 Architect as abstractionist

The participants identified the ability to abstract as the most important skill for designing software architecture. The act of design in any medium puts an artefact where none previously existed—it is essentially a creative act. This ability to create is a skill possessed by some but not all people. Creativity is central to software design regardless of the level of abstraction or domain (T6.8, T6.9). Software architects employ their abstraction skills in almost all acts of design, and even apply them when designing non-software artefacts, such as documentation structures, or a personal or team design process (T6.40). The architect’s abstraction skill is therefore one of recognising abstract elements, concepts and structures through detail and noise to selectively form, use or reuse structures, patterns or processes, regardless of the medium.

##### *Purpose*

In the process of software design, abstraction serves the invention of structure, management of complexity and the demarcation of work. Abstraction facilitates the discovery of the underlying structures in the problem and solution spaces. When these structures converge, frequently through the discovery of commonality of structure in the problem and solution spaces, the architecture is simplified. Reducing, collapsing, converging, simplifying structure are all signs that good abstractions are being found and that the design process is progressing (T6.32). In the first case study when *Breuer* found a matching solution structure for the rule-structuring problem, the right abstraction revealed

previously obscured similarities across the architecture and consequently dissolved complexity in a dramatic fashion.

Complexity is another motivation for the architect's reliance on abstraction skills. Architects must design for complexity in the problem domain whilst striving for simplicity. Some architectural complexity ('necessary complexity') is unavoidable. However, architects may inadvertently introduce complexity by choosing unnecessarily complex design options ('introduced complexity') or by missing recurring problem or solution space patterns. Skilled architects consistently minimise introduced complexity as a result of their experience and their knowledge of solution archetypes, patterns and practices. Consequently, an architect's design skill is at least partly revealed by its complexity and partly by how well their design addresses problem complexity (T6.31).

Abstraction serves demarcation of work when architects modify their abstraction process and their abstractions to consider the basis of the architecture's downstream consumer (T6.37). One of the ways that architects consider the consumers of their designs is by selecting an appropriate level of abstraction. The chosen level is not applied universally—rather, some areas of the architecture will be elaborated while others are left more abstract, unfinished, or high-level.

### *Structure*

As a design medium, software dictates no structure but instead provides language-level mechanisms for the designer to choose a structure to suit the requirements of the solution. All structure in software architecture is therefore at the designer's discretion and serves the designer's preferred purposes. Abstraction in software design is concerned with forming and selecting structures, and continuously assessing the quality and usefulness of candidate or existing software structures. In general, problem space structure is mapped onto business objects and relationships within layered software architecture. Solution space structures may be found in mechanisms in the architecture (such as Model-View-Controller) that implement required non-functional system requirements such as synchronisation of views, distribution or scalability. The architect sources, selects and converges his design's structure from candidate problem and solution structures as the design proceeds (T6.13). However, the architects diverge on the importance of problem-solution transparency.

*Abstraction in the design act*

As expected, abstraction features heavily in the architect's accounts of their personal design processes and the act of design. Software architects make no distinction between abstraction processes for designing at the architectural level or the component or intra-component level. This is evidenced by the fact that when the participants were asked about the difference between 'software architecture' and 'software design', only abstraction level separated their definitions (T6.14). That architects report moving between levels of abstraction constantly when designing suggests that the difference between software architecture and software design is immaterial from the perspective of personal process. If there is a difference, software *design* differs from software *architecture* only in that it resolves problems or forces that arise from (and remain unresolved by) the software architecture (T6.15).

Most software architects appear to exhibit a preference for designing at a particular level of abstraction, to which they naturally gravitate when thinking about software architecture. Some architects prefer to spend most of their time at higher levels of abstraction (ie. conceptual or class-level models), subjugating code-level concerns to modelling concerns, while others prefer to think at the level of code and let the conceptual or class-level model 'take care of itself' (*Morris*). The architect's preferred abstraction level broadly follows the two categories that emerged from the analysis of approaches to conceptualisation—top-down and emergent (T6.42). Architects do not universally practice a traditional step-wise refinement or decomposition process to discover abstractions—they may arrive at an initial design by progressing from the abstract to the specific, or from the specific to the abstract, or they may follow a combination of these two trajectories. Some report working both trajectories more or less simultaneously, both on different and on the same parts of the solution space (T6.38).

While it seems commonsense that architects be allowed freedom to roam over the complete architecture until such time as the architectural design is complete, *Breuer's* case study account suggests that this is not always the case. His project evidenced a project-wide process pattern in which an initial high-level package decomposition was done and then each package was passed to a separate development team for intra-package design and development in parallel. This had the effect of limiting interaction between designers in the teams, apparently in the interests of project productivity. This lack of communication across the large project inhibited discovery and propagation of simplifying design patterns (T6.39). *Breuer's* courageous promotion of his Decision Tree pattern

signifies a dysfunctional design culture and should not have been necessary in a project that fostered investment in architecture design effort beyond the initial package-level decomposition.

As would be expected with any form of conceptual modelling, an architect's abstraction skill improves with experience. In practical terms, experience arms the architect with personal patterns and archetypes (8.2.7, 'Architect's memory', T6.44) and also serves to increase the architect's speed of abstraction discovery or selection from alternative candidate abstractions (T6.30, T6.46).

### *Generators*

The concept of generators from design theory was introduced in earlier chapters (6.6.13) in order to determine what its analogue in software design might be. We can distinguish between domains and generators. A domain is a region of the problem space in which constraints, characteristics and requirements can be partitioned. Generators, by contrast, are an element of the problem that has a direct realisation in the solution space that clearly serves to unify, simplify and converge a solution architecture that is otherwise expansive or complex. Examples of generators include any pattern in the software architecture that reproduces or maps a significant, central or recurring behaviour in the problem space. *Breuer's* Decision Tree is an excellent example which owes its power to the fact that it maps the problem structure perfectly and as a result, its solution form significantly simplified the system's architecture. Another example is *Le Corbusier's* 'Exchange'. *Sullivan's* 'AppCentral' and *Lethaby's* 'frisbee' are not convincing generators but rather solution space patterns that the teams discovered and promoted—while undoubtedly useful, they have no particular problem space relevance and therefore do not introduce the degree of simplification of a strong generator. Some of the architects described seeking to anchor their designs on 'primitive' principles or structures (T6.41, T6.43). They describe these as 'architectural principles' and they may source them from the problem domain or from a solution archetype (an amalgam of architectural patterns). These are probably types of generators as well.

To conclude, a generator is a problem space or domain-specific pattern, that incorporates one or more abstractions, that simplifies the software architecture as a result of its use. A generator may provide key concepts to guide the design, or may be a recurring structural pattern that helps the architect to establish the architecture during its conceptualisation. Choice of one or more 'generators' can lead the architect to select particular patterns or

archetypes. Obviously, the choice of a generator is critical to the success of an architecture.

*Problem-solution transparency*

By this definition, software generators rely on problem-solution transparency to achieve their structuring and simplifying power. However, when questioned, the architects were divided on whether transparency in the mapping from problem space structures to software architecture structures should always be a design goal. Some regard this as the *raison d'être* for object-orientation while others think it naïve. The argument for transparency follows from the philosophy of object-orientation as a simulation paradigm in which models are concurrently designed and executed. There is evidence for this interpretation in the second case study—*Le Corbusier's* design achieved transparency between problem domain abstractions and structures (exchanges and their associated equipment) and solution structures (exchange classes and collaborators) (T6.41). Some of the appeal of *Le Corbusier's* design could be attributed to its simple, uncluttered representation of objects in the domain and the way he was able to use object technology to avoid introducing another dimension (object-relational mapping) of solution complexity. *Le Corbusier* used this transparency to engage stakeholders—the degree to which it aided design quality and delivery over the life-cycle is less clear.

The argument against transparency hinges on the claim that ‘real world’ software architecture is necessarily complex and must incorporate structural patterns in the solution space (such as Model-View-Controller and others) for which no problem-space equivalent exists. Most architects agree that problem-solution transparency has value and should be pursued in a domain or business object layer (or package) in the software architecture. Beyond this, it does not appear to represent a universal or even common design goal for software architects outside of simulation systems.

*Paradigm bias, perspective, perspective-shifting*

Software architects, like all designers, are subject to bias. In 8.2.2 (‘Architect as professional’) the sources of bias that result from professionalisation of the software architect’s role were discussed. Another source of design bias is paradigm bias, in which the architect is anchored (at times unreasonably) in a particular paradigm, notably the data paradigm (T6.36). This biases their models and predisposes them to particular perspectives on modelling and design problems. In the first case study, *Breuer* explicitly recognised that *Chen's* (data) paradigm bias blinded him to finding a solution to the

constraints-representation part of the problem because there was no workable solution in a data model. *Chen* had gone as far as his paradigm-bias would allow him to go. Devotion to a single design or decomposition paradigm limits the designer to the boundaries inherent in the paradigm. In the second case study, *Mackintosh* blames *Le Corbusier* for falling to his underlying data paradigm bias (evidenced by his commitment to the centrality of the exchange class) despite *Le Corbusier* being an object technology protagonist. This case illuminates the subtleties of paradigm bias—even though *Le Corbusier* immersed himself in object modelling, he does not appear to have considered abstractions founded on identity or behaviour (rather than state), which might have led him to consider ‘measurement’ as an alternative generator.

Ideally, architects should be capable of disbanding their paradigmatic perspective and adopting another, even if only as a mechanism for validating their preferred paradigmatic stance (T6.35). For example, an architect capable of shifting paradigms might adopt different paradigmatic bases—data, function, object—from which to assess the suitability and viability of a solution architecture or model, with the objective of exposing design options and fully exploring requirements. Although this appears obviously advantageous, little evidence could be found for this behaviour. The first case study presents a particular series of interactions between four designers all addressing the same problem in which each designer adopted a different paradigm or perspective. Despite the obvious design skills and experience of *Nygaard*, *Goldberg*, *Chen* and *Breuer*, none of these individuals demonstrated an ability to perspective-shift whilst designing. However, all were able to see each other’s peculiar modelling perspectives when assessing each other’s candidate designs. The story illustrates perspective and paradigm shifting on the part of the design collective rather than any one individual designer. The ability to consciously shift perspectives when designing, or between design episodes when evaluating, would be a valuable design skill. It is certainly one that could be taught.

### 8.2.7 Architect’s memory

Although this research has not attempted a psychological study of designer cognition, some generalisations can be made about how the participants described their memory and their experience of recall.

#### *Personal patterns*

The participants talked of relying on ‘personal patterns’ when designing (T6.47). While it is true that most architects can recite the names and some detail of a number of ‘Gang of

Four’ patterns, they also regularly use a number of what one of the architects referred to as ‘small-p’ patterns—design heuristics, rules of thumb, and small, self-contained design, code and process structures. These are drawn from the architect’s personal recollections and reconstructions of published patterns, design fragments, idioms, and known solution fragments. They accumulate with the activities of software design—code reading and writing, thinking through solution options, and designing in teams. These personal patterns overlap with and duplicate known architectural and design patterns. Although this suggests inefficiency on the part of the architects, they appear not to be concerned by this rediscovery, and attribute it to the cost of acquiring knowledge and experience.

Architects do not remember the detail of their personal patterns in encyclopaedic fashion. Instead, they remember the value that they believe a given ‘pattern’ contributed in a particular situation (T6.48). This demonstrates a kind of associative memory in which the knowledge fragment is associated with the experience of its use in a past situation. To reinforce their ability to reuse or implement a given pattern, they will call up previous code samples or other examples when evaluating or reifying the pattern. It is only when they ‘call up’ a previous pattern for potential use to solve a presenting problem that detail becomes important, and in most cases, the detail is reconstructed. Thus personal pattern reuse is a situated phenomenon. Also, architects do not appear to invest any effort in generalising, or organising their personal patterns, nor do they attempt to map them into some kind of canonical form. Instead, they are content to let knowledge accumulate in an apparently *ad hoc* fashion, and use the process of recalling past design fragments and solutions as an opportunity to reinstate, clarify or explore them in the context of a reuse opportunity.

### *Archetypes*

In addition to remembering personal patterns and perhaps as a specialisation of these, experienced software architects abstract simplified ‘archetype’ representations from complex software architectures (T6.45). The archetypes they manipulate are an amalgam of structure, metaphor, conceptual machine and the associated heuristic knowledge gained from past experience of implementation. Some architects are able to narrate large and complex system architectures in remarkably simple terms. Their depictions might involve basic message traces and paths, holonic self-similarities, metaphors, or combinations of these characteristics. These narratives are abstract and span or imply many patterns or combinations of patterns. They primarily operate at the system or subsystem level (rather than at the object level) and are able to represent very large and complex systems in highly



abstract terms.

Archetypes are kinds of knowledge schemas or frames and can be thought of as the ‘know-how’ required to design object-oriented frameworks. They generally do not include code artefacts but the architects may retrieve the source code of architecturally significant components from previous systems when elaborating, fitting or working with an archetype. Archetypes are an efficient and minimal form of knowledge template—the architects retain only the key characteristics they need to reify or fit the archetype into the new situation. Consistent with their apparent treatment of personal patterns, they appear to promote the parts of the system into the narrative based on its relevance to them personally. This suggests that they use archetypes to recall and re-tell the design’s *meaning*—what made it successful in their personal experience—not its encyclopaedic detail, which would be well beyond the mind’s capacity to remember or recall with any accuracy. Again, this is consistent with their recall of personal patterns.

In 8.2.3 (‘Architect as negotiator’) we saw how architects sometimes choose to negotiate aspects of the presenting problem so as to look more like a problem that they are familiar with or have solved before. A similar opportunity to negotiate arises after the problem has been agreed when the architect approaches the design of the software architecture. Some architects use their abstraction skills to make presenting problems look like problems for which they have known solutions. This kind of ‘structure-fitting’ is related to negotiation of requirements (T6.3). In essence, the architect abstracts details of the problem (rather than negotiating or varying the problem itself) to achieve a viable fit with a known architectural pattern or system archetype. Structure-fitting via abstraction does not always follow problem negotiation—architects use a known solution or archetype to drive their negotiation of the problem, forming and re-forming negotiating key abstractions as they progress.

Just as architects interpret problems to fit known solutions (T6.28) and to fit known abstractions (T6.39), they engage in a similar form of interpretation at the solution level using archetypes. In the first case study, *Chen’s* ability to contribute the third model variant apparently ‘from memory’ followed from his experience with using it previously on at least several occasions. The situation of the *Nygaard/Goldberg* review meeting provided a stage for him to fit his ‘catalogue-order’ archetype (T6.45) with impressive speed and clarity. *Breuer’s* observation and account—that he had a name for his pattern (the ‘catalogue-order’ model) and believed he had derived from his own modelling experience (T6.48)—reinforces it as an archetype.

*Conceptualisation preferences*

Software architects use conceptual, static, dynamic and historical views interchangeably and at times concurrently (T6.34). Some express a clear preference for one view over another, particularly when doing conceptual design, with static (class) and dynamic (object collaboration) views being dominant. Their stated preference appears to be independent of problem type or domain, and may be primarily motivated by personal preference.

*Ontology*

The process of software design naturally expresses a language of the solution (T6.33) and in object-oriented design, objects (classes) directly define the solution's ontology. The ontology should always include classes but may also include relationships and patterns of collaborating components and objects—all of those software-structural elements that can be named and instantiated. Some software architects build a narrative around their object ontology that enunciates or confirms their vision (T6.4). The emergence and health of such a design language is an indicator of the team's collective understanding. Parts of a solution's ontology are retained as personal patterns that architects take forward from project to project (T6.24, T6.47).

## 8.2.8 The 'design act'

The phrase 'the design act' was introduced in the qualitative analysis to describe the architect's specific action of putting a design where none previously existed. It is intended to distinguish the specific actions which designers recognise as accounting for the creation of a software design. A few of the participants were able to narrate their approach lucidly whilst others offered indirect accounts via stories and recalled experiences. Software architects exhibit a personal preference for how much effort they are prepared to invest in indirect manifestations or representations (ie. models) of a software architecture or component design (T6.16). At the 'explicit' end of the scale, a software designer elaborates a set of architectural models with design-level detail before committing them to code. At the 'emergent' end, a designer goes straight to code without any external models or representations and allows the experience of rapidly iterating the code to evolve the architecture's structure. Emergent designers do acknowledge thinking about the design, sometimes in a semi-formal sense, before and during their coding effort. By not externalising their models and concepts, the emergent designers reduce the possibility of collaborating on a design and demand that their team members adopt a similar emergent approach. The drivers of this personal preference for explicit versus emergent design are

not evident from the research data, but may include the individual's visual orientation or their ability to abstract from flat (textual) representations (such as code) into models or structures. Neither orientation was considered by the architects to be preferable, although some participants who held a strong preference expressed concerns about their opposites.

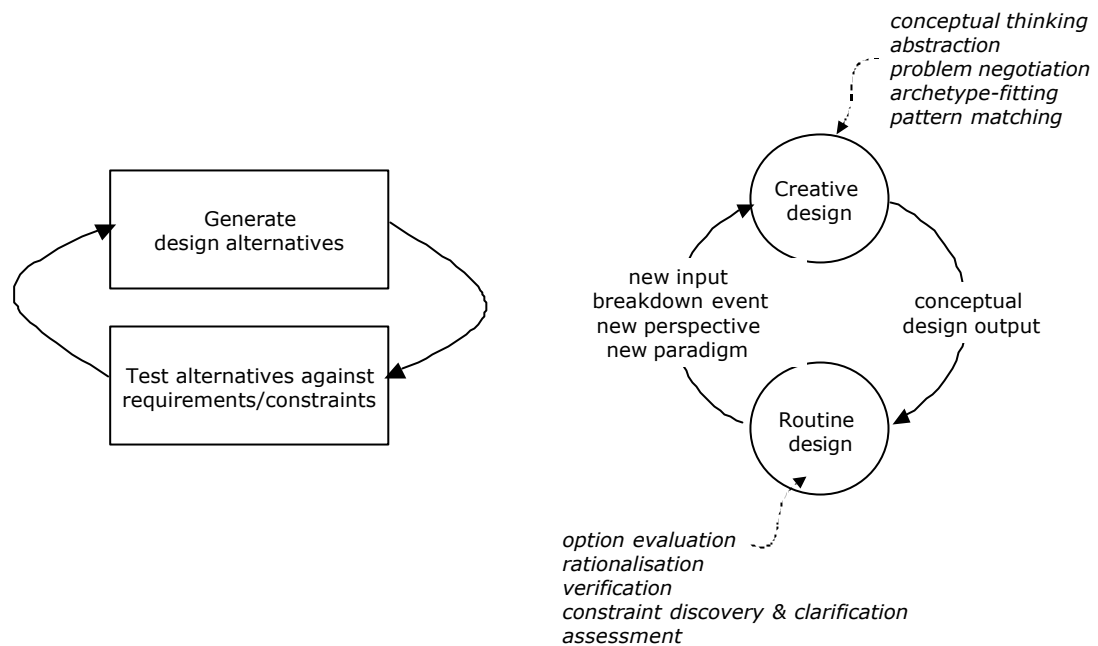
### *Personal design process*

When software architects approach a design task, they generally do not have a common way of starting or a personal process to invoke that they can externalise or account for (T6.27). The architects initially appear to search for a path forward—trialing ideas, sketching and forming concepts, in what may appear to an external observer to be a largely *ad hoc* personal process. As well as trialing early design elements or options, they are also assembling and trialing a personal design process for the design situation they find themselves in.

Software architects alternate between creative and routine design acts (T6.12). Creative (or conceptual) design drives the design act that puts a software artefact where one did not previously exist. The architect's creative mind proposes new concepts and solution structures and makes intuitive leaps into new design conceptions. The architect's rational mind evaluates these creative propositions, verifying the artefact's fitness for purpose and filling in its detail. These two opposing modes of designing operate alternately in a kind of symbiotic relationship.

Architects are critical of methods and processes that attempt to dictate or direct the creative part of this personal design process. To avoid constraining the design act, methods, processes and plans must facilitate both modes of design. They must provide a structure within which creative and conceptual software design can occur by leaving space for creative acts, and also provide a supporting structure in which rational evaluation of what gets created can follow.

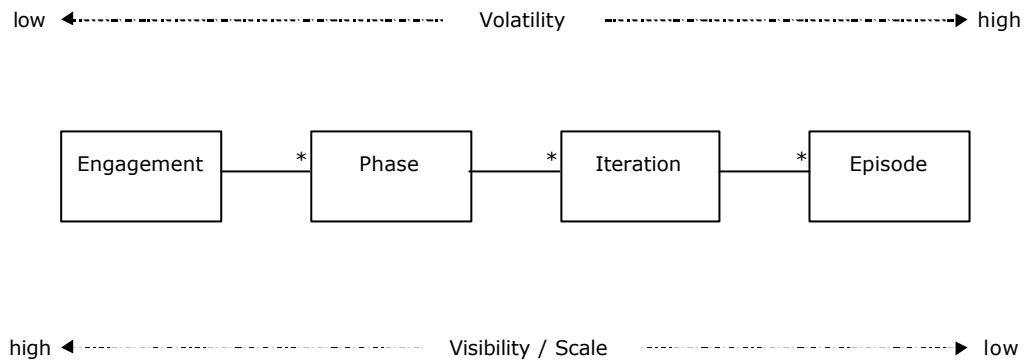
The participants report a relationship between these two modes of design in their experience of designing software architecture. Bursts of creativity or leaps of intuition which put a design or architectural element in place are frequently initiated by an external event. These periods are often followed by one of designing in an analytical mode in which they evaluate and rationalise their design or candidate options to date. Certain events act as a catalyst or trigger for the architect to 'toggle' between these two modes. Figure 21 illustrates a simple model of these phenomena.



**Figure 21: Simon's Generate/Test cycle (Simon 1985) and a model of the participant's reported 'design episode' phenomenon.**

Architects working on the design of large and complex software architecture report iterating this cycle many times during the complete design process and also at many levels of abstraction. Each set of iterations of creative design, followed by rational assessment and routine elaboration that delivers a design (or design outcome) can be thought of as a 'design episode'. A complete design engagement would normally involve at least several and possibly many distinct design episodes (Figure 22).

In the first case study the architect's alternation between creative proposing and rational assessing (T6.9) can be seen in how they changed roles. In each design episode the creative designing was done by the model proposer whilst the other architects adopted the role of rational assessors. What is so illuminating in *Breuer's* story is how each episode delivered such a markedly different model variant and corresponded precisely with an exchange of roles, all without any externally imposed plan, management, method, process or facilitation.



**Figure 22: Relationship between design engagement, phase, iteration, and episode.**

Design theory suggests that better designs can be delivered if the designer generates and then evaluates a number of candidate or alternative designs. However, software architects report investing ‘think-time’ but generally little design effort in evaluating options (T6.30). Experience significantly expedites the generation, evaluation and selection of options, and provides higher confidence in the chosen option. Generally, each distinct design episode in a design engagement serves to refine an architecture under development but may in fact generate replacements or alternatives, which the architect must assess.

Design episodes may be consciously constructed by an architect in order to initiate design progress. One of the primary motivations of software architecture is to manage complexity, and software architects may choose to manipulate constraints as a mechanism to temporarily simplify the presenting problem. Some of the architects report alternately relaxing and tightening selected constraints in order to separate design iterations, and to simplify the number of constraints in any given design iteration (T6.29). For example, an architect may ‘selectively ignore’ non-functional constraints or requirements (such as performance, security or scalability) then consider the effect of these constraints in subsequent design episodes. In the first case study, each episode illustrates how the architects relaxed different constraints of their choice in order to achieve a model. The initial (project team member’s) model ignored too many constraints to be viable but constituted a model nonetheless. The *Nygaard/Goldberg* model ignored or over-simplified the distinction between allocated and available product parts, and *Chen’s* model ignored rules between product parts. In all three cases, relaxing a constraint allowed creation of a new candidate model but the subsequent attempt to tighten the relaxed constraint resulted in a breakdown, which triggered the ‘passing of the baton’ to the next designer. This kind of constraint relaxation is one type of breakdown event in the design process.

*Role of breakdowns*

A breakdown is a kind of occurrence or event that marks the designer's discovery of some inadequacy or flaw in the design as it stands (T6.51). This forces the designer to re-think a particular aspect of the design. A breakdown event typically marks the boundary between design modes—from creative to evaluative or vice-versa. As a result, a breakdown can initiate a shift in perspective or paradigm in an architect's personal design process (T6.50). The architects describe both 'hard' and 'soft' software design failures as kinds of breakdown events (T6.49). A 'hard' failure must be addressed by conscious re-design or refactoring. A 'soft' failure is typically less obvious and may go undetected for some time, but when detected, has the same result. Architects describe using a kind of 'aesthetic sense' to detect and assess 'soft' failures.

*Tension and release cycle*

There is a fundamental cyclical pattern within the personal design process that is formed by the tension that a designer feels about an unresolved problem, and the sense of release that resolution of a design force or problem brings (T6.52). The Decision Tree story illustrates the tension and release cycle both in *Breuer's* personal account of how his design came about and in each of the four design episodes. The tension experienced by the designers is evidenced by their sense of forces unresolved in each of the first three candidate models. The resolution brought by *Breuer's* model was acknowledged by all of the designers when they declared comments such as 'the sun came out'.

*Role of Aesthetics*

Software architects are guided by a personal aesthetic sense. This sense of software aesthetic is difficult to define but is correlated with symmetry (T6.53), closeness of fit to a desired or trusted reference structure, closeness of fit to obvious or self-evident problem space structures, minimalism, consistency and self-similarity. Architects use their aesthetic sense to determine when a design is satisfactory, and hence how to choose between design options and also when to terminate a design episode. In the first case study, an aesthetic sense appears to have been a significant factor in the assessment of each option. Some of the appeal of *Breuer's* Decision Tree pattern could be attributed to the symmetry. *Le Corbusier's* model reportedly had a quality derived from simplicity, clarity and transparency with the problem domain.

### *Termination*

The software architect's design process often terminates as a result of a perceived convergence, and the architect's sense of diminishing returns. Convergence appears to be driven by the same phenomena that define the aesthetic sense, particularly elimination of redundancy, symmetry and closeness of fit to an idealised architectural structure, generator or archetype. The architects confirm that a maturing object-oriented architecture reduces in scale through the discovery and elimination of duplication as well as the discovery of better-fitting structures. Thus architectural reduction frequently signifies design progress (T6.32). Elimination of complexity often follows from the discovery of more naturally fitting structural patterns. While *Chen's* model reduced the introduced complexity inherent in the earlier models, *Breuer's* model addressed the 'constraints problem' and, most significantly, collapsed complexity across the entire project. For all of these termination drivers, the architect employs mostly subjective assessment of the design's overall quality, often to informal or even personal criteria, and often primarily motivated by the designer's personal sense of software design aesthetic.

The architects are aware that it is common for sponsors and stakeholders to associate investment in effort (time and therefore cost) with progress; that volume of documentation is additionally roughly associated with progress, but that quantity of design documentation does not imply completeness or quality of design. They report that projects sometimes reach a point of investment in artefacts—both documents and code—that makes rework or re-architecting difficult or impossible, even though flaws in the architecture become evident. Sunk cost tends to increase architectural rigidity or 'crystallisation' (T6.44). Crystallisation often constitutes an unsatisfactory kind of termination.

## **8.3 Conclusion**

This chapter constitutes the synthesis of the research findings, and directly responds to the research aim—to *explain how experienced software architects and designers understand, reflect on, and describe the ways that they draw upon rational methods, past experience, contextual factors and other inputs to design enduring object or component-based software architectures in industry and business contexts*. These findings complete the 'field work' and analysis part of the research, leaving only the theoretical question posed by the hypothesis—that *software design can only be meaningfully understood when viewed as situated action*—to be addressed in the remaining chapters. Consistent with the outputs of qualitative research, these findings are expressed as rich narratives that lend themselves to interpretation by the reader to other software

design situations and contexts. As such, their value goes beyond being an account of the designers encountered in the process of executing this research to be a theoretical basis for future observations and comparisons.

The next chapter (Discussion) addresses what these findings mean for a holistic depiction of the practice of software architectural design, using comparisons of the findings with existing theory and knowledge as input to the discussion. The final chapter (Conclusion) takes a position on the hypothesis.



## Chapter 9: Discussion

Attentiveness to context, not to self-expression, is the skill we have to foster, to encourage, to share. In natural evolution inattentiveness is death. So is inability to adapt to what we see happening. The *context*, not the boss, has to become the manager of what is done, and how. The bosses' role becomes that of designing the meta-process, designing the situation so that designing collaboratively is possible, so that the interaction (of what everyone is noticing with what everyone is doing) flows. (Jones 1988, p. 225)

### 9.1 Introduction

This penultimate chapter responds to the themes introduced in the background chapters (Chapters Two to Four) in the light of the findings, leaving the conclusions on the hypothesis to the final chapter. The discussion is structured using the four architect personas (rational, pragmatic, critical and radical) introduced in Chapter Two. Questions posed in the background chapters are answered to the degree that the study's findings allow, and within the persona that best fits the particular question or phenomenon.

### 9.2 Rational themes

The rational persona in 2.5.1 ("The Rational Software Architect") characterises the software architect as a highly trained and highly valued technician, skilled in software implementation technologies, able to follow methods accurately, and always conscious of the economic relationship between time and effort. The rational architect-technician primarily adopts an objective orientation to design, treating requirements as absolutes revealed by a discovery process the refines through successive application, employing (rather than constructing or assembling) design methods and processes, and using plans in a predictive sense. Personally, the rational software architect subjugates tensions in aspects of their personal practice of design to the authority of theory and its expression in method.

The rational designer may acknowledge limits on what can be achieved in practice, subscribing not to a ‘hard’ rationality but to a form of bounded rationality (Simon 1983). Even so, bounded rationality does not abandon rationalism as its underlying philosophy. The boundedly rational designer does not acknowledge relativism, criticalism or radicalism as alternate philosophical positions of value, and as a consequence, remains locked into a paradigm that cannot explain pragmatic, critical or radical phenomena that confound rationalistic design in all its forms.

Recognising that rationalism can take many forms (Meredith 2002) it is worth summarising what has been assumed about the rationalistic approach to software design. Rationalism in software design exhibits the following characteristics—it asserts an objective reality in which every requirement or constraint can be resolved to absolute terms; that problems yield (through correct and repeatable analysis) convergent solution structures; that all systems are ultimately amenable to closed systems principles through analysis and decomposition; that design is a process by which problems are decomposed into components which individually yield to mechanisation; and that the design process is ultimately value-free, deterministic and repeatable in the hands of a skilled executor. This thesis and the points selected for discussion in this chapter focus on where the participant’s accounts highlight observations and practices that are contradictory to this portrayal of rational software design.

### 9.2.1 Rationalism and subjectivity

Some findings challenge a rational approach’s objective basis, most notably how the architect deals with the inherent unpredictability of the design trajectory, the relationship between effort and design outcome, and the meaning of negotiation. The notion of ‘return on investment’ follows from the rational assumption that design progress and completeness is proportional to effort. Most architects have little experience of making return on investment assessments of software architecture and are generally unsure of how to make such justifications. Although the participants left little doubt that under-investment in architecture can be fatal to a design-based system project, they confirmed that additional investment does not always raise architectural quality, or that return is necessarily proportional to investment. Instead, the participants described the relationship between design investment and architectural quality as a complex one, influenced by project phase, the capabilities and orientation of the designers, and many other contextual factors. Return on investment implies a mechanistic design process in which outputs are a function of inputs. The architects confirm this to be an over-simplification that is frequently projected

onto design activity from other, more routine forms of design and development. This confirms the related finding that separation of conceptual from detailed or routine design is artificial. Alternate philosophical perspectives lead to polarised interpretations of the return-on-investment problem. From a rational view, when architects blame prevailing business, organisational or societal culture for a poor design outcome, they expose their own inability to engage sponsors on matters of design and delivery. From a pragmatic view, the prevailing belief that all aspects of system development—including design—should obey rationalistic principles is entirely to blame (Truex et al. 2000). The application of a rational process to an open systems (Checkland 1981) (or ‘wicked’) problem (Rittel and Weber 1984) inevitably results in force-fitting objective structures and measures to problem characteristics that are inherently not objective or do not behave rationally. The participants expressed the resulting tension in many ways. As a general observation, the more experienced architects have developed ways of manipulating problems and situations to alleviate subjectivity and it is these approaches that most naturally fit a situated (or *a-rational*) model.

### 9.2.2 Rationalism and negotiation

That architects negotiate almost every piece of the presenting problem and solution challenges the rational persona, particularly the finding that some architects engage in these negotiations in order to re-shape the client’s perception of the problem to be a closer fit to a known solution. In rational software engineering terms, this looks a lot like the tail wagging the dog. Clearly, this claim represents a generalisation because some requirements—the need for end-of-day balancing in a financial trading system, for example—are and always will be absolute. Experienced architects, however, do engage with the client in the definition of a shared understanding of business goals so concrete requirements can be negotiated. A number of participants described engaging in such goal-setting negotiations to influence the client away from ‘difficult’ requirements and instead toward ‘achievable’ ones. This phenomenon of requirements negotiation challenges the rational view which holds that an objective understanding of the problem is discovered through successive application of rational analysis techniques. It also refutes the modernist principle that design is (or should be) value-free (Thackara 1988). This is another phenomenon with polarised interpretations. In the rational view, problem negotiation might be interpreted as the negotiator’s imprecise and even clumsy attempts at uncovering objective problem attributes. From a pragmatic view, problem negotiation is precisely a situated design activity and evidences Floyd’s (1992b) and Winograd and Flores’ (1986)

hermeneutic approaches, because the results of the negotiation are primarily dependent on the actors and situation. The further recognition that situated actors engaged in negotiation carry agendas locates the phenomenon in the critical persona. The accounts of *Mackintosh*, *Breuer* and others provide vivid examples of how commercial and business forces can drive such negotiations.

To conclude that problem understanding and requirements are *both* objective (absolute) and negotiable (subjective) is uncontentious but also unsatisfying. This position leaves the question of demarcation open—when is an objective (absolute) requirement not negotiable (subjective), and vice-versa? Another possible conclusion is that problem understanding and requirements follow a maturity-based lifecycle model, from unsubstantiated and subjective representations via rational analysis to objective assertions. However, this model overlooks the power of the negotiation phenomena as described by the participants to open up, renegotiate and undo requirements previously accepted or thought by designers and clients alike to be absolute. Because the participants describe negotiation as being able to define as well as redefine, the phenomena’s rational basis is eroded. What can be concluded from the findings is that architects negotiate continuously in an attempt to establish social contracts, that these social contracts build to a shared construction from which the designer may choose various means of representation (including formal representations), and that any contract negotiation will likely influence others.

### 9.2.3 Rationalism and method

If the architect’s judgement is to be relied upon to demarcate between the need for objectivity versus subjectivity more or less continuously in a design engagement, then method must necessarily incorporate both interpretation and judgement. By characterising the software architect primarily as a method-follower rather than a method-interpreter or method-author, the rational persona amplifies this conflict between a method’s instruction and its interpretation:

*How do software designers resolve tensions between the selection and application of methods and the use of pragmatic know-how, particularly in cases where these conflict?*

The rational view asserts that methodology is a primary driver of design activity (or what has been referred to as the design’s trajectory). The participants report otherwise. They describe a mistrust of methodology *as a driver of design*, preferring to retain personal control of the design process. There are a number of possible explanations for this finding. One is

a fear of redundancy—professionals of all kinds vehemently resist their skill being commodified or encapsulated in a process. This fear is unfounded both historically and theoretically, as earlier attempts to ‘automate the architect’ (Alexander 1964; Cross 1977) are now widely regarded as having failed (Alexander 1988).

In reviewing the participant’s accounts of failed attempts to use methodology or prescribed process to deliver high quality design outcomes, it is difficult to distinguish between a ‘broken’ rational process and a rational process badly followed. *Breuer* and *Stickle/Eames* described projects in which the architecture was considered to have been complete with the delivery of a UML package model. The architectures of both systems suffered from the premature cessation of design effort. Both stories illustrate a poorly understood or applied process rather than a necessarily broken one—there can be no suggestion that these failures were unambiguously due to the design process being followed. A number of other stories from the participants described badly followed (or inappropriately selected) industry-accepted methods. In most cases, the error appears to have been in the mismatch between process and problem. Thus the participant’s rejection of methodology should be distinguished as rejection of method as primary driver of their personal design activity. An implication of this finding is that architects are accepting of methods that allow them to perform interpretation and to retain control of the design process (although there was no data to explicitly promote this implication to a finding).

Rationalism emphasises logical method execution over method interpretation. Ultimately, practitioners make their assessment of purpose and value of method from experiences such as these:

*What purposes do methods serve in the hands of experienced software architects?*

The architects generally agree that methods provide tools and techniques for rational assessment and evaluation of both design outputs and the design process. The fundamental problem for methods and methodologists is that designers need instruction but do not wish to release control of the personal design process, which they regard as fundamentally creative and not amenable to any form of method-based expression. A method can provide an overarching design and decision-making framework, can instruct on evaluation of design artefacts after the fact, but cannot drive creative design action, which is an act of situated human cognition. Design evaluation is rational but the ‘design act’ is not. *Utzon* verbalised this most clearly:

There needs to be a balance between what can be rationally expressed, the sort of

rational parts of the method, and what is really the other kinds of knowledge that people have, the intuitive knowledge, the experiential knowledge, the creative knowledge, that we really can't talk about very easily other than to talk about it in the most mystical terms. —*Utzon*

The participant's rejection of methodology is qualified. *Gropius*' declaration that he is a 'big supporter of tools... not much of a supporter of methodology' encapsulates their rejection of methodology as a controller of the design process but their acceptance of it as a source of usable, ready-to-hand tools and design resources. The architect's relationship with tools suggests a vernacular, workshop-based model of design in which tools and a specific design process are separated—that is, tools do not embody or dictate a particular design process but instead facilitate an unfolding design process in the hands of the designer-craftsperson. The design act belongs exclusively to the architect, but is mediated by his tool selection and use.

The participant's accounts position them at times in 'the empirical exchange' (Walker and Cross 1976) (Fig 71, p. 58) (in which software designer is also software maker) and at other times in 'direct patronage' (Fig 70, p. 57) (in which software designer is separated from the software maker by an intermediate design artefact such as a model). None of the participants described design in terms of Walker and Cross' 'mature' design process (the 'rational design network', Fig 70, in which designer is permanently separated from patron via a brief and from constructors via models). This asserts evidence for a vernacular model of software-architectural design.

*Kahn* implies no formal methodological reliance when he declares that 'I've got some good people... those good people are just going to have to take control of other people, which in some sense is, I guess, a methodology that you're relying on'. The participant's self-reported propensity to appropriate method fragments further qualifies their collective rejection of methodology. *Kahn*'s 'good people' undoubtedly draw upon an amalgam of commonly accepted method fragments and techniques. *Gropius* and *Kahn*, like many of their counterparts, write their own situated 'methodology' for the task at hand, collecting fragments into their bower and assembling these in a loose and informal process suited to the demands of client, problem and team. The process is not systematic but is equally not random or haphazard, although it may appear as such to an observer. Typically, published design methods provide not more than an initial basis for the construction of this *in situ* method. Rationalism can attempt to make method out of any enacted or observed design scenario. But 'method engineering' and meta-methodology (which allow a designer to engineer a method before applying it in a design scenario) do not substitute for the

participant's reported process-assembly phenomenon, which is a uniquely situated activity.

The architects describe collaborating on design to achieve a shared vision. Establishment of this 'design momentum' or collective understanding may be explained from the rational perspective as the responsible execution of an open and accountable design process—one that is successively revealed and understood by the architect and team as the designing is performed. Conversely, it can be viewed as pragmatic and situated team behaviour, evidencing Floyd's (1992b) 'web of decisions' through the construction of a reality (a problem context) or equally as recognition of Jackson's (1995) phenomenological problem domains. Both of these constructivist views serve the team's purpose of progressing the design. These different interpretations can again be separated by understanding the role of method (or method encapsulated in design and team process) as driver of the design's trajectory. In the rational view, the architect's relationship with method is as executer. In the pragmatic view, the architect is a method-author and method-assembler, such that the method is subservient to the architect's needs at all times, and is frequently defined on an as-needed basis.

The architect's rejection of method as design driver must be scrutinised on one point. Because the participants will have internalised methodology to some degree, their reliance on it can never be known accurately. For example, *Gropius*' claim that methods are less useful than tools may reflect sub-conscious or tacit knowledge of method—in other words, we cannot know to what degree *Gropius* has internalised a working knowledge of methodology such that he regards it as superfluous, even though he tacitly relies upon it. An inherent limitation of interpretive research is that it cannot tease such accounts apart in order to remove ambiguity in such data. The strength of interpretivism is its ability to generalise about reported phenomena and construct a view via such interpreted accounts. That software architects discount the usefulness of method because they long ago internalised it is less important a finding than that which explains the manifestations of this internalised knowledge. The question of how expertise in a particular domain is based on formal knowledge made tacit is a different research question.

#### 9.2.4 Plans as predictors of the design trajectory

We have seen how Suchman's (1987) model of situated action and other models of emergent design theoretically challenge the rational conception of plans as directors of activity:

*How do experienced software designers regard the use of architectural*

*projections or plans?*

And also:

*Do software architects believe that these plans should drive or shape their designs? In the cases where plans and final designs diverge, how do architects explain or rationalise these inconsistencies?*

In the context of these questions, a ‘plan’ is a project plan, design schema, or any *a priori* construct that purports to prescribe the design’s trajectory and how much design effort will be required at any point. These questions probe whether there is evidence that the plan (abstract representation) or the situation is the primary initiator of activity. The analysis yielded two broad categories of designers—‘explicit’ designers who invest in indirect manifestations or representations of the evolving design (i.e. plans and models) and ‘emergent’ designers who are content to allow the experience of rapidly iterating the code base to evolve architecture over time. These extremes fit the rational and pragmatic personas respectively. Explicit or top-down architectural design is rational in as much as it is a conservative approach based on successive refinement of models. Rationalists would argue that the degree of investment in the models is a predictor of how trouble-free subsequent detailed design and implementation will be. Thus the plans and models are artefacts of a (notionally) rational process of design and decomposition—a process which could be observed and mapped (as methodologists do), one which could ultimately stand as viable and independent of both problem instance and situation.

On the other hand, emergent design is a result of a situated process of discovery in which the designer begins with certain preconceptions and impressions, reifies these in code, and then intensely redesigns the resulting artefact via stakeholder interaction with the nascent system. Emergent designers such as *Morris* claim to rely upon an aesthetic sense to guide their decision-making when working in this fashion. This approach to design confounds conservative planning approaches, and as a consequence, proponents of Agile methods adopt planning models based on time-boxing and continuous iteration. At issue here are three main points—evidence for the participant’s reliance on plans ahead of (or at the commencement of) the design engagement, on architectural models as intermediate representations of a design before it is committed to code, and on how divergence from these plans and models is dealt with as the design progresses.

Firstly, the participants report that their reliance on plans at the commencement of a design engagement to guide their design effort is minimal. They report that, in general, planning



for architecture and design effort is not done well and that those plans that are created are often not particularly useful. *Breuer*, *Stickleley* and *Eames* all relayed stories of poorly planned system development projects which compromised architectural quality and design outcomes, in several cases permanently. These and other participants complained of not being able to influence planning timeframes and decisions which then directly impacted their design effort. Generally, this situation did not improve as the project progressed for a variety of reasons. *Stickleley*, *Eames* and *van der Rohe* all described a related phenomenon—large investments in planning and specification typically deliver substantial documents and models that collectively embody an irreversible momentum. This ‘sunk cost’ eliminates the architect’s ability to change the design’s trajectory, almost regardless of the obviousness of need. *Morris* evidences a commitment to feature-driven or agile planning, in which the relationship between designer and sponsor is grounded in an agreement to negotiate very short-term plans as the design engagement progresses. This planning model would appear to be workable in some circumstances but not universally. Thus planning continues to be an intractable problem for the architects—*a priori* planning works only at the highest of levels, at which its usefulness in guiding the design trajectory is dubious. But emergent planning does not always allow a workable or viable design engagement to be struck. When market forces drive architects to agree to time-boxed (that is, ‘fixed price’) *a priori*–planned design engagements, they risk finding themselves in situations like those recounted by *Stickleley* and *Eames*. In general, the participants agree that *a priori* plans do not lead to quality architectural design, but they diverge on the question of what to do about it. Their mistrust of plans forms a self-reinforcing feedback loop—an inability to plan design effort reduces confidence in their effort estimation, hence the return-on-investment problem discussed earlier in the chapter.

Secondly, reliance on architectural models as intermediate representations of a design before it is committed to code is favoured by some architects more than others. Almost all of the participants described visualising models, abstractions, mechanisms and archetypes in their mind before committing these to some form of representation, be it code or model. Where the architects differed was whether they prefer to move from this mind’s-eye conceptual machine or schema to an explicit model or direct to code. *Le Corbusier*, *Howard*, *Breuer*, *Gropius*, *Cook*, *Piano*, *Mackintosh*, *Moore*, *Ruskin* and *Griffin* all preferred to express design in intermediate models, citing management of complexity and the speed with which they can manipulate a symbolic design model as their reasons. *Morris*, *van der Rohe*, *Johnson* and *Sullivan* all expressed wariness of any significant investment in code-independent models—*van der Rohe*’s account of ‘crystallisation’ summarises their primary concern. *Kahn*

and *Utzon*, amongst others, appear to sit between these two opposing positions. The analysis did not reveal the basis for this diversity, although solution domain, complexity and scale are obvious candidates. Revealingly, *Morris* admits at one point to performing a certain amount of conceptualisation ‘in his head’, which begs the question whether the emergent designers are really highly skilled top-down designers with good memories. What is clear is that architects use indirect models to represent their abstraction and early conceptual models when designing rather than as a planning mechanism. The result is that even when the opportunity exists to construct models during periods of planning or conceptual design, the architects believe that they do not significantly influence the plan or address planning concerns.

Thirdly, how the architects deal with divergence from these plans as the design progresses reflects on their choice to use them in the first place. They report that divergence is often handled poorly, and that they are often not well positioned or equipped to deal with divergence. Divergence is typically a point where additional investment in architecture is needed but this is not usually what happens. Because divergence is frequently interpreted as project instability, it is often deliberately ignored or ‘battened down’ by stakeholders, resulting in the unsatisfactory situation *Eames* described. In general, being subordinate to other roles (such as project managers and enterprise architects), the architects reported a low level of influence where planning, direction and resourcing were concerned. This may explain why the participants are not committed to plans or planning, particularly in the large-scale (or town planning) sense, and why their relationship with plans and planners is fraught. *Breuer’s* influence over his project’s direction (in the first case study) is the exception, and the substantial personal commitment he found necessary to effect this re-direction reinforces the point.

#### 9.2.5 Rationalism versus emergence—the design episode

Rationalism does not exclude the possibility of new discoveries, change or divergence as a design process progresses—rather, it posits that rational designers must achieve a balance of planned and emergent structure:

*How is a balance between forward-engineered and emergent design and architecture reached?*

In general, the participants were not confident of their ability to achieve this balance. The finding that designers report distinct episodes of alternating between creative and rational design activity appears to fit both rational and pragmatic personas respectively. However,

the two modes of action are strongly polarised. The participants consistently criticised rational methods and processes for their failure to guide ‘the design act’—the point at which a designer puts a design where none previously existed. The ‘design episode’ phenomenon accounts for the way a design matures in the designer’s mind—first hesitantly, then with increasing conviction, and as a result of a number of (sometimes small) creative leaps interspersed with routine or rational assessment. ‘See, the rational cannot create anything’, *Utzon* claims, ‘...it can only criticise and assess... it cannot actually put forward anything new’. *Utzon’s* insightful analysis—that methods exist to provide a predictable, evaluative framework within which the skilled designer works in isolated bursts of creative output—succinctly summarises the participant’s limited expression of this phenomenon:

How do you identify classes in a business model? That’s not a rational process... the only thing you can do is to say, if you hold in mind the purpose of the system, the human brain being what it is, suggests creatively a set of classes if you work your way through it... then you have to apply some sort of goodness to that which is the rational act which is the refining and the testing... and those two things have to go in balance.  
—*Utzon*

The explanation that all software designers operate in both explicit (rational) and emergent (pragmatic) design modes at times—and that they adopt one or the other mode as a vehicle to describe their actions as it suits them—is a reasonable conclusion. Software design is rational in as much as the evaluation of creatively-proposed structures proceeds according to objective criteria. This part of the design process is repeatable, is not strongly situated, and does not depend on who the actors are or where the evaluative action occurs. The rational persona cannot, however, account for the creative part of software design, which is in most ways the antithesis of rationalism.

*Breuer* suggests one possible middle ground for the architect in this explicit-emergent continuum. ‘I see so many things that are decided at far too high a level’, he observes. Rather than dwelling on the tension between creative design transformations and the alternate structured process of assessment, he suggests that the architect should ideally concentrate mostly on the former and delegate the latter. Using da Vinci’s Renaissance art as metaphorical inspiration, he suggests that the architect should ‘do sketches... sketch in the outlines—and leave it to subordinates to fill in the details’. *Breuer’s* software architecture sketches should be ‘5 page documents without all the trash... the outlines, the essential concepts’ from which a developer would go away, complete the structural design and layout detail, then implement. ‘Methodologies work against this’, he claims, ‘they tend

to say, OK you have to name everything, you have to do this, that and the other’. *Breuer’s* separation allows the architect to drive the predominantly creative part of the design process but relies on immediate feedback from the detailed designers when breakdowns occur. While this model addresses the problem of a methodology forcing resolution down to minute levels prematurely, it conflicts with the finding that architects prefer to work at all levels of abstraction simultaneously, and as a result, would be unacceptable to emergent designers such as *Morris*. In summary, most participants revealed a definite preference for designing between the explicit and emergent extremes, and gave little evidence of conscious movement from this preferred position.

#### 9.2.6 Does rationalism have a role in explaining software design practice?

The overarching question concerns usefulness of the rational persona as a means to describe software design:

*What aspects of software design does the rational persona usefully describe?*

This discussion of the rational perspective can be concluded with three points—routine design (assessment and evaluation) is rational but creative design is not; design effort and design outcomes are not proportionally related; and the design trajectory is not amenable to conventional planning regimes. Rationalism is therefore an essential part of the design engagement but is not its universal basis. A rational perspective cannot account for every design outcome, nor can every design outcome be explained in terms of its degree of adherence to a rational design process. Rationalism frequently attempts to explain shortcomings as inadequacies on the part of the actors—their inability to correctly follow rational processes of discovery and execution, for example. Rational design in software, as in other mediums, risks isolation and disconnection. In the second case study, the pseudonym for participant *Le Corbusier* was deliberately chosen—the 19<sup>th</sup> century French modernist architect’s ‘*unité jardin verticale*’ and the 20<sup>th</sup> century software architect’s Exchange model have certain similarities. Both are characterised by ego-centricity and ultimately, sterility. In line with the history of modernism in the twentieth century, rationalism turns out to be an over-simplification of design in software fabric as in other media.

### 9.3 Pragmatic themes

Under Coyne’s *pragmatic* theme, meaning and intent are neither pre-existent nor absolute, and designing cannot be done without adopting the interpretive norms of one’s community

and making grounded judgements from within that context (Coyne 1995). Pragmatic design is primarily concerned with interpreting and setting expectations, as objectivism and universality give way to contextualisation and constructivism. Community replaces individual and vested authority, pre-existing artefacts become objects for reinterpretation, and in the act of design, situated performance replaces detached planning or execution of pre-existing scripts. In all forms of design, situation dictates to the design act and design evaluation more than theories, universal principles, methods or frameworks.

### 9.3.1 Pragmatism and the ‘social contract’

The pragmatic persona introduced in 2.5.2 (‘The Pragmatic Software Architect’) poses questions that attempt to make pragmatism real for practicing software architects, starting with the basic characteristics of pragmatic practice:

*What characterises the pragmatic software architect?*

The pragmatic software architect designs and constructs in context and in collaboration with other designers and the community of stakeholders at all times. The pragmatist has no need for the pursuit of absolutes and no time for the pursuit of perfection. To the pragmatic architect, theory and practice are imprecise distinctions—if theory is externalised, it is treated as following from practice. Reliance on plans or abstract models as directors of design effort is minimal, and openness to situational, contextual, communal and cultural cues is paramount. The validity of a design technique is measured in terms of its proven utility, community acceptance and relevance to the situation at hand. The pragmatic software architect takes away grounded experiences and heuristic knowledge from a design engagement to inform future work in similar situations, recognising that each new design context will necessitate new interpretations of familiar objects or techniques.

Much of the participant’s accounts fit comfortably within this pragmatic genre. Pragmatism is evidenced by the architect’s recognition that each design situation—particularly the actors and their individual and collective capabilities—is different, and that to design independently of these capabilities or to some imagined or idealised capability is at best inappropriate and at worst negligent. Prime examples of pragmatic design techniques and behaviours include using software architecture as a mechanism to distribute work across a team, and the use of framework design techniques to encapsulate or distribute complexity based on team capability.

The phenomena of requirements, problem and solution negotiation also evidence the

pragmatic persona. The key pragmatic concept is the ‘social contract’. The pragmatic designer understands requirements as a contract between stakeholders in a given situation and design as the process by which such decision-contracts are mapped to a system or technology infrastructure. The pragmatic architect advances software architecture on a foundation of agreements, knowing that most will be sufficiently stable for the purpose at hand, and that defensive design techniques, design skill and re-negotiation will allow re-conceptualisation of those that prove unstable. Correctness is less relevant than *viability*—the combined stakeholder’s confidence at any point in time in a set of social contracts between an *ad hoc* cabal of stakeholders, of which the architect is but one. The first case study (Breuer, Chen, Nygaard and Goldberg) richly illustrates this combination of designer behaviours and phenomena. The pragmatic architect uses the architectural vision as a vehicle for contract-building with stakeholders and as an initial anchor-point in the negotiation. From their vision, with dialogue, specific agreements (contracts) solidify. The architect’s vision, and the contracts that follow, are shared interpretations rather than binding absolutes.

### 9.3.2 Pragmatism and paradigm

Pragmatic architects do not allow method to direct their design effort—instead, they design flexibly, at all levels of abstraction seemingly at once, under the direction of exigency. The finding that the participants could not identify significant differences between designing at the level of software architecture versus designing at the level of components or objects evidences this. It is also a testament to the scalability of the object paradigm—object-orientation facilitates a consistent use of design skill through multiple distinct levels of abstraction in a way that would not be possible in other paradigms. The implication is that pragmatism—and freedom from the dictates of method—is enabled by the design paradigm. The characteristic of the object paradigm that allows this designer behaviour is its self-similarity—a consistent and self-referential paradigmatic meta-model dissolves distinctions between conceptual, architectural, logical and physical design. Freed from distinctions and dependencies between conventional modes of software design, architects are able to break from sequence, instead working freely and interchangeably at all levels of abstraction. In effect, by supporting architectural design, physical design and everything in-between, the object paradigm unifies these previously segregated design modes such that they all look and feel the same to both enactor and observer.

### 9.3.3 Pragmatic knowledge structures

*In pragmatic, situated practice, how is generalised design knowledge explicated and passed on?*

The finding that architects describe their experience as an accumulation of ‘small-p’ patterns in a kind of personal pattern store is also strongly pragmatic. It replaces the notion of structured, catalogued personal knowledge with one of accumulated epistemological bricolage. In this loosely bound and unstructured accumulation, each collectible knowledge fragment is retained by virtue of the architect’s memory of how it helped solve a problem in a particular situation. Conventional notions of structure, taxonomy or organisation cannot be superimposed over such a dynamic and value-laden storage and retrieval system.

That architects do not attempt to structure their personal patterns, or express concern that some may constitute re-invention of published patterns, reinforces the architect’s pragmatic stance on accumulated knowledge. Architects recall previous solutions most effectively only in the context of a presenting problem, and it is the need to solve a software design problem that drives the personal process of recall and re-conceptualisation. The first point essentially declares that recall is situated and that it increasingly loses meaning as it is removed from context. The second point illustrates the architect’s implicit awareness of the cost of recall, and serves to explain one reason why professional software architects do not generally practice externalisation of their experience—their propensity to protect their personal experience notwithstanding. The architect’s accounts of their use of system archetypes (for example, *Stickley’s* narrative) echo the same themes of personally relevant, minimal, contextualised knowledge and the situated nature of its recall.

In the participant’s accounts, knowledge is shared in context and by peers in an ongoing dialogue, both with each other and with the materials. Knowledge exchange via social networks echoes aligns with communication theory (Allen 1977). Knowledge explication and sharing is via social processes and design occurs as a consequence of the designer’s participation in this social network. The pragmatic architect therefore realises the power of the vernacular design models to design efficiently and adaptively, as well as suffering their inability to scale or generate theory from which new designs and designer behaviours can be predicted.

### 9.3.4 Pragmatism and the design act

*In pragmatic, situated practice, how are designs created, or arrived at?*

In the discussion of rational themes, the evaluative part of the design episode was declared to be a rationalistic action. It remains to determine in which personas the creative part most comfortably sits. When the architects talk of ‘creative or intuitive leaps’ they describe moments of unforeseen insight, of making previously unrecognised associations or connections, or of picturing new concepts, abstractions or mechanisms that resolve conflicting forces. Doing this creative part of design consumes all of their designerly powers, and as might be expected, the designers themselves have some difficulty describing the act or the moment with any degree of specificity, as these accounts from *Breuer* and *Stickley* suggest:

I went home and thought about it at every level... I don't recall *not* watching television... yes I do. I think I just fiddled with some papers, played piano, did something, let it lie fallow for a bit, but it kept on re-echoing... I know I probably didn't go to sleep very early, it was probably more like 3 o'clock in the morning... *Chen* had encapsulated the problem exactly, and I was predisposed to thinking about representing business rules in a hierarchy... it kind of seemed just obvious from experience. —*Breuer*

I'm not conscious of it... it's subtly different each time... and it evolves... it's not possible for you to forget it, it's not a bit of knowledge you've got, it's a bit of understanding you've got, it's a part of your person... it kind of happens in an instant. —*Stickley*

Design occurs as a pragmatic action when the architect is implanted in the problem space and embedded in the situation. These moments of creative software design appear to be correlated with immersion, dialogue, interaction and collaboration. This research has found that better software architectural design outcomes result when the designer collaborates than when the designer works in isolation. For example, when *Utzon* describes his ‘strategies’ as he approaches a software design challenge, he mentions ‘recognising his own game’, reflecting on the problem and his approach to it to make design drivers conscious (rather than leaving them subconscious). Then, he describes exercising whatever kind of leadership the situation dictates—‘and that can be everything from trying to conciliate amongst people, to, in some circumstances, dictating, if that's what needs to happen, because, ultimately, it's the team effort that matters at the end, not the individuals who have come together to build the system’.

Explicating design drivers and exercising contextualised leadership sets the scene for the pragmatic designer to practice many of the behaviours discussed in the findings (Chapter Eight). In the pragmatic design act, problem negotiation goes hand in hand with architectural pattern and archetype-fitting. *Cook* confirms this when he declares that ‘the



most frustrating part of architecture and design... is when I come across something that doesn't conform... when I can't make it look like something else I already know'. Design is 'always an application of patterns', he concludes. There are other findings that illuminate the design act. In essence, these findings combine with *Utzon's* collaborative leadership to close in on the question of what situatedness actually means for software design:

*What form do situatedness, opportunism and being open to contextual cues take in practice?*

Firstly, the design act appears to often centre on a previously overlooked association, or the recognition of an association that only becomes obvious after a change of perspective or a key abstraction changes as a result of a new piece of information. Generalising from this observation would lead to the theory that creative software design is a pattern-matching process in which the formation of associations between abstractions represents the key to understanding the design act. This leads back to the typographical models of design discussed in 3.3.1 ('Categories of Design Models'). Other accounts (*Sullivan, Cook, Lethaby*) suggest that the creative design act is more unpredictable and complex than can be explained by a process of typographical matching alone. The participants confirm that they work on multiple dimensions of the design engagement concurrently (of which typographical associations are but one). The architects describe varying their abstractions, the abstraction's associations, relaxing and tightening constraints when creative breakthroughs occur within a design episode. Typographical pattern-matching therefore aligns with only one dimension of the multi-dimensional process of creative software design.

Thirdly, the creative or intuitive leap is described as sometimes being consummated by the momentary adoption of an alternative perspective, or in the case of a modelling breakthrough, an alternative paradigm, by the same or a different designer. Such breakthroughs come through combination and re-conceptualisation—fleeting interpretations, fuelled by a particular combination of circumstances. While situational factors do not *drive* the creative design process, they combine to assemble its context at any moment in time, and as a result, they feature prominently in the architect's accounts of intuitive leaps and creative breakthroughs.

### 9.3.5 Pragmatism and method

Unseating the rationalistic notion of methodology as controller of design practice resulted

from the findings and discussion of the rational persona. This leaves the question of what pragmatism means for methodology. A pragmatic method cannot strictly sequence activities, dictate practices based on preconditions, or impose universal laws on design activity. Some re-conceptualisation of methodology is necessary in order for it to support this design mode:

*What does it mean for methodology to be contextualised, to become implicit in a design—to become situated?*

To generalise from the architect's accounts into some form of 'method' for the creative part of the design act would be both a denial of the argument for pragmatic software design and an implausible stretch. However, some characteristics did emerge which shed light on the phenomena. Contextualisation of a method occurs in the hands of the practicing designer, who has learned to reflect on actions, before, amidst and after the action is completed. This individual and collective reflection constitutes a source of feedback that drives method contextualisation at both gross and fine levels. Contextualisation of method is the process by which a method becomes situated. However, as previously noted, this is definitely not a pre-existing method that 'becomes situated' through some process of method meta-engineering, fitting or adjustment. Instead, it is the designer's conscious and sub-conscious assembly of method fragments, tools and prior design knowledge ('personal patterns') for the situation and task at hand.

Tools are a significant enabler in this contextualisation. When it comes to assisting the design act, methods are cumbersome, bureaucratic and inaccessible, whereas tools are at-hand. Tools (*Gropius* names entity relationship models and sequence diagrams) are highly scalable, in accordance with the diminishing differences between object-oriented architectural and component design, or between conceptual and detailed design. To assist pragmatic design practice, a tool should be applicable across levels of abstraction, across the design's lifecycle, across architectural representations at different levels of maturity, and across domains. It must facilitate designers to move freely between creative design and rational evaluation, and should facilitate interaction between stakeholders and collaborating designers. And it must be able to be internalised, used without computer support, used 'lightly' and effortlessly, such that the designer's awareness of the tool's existence diminishes in use, like Winograd and Flores' hammer (1986).

*Gropius'* description contrasts the lightness, useability and usefulness of tools that empower him to move forward in a design engagement against the comparative heaviness of a method. The same principles of tools and tool use apply to how pragmatic software

designers assemble a custom design process comprised of heterogeneous tools, techniques, ‘personal patterns’, and archetypes. In both cases, method is internalised but its elements are reified via use or reuse of tool, technique, pattern or archetype. The question ‘how does methodology become contextualised’ is therefore misleading. Pragmatic software architects do not contextualise a given method—instead they assemble atoms of a method with atoms of other methods, and with other knowledge artefacts, to suit the problem and situation. What results is a method of sorts, but one that bears only fleeting similarities with what most practitioners would think of as *methodology*.

### 9.3.6 Aesthetic as reflective practice

Contextualisation of method relies on continuous feedback in the designer’s personal design process. Personal reflection amidst practice occurs in a number of ways:

*How do experienced software designers employ reflection when they design? Do they acknowledge reflection at all?*

Some participants described reflection in terms of an aesthetic sense of an architecture or code base during its design and development. They described reflection not as a distinct activity or end in itself but rather as being anchored to a tangible artefact—in most cases a specific class or module within the evolving code base. On face value, an aesthetic sense appears purely *a*-rational and strongly pragmatic. However, those who were able to describe it in more concrete terms identified consistency, minimalism, symmetry and self-similarity as hallmarks of the phenomenon. Interestingly, all of these characteristics of software architecture can be objectively measured. But as none of the participants stated that they made design decisions on the basis of such measurements, ‘software aesthetic’ must be treated as a phenomenon to be interpreted.

Software aesthetic sits most comfortably in the pragmatic persona as the basis of the designer’s approach to decision-making. Design decisions informed by a sense of, say, self-similarity or symmetry, cannot be automated—some breaking of symmetry, for example, may be tolerated because of other overriding design concerns. Some of the participants talked of using their software aesthetic sense to detect ‘breakdowns’ (particularly ‘soft’ breakdowns) which regulate the alternation between design modes within or between design episodes. What might be considered a breakdown on aesthetic grounds to one designer may be tolerated by another—hence software aesthetic is a personal interpretation of a situation which involves both objective assessment of architecture and code structures as well as subjective assessments of other factors, such as the motivation of the available

people to remediate or fix problems, or other project considerations at the time.

The ‘emergent designers’ describe their sense of aesthetic changing with time, and being influenced by the individual’s growing awareness and experience of personal patterns, archetypes and solutions. There is evidence that this aesthetic sense is shared—*Breuer* reports that he, *Chen*, *Nygaard* and *Goldberg* shared a sense of partial (and finally complete) fulfilment with each subsequent design episode that marked the engagement’s design trajectory. *Morris* and *Lethaby* both described getting team agreement on matters of architectural degradation. The possibility of agreement builds evidence for its underlying objective basis. These accounts suggest that software aesthetic is a designer’s intuitive and even sensual response to perceptions of objective design and code quality. A rationalist might label these self-proclaimed code aesthetes lazy, because instead of working to resolve problems in the design or code base to objective terms, they instead use subjective assessments and terms. While this probably does no harm, it signifies immaturity rather than an area of design practice that is naturally situated.

### 9.3.7 Does pragmatism have a role in explaining software design practice?

The overarching question concerns the usefulness of the pragmatic persona as a means to describe software design:

*What aspects of software design does the pragmatic persona usefully describe?*

This discussion of the pragmatic perspective can be concluded with two points—the usefulness of pragmatism’s description of the creative part of design, and the usefulness of a pragmatic view of the design process overall. Firstly, the pragmatic persona suits the participant’s reported experience of design activity particularly well. The evaluation of the rational persona concluded that design effort and design outcomes are not proportionally related, and that the design trajectory is not amenable to conventional planning regimes. The pragmatic persona explains why these characteristics are as they are. The process of pragmatic software design is best evidenced by the Decision Tree story, in which a design breakthrough resulted from the interaction of four highly experienced designers, each with their own different personal perspectives on the same problem. The pragmatic elements included self-regulation (they followed no externalised method or script), demarcation of paradigms and perspectives (each designer worked from a different view of the problem), willingness to work together (no participant was dominant), egalitarianism (they all agreed when the ‘best’ design option was reached), and a critical motivation (they pursued their designing ‘under the radar’ of the project regime).

Secondly, pragmatism as a general model of software design activity suffers from the limits of vernacularism. If pragmatic design is to achieve acceptance within the software industry (and particularly within the discipline of software engineering) it must resolve its opposition to conservative methodology. *Utzon* comments that ‘people want to see a process that you are following... they certainly don’t want to rely upon some sort of artistic binge that happens late one night’. The Agile methods movement has addressed this to some degree, and hybrid methods like Feature-Driven Development (Palmer and Felsing 2002) are starting to bridge the gap between Agile and conservative methods. Pragmatism is closer than rationalism to being a uniform basis for the experience of software design. It accounts for all of the things rationalism cannot—a design’s unpredictable trajectory and the non-linear effort-return relationship, for example. Whereas rationalism depicts these phenomena as inadequacies on the part of the actors—their inability to correctly follow rational processes of discovery and design execution—pragmatism explains them in terms of situated human cognition. Pragmatic design in software, as in other mediums, richly engages the designer in context, setting the designer free to assemble, amalgamate, combine, re-form and reuse.

## 9.4 Critical themes

Critical theory targets the pervasive structures that exist to preserve the *status quo* by asking ‘who is in control?’ A critical assessment of a situation leads to a discernment of the distribution of power, to the revelation of power structures, the oppressors and the oppressed. Design is, by its nature, the process of altering the world, and the design of software and computer systems always has the potential to change the distribution of power. The ways in which software architects consider these issues in a design engagement are largely unknown. This research provides some data on the matter.

### 9.4.1 Critical analysis and the design engagement

Experienced software architects will have worked in a variety of organisations, projects and business cultures, and will have experienced project failure. Participation in failed system projects inevitably leads to reflection and interpretation of the reasons for failure (Petroski 1992). Breadth of experience develops a critical perspective of software design in the professional architect, by which methods and project structures may be deconstructed to reveal underlying power structures. Although the software architect might not need to be the controller of all things technological, professional architects

should know where control lies. A critical view of methods and project structures as devices for assertion of power between competing factions inevitably emerges in any examination of software development teams or designer behaviour:

*Is there evidence to support the assertion that software architecture and design can influence or redistribute the balance of power in a design situation?*

Some of the participants described finding themselves amidst situations of inappropriate distributions of power, but usually as a result of other's decision-making rather than their own. The situations described by *Lethaby* and *Breuer* in which they were engaged via a software product vendor require little critical analysis to be exposed for what they are—an open invitation for the vendor to control the design engagement to achieve their own ends, usually at the expense of both the architect's independence and design quality. Once in place, the vendor-architect's ethic is the only authority that can halt unilateral appropriation of the design process to extort commercial value from the engagement. Fortunately, in both stories, the individuals appear to have been able to redress the imbalance—*Breuer* by investing intensively in design effort to ensure that the solution architecture delivered via his employer's product was fit for purpose, and *Lethaby* by counter-arguing on technical grounds. Both stories vindicate the architect's ethical conduct and account for how architects can act to reorient the balance of power within a design engagement in order to ensure both the delivery and quality of software product.

The finding that few of the participants understood the overall totality of their designs suggests that software architects are generally unaware of the degree to which their design decisions can change the balance of power. Software architects are complicit in acts of deception when they do not act with adequate knowledge of the consequences. To proceed with the design of a system architecture that maximises one's employer's product license revenue is a significant deception. Knowing that architectural degradation is occurring within a code base and not attempting to initiate corrective action is similarly negligent (although less visible) and an abuse of the power invested in the architect role. The participant's views on inequalities in the design engagement suggest that they recognise imbalances but often feel ill-equipped to effect resolution, *Lethaby* and *Breuer's* stories notwithstanding.

There was evidence in the analysis of software architects using their design role to benefit themselves, if not transfer power to themselves or their teams. One example is *Le Corbusier's* engagement of a few well-placed telecommunications engineers during the design process allegedly (according to *Mackintosh*) to position his company to win further

work. Descriptions of design engagements being used as an opportunity to acquire marketable skills is another, although designers have always used their client's resources to indulge in new materials and new styles. A critical analysis of this behaviour exposes the individual architect and his adherents as mutual benefactors of the design engagement. As long as the architect does not allow the allure of a new technology to fog his judgement of its suitability, no harm is generally done. The case studies suggest that where architects have worked new technologies into architectures, risk has been introduced not as a direct result of the choice and deployment of the new technology but rather as a result of the team's lack of knowledge and delivery experience with the new technology. Under critical analysis, some of the participants may be guilty of risking destabilisation of a design project in such cases.

#### 9.4.2 Design as a means of resolving inequities

A designer's sense of responsibility also concerns use—how the product may change its environment in the hands of users. A designer may design a tool without concern for how it is used, or may go further to consider in what contexts the tool will be most used and how it might change these contexts from a social, economic, cultural and humanitarian perspective:

*Does the software architect perceive responsibility for how the product or system will be used, or for the implications of its use on people, regardless of their role or position?*

In general, little evidence was found that software architects perceive responsibility for how their product or system will be used, or for the implications of its use on people. A related question concerns the degree to which software designers take responsibility for intervening in inequities, as are often found in environments of sweeping change on the back of technological change, social engineering or workplace reform:

*Do software architects regard themselves or their designs as resolvers of inequities?*

Again, there is no substantive evidence in the analysis for this. On the contrary, architects like *Voysey*, *Ashbee* and *Eames* expressed a strong sense of powerlessness and an inability to influence the mechanics of the business enterprises within which they found themselves. In this regard, software architects may differ from their built world counterparts. Development of a wider social conscience for IT architects, hand-in-hand with education in professional ethical practice, could be an important project for IT professional bodies in

the future.

#### 9.4.3 Criticalism and method

Methods are ostensibly adopted by organisations to improve the quality and timeliness of software design. They are frequently justified as a means of reducing system development risk, but in a critical light, as a means of distributing or re-allocating risk:

*What purposes do methods serve in the context of organisations and the competing parties engaged in economic enterprise?*

The participants are highly critical of the purpose of method in some design engagements, often carefully interpreting the method's specific prescriptions, to ascertain how its use will advance or constrain their options. In general, architects will follow a method that they assess as being useful in providing an organisational or administrative function for their project team but benign in terms of the distribution of power. Under critical analysis, the failure of *Breuer's* team to deliver quality architecture in the first case study suggests the use of the project's process as little more than a tool to eliminate discovery and experimentation and to force a fallacious and dangerous façade of progress. Ultimately, by enforcing a communication and collaboration embargo between the teams in the mistaken belief that such collaboration would invite opening up previously closed architectural decisions, the project's management deceived only itself. The use of methods as a vehicle for exerting control on project teams makes architects believe they are justifiably wary of methods, and this further erodes their confidence in them. In some situations, software designers deliberately subvert authority (or method) in the interests of what they believe to be design quality:

*What compels software designers to apply post hoc rationalisation in their accounts of design work performed, or to mask investment in design or architectural quality from their managers?*

*Post-hoc* rationalisation is a documented phenomenon in design in general (Crellin et al. 1990) and in software projects (Parnas and Clements 1986). A number of the participants described covering the true costs and activities of software and system design. *Breuer* had no mandate to change his project's direction mid-stream and most of the Decision Tree's design and prototyping was his personal initiative, motivated primarily by a desire for the adoption of a better solution option. *Eames* also described various architect's attempts to improve his system's architectural shortcomings 'under the radar' of the project's planning and management regime. Others reported behaving similarly at times. The compulsion for



architects to take the trouble to make improvements to architecture independently of management support appears to be personal pride in workmanship and in design quality—both in the way the design process is performed and in the product itself.

#### 9.4.4 Does criticism have a role in explaining software design practice?

The overarching question concerns usefulness of the critical persona as a means to describe software design:

*What aspects of software design does the critical persona usefully describe?*

Three kinds of self-reported behaviours can meaningfully be considered critical. The first is sensitivity to the power-influencing capability of project structures and methods, the second is an awareness of compromises inherent in their role in a design engagement, and the third a propensity to subvert project authority to achieve certain design goals. All three are behaviours that attempt to preserve the designer's independence and ability to fulfil their role.

Beyond their personal myopic orientation, the architects generally cannot see the wider implications of their designs and do not consider them particularly important. This is perhaps unsurprising—the relative immaturity of software design as a discipline would suggest an immature social conscience when compared with designers of consumer products or buildings. The cases they describe where they intervene or subvert authority or power structures are, in general, due to uninformed, badly conceived or improperly executed design engagements. This may also reflect role immaturity in industry. The critical persona is therefore most useful as a view of software design practice that serves to explain designer self-protection and self-preservation, particularly when designers find themselves in situations not of their own making.

## 9.5 Radical themes

Finally, the radical persona introduced in Chapter Two encapsulates the self-evaluative and self-challenging behaviours of designers. Radicalism is in direct opposition to rationalism because it repeatedly attacks the foundations of orthodoxy in a tireless quest for new insights. Radicalism would appear to be a valuable characteristic in software design because it has the potential to drive alternative conceptualisations in a design fabric that is almost infinitely malleable. The radical software architect uses re-conceptualisation to discover new truths in various situations. Acting to deconstruct a situation, the radical

software architect reverses, inverts and demolishes opposing cultural positions that are revealed from analysis of ‘texts’ (any artefact that can be read, observed or interpreted), generating subversive discourse that challenges the foundations even as they are being built.

#### 9.5.1 Radicalism and conceptualisation

A software designer might engage this stance by first recognising, then deliberately and consciously subverting design orthodoxy:

*What techniques do software architects adopt in order to question or evaluate the use (and reuse) of familiar (or popularly subscribed) designs or design methods?*

In essence, this question concerns how software architects subvert incumbent methods, conventional approaches and assumed positions. There was little evidence for this kind of practice in the analysis. One explanation is that software architects are frequently engaged after decisions about technologies have been made, so their opportunities to employ radical thinking are limited to requirements and architecture negotiation, as has been discussed. Radicalism requires stark awareness of one’s assumptions and the cultural norms of the context of a design engagement. Software architects are often so deeply embedded in the minutiae of software development—programming languages, technologies and platforms—that such awareness is not often called for.

*What processes do software designers use to argue for, or build a case for a particular design solution over others?*

Perspective and paradigm-shifting (T6.35), whereby architects actively and consciously shift their paradigmatic perspective on a problem in order to consider an alternative conceptualisation, may be considered a radical behaviour. Paradigm shifting is radical because it challenges and (at least temporarily) overthrows the orthodoxy—the assumed paradigm—in order to ensure that opportunities for better solutions are not missed. Despite the apparent value of this phenomenon software architects rarely practice it. One reason may be that contemporary application development technologies and platforms (such as J2EE and .NET) do not support different paradigms but instead enforce reuse of standard paradigm-locked architectures, frameworks, archetypes and patterns. Another possible explanation, as evidenced by *Breuer’s* description of *Chen*, is that the years of experience required to fill an architect’s role embeds the designer within a paradigm and technology set. Even so, many of the architects reported experiencing considerable

freedom within the constraints of a technology, paradigm or language, particularly where modelling is concerned.

The question of whether paradigm shifting is useful remains. While it sounds like an appealing skill for an architect the lack of evidence of its use suggests otherwise. The theoretical argument is that of Feyerabend. A propensity to generate alternatives, or to remain open to design alternatives that have been rejected in the past, is not primarily about an ongoing investment in a range of design alternatives but a meaningful discourse with others as to why selections are made:

*Do software architects always understand how and why a particular design was chosen? If they do, can they usefully act on such insight?*

Awareness of options leads to a sensitivity to the benefits of deconstructing the assumed perspective or paradigm. For the software architect, this may mean adopting a paradigm-independent perspective in order to understand why the design's current trajectory is the way it is. The evidence from the analysis suggests that the architects do not, in general, question how and why a particular design option was chosen. Limited awareness of explicit paradigm or perspective-shifting, and the propensity of architects to design from within a technology vertical are two potential reasons. Another finding reinforces this conclusion—the individual architect's accounts of design breakthroughs and creative leaps are predominantly intra-paradigm and do not generally challenge the designer's assumed paradigm.

Apart from paradigm-shifting, negotiation of problem and solution characteristics presents an opportunity to adopt a radical persona. For example, in *Cook's* account of his 'most successful' project architecture, he attributed his success to his ability to 'continually take problems, and re-evaluate them in the context of what we'd already done'. His negotiation of new stakeholder requirements *in the context of his existing architectural framework* represents radicalism at work. His propensity to deliberately choose an interpretation of the problem that countered that which a rational analysis would have produced can be seen as either selfish or sensible. In his accounts, he clearly regarded it as the root cause behind his best work.

#### 9.5.2 Does radicalism have a role in explaining software design practice?

The overarching question concerns usefulness of the radical persona as a means to describe software design:

*What aspects of software design does the radical persona usefully describe?*

Paradigm shifting represents an appealing radical behaviour but there is little evidence of its use by individual architects. However, *Breuer's* case study clearly illustrates its power to dramatically improve design outcomes by stimulating the design trajectory at key points. In summary, the power of radicalism to generate new and useful insights and options appears to be unlocked by designer collaborations.

## 9.6 Conclusion

This chapter has discussed the correspondence between the themes introduced in the background chapters (Chapters Two to Four) and the findings (Chapter Eight). The four architect personas (rational, pragmatic, critical and radical) introduced in Chapter Two were used to structure the discussion. As predicted in Chapter Two, the findings support the relevance of each persona for explaining certain aspects of designer behaviour and design outcomes, but no single persona offers an entirely prescriptive view of design practice. The findings predominantly support the rational and pragmatic personas. There is evidence that designers alternate between these two design modes, and that this alternation fits Simon's (1985) Generate/Test model. The behaviours that associate with the personas do not generally overlap—for example, the critical behaviour of hiding design effort is strongly evidenced but is in no way rational or even pragmatic.

As a result of the discussion, one additional conclusion can be drawn. The findings evidence specific ways that expert software designers work to achieve design outcomes. Many of these techniques and behaviours share a common characteristic—under the designer's control, they modify the designer's environment or context to increase the likelihood of achieving a successful design outcome. For example, the architect-negotiator negotiates and rearranges scope, requirements, system structure, even perceptions of the problem itself, in order to make them more like problems for which known and trusted solutions exist. The architect influences and arranges both external (stakeholders) and internal (team) elements of context, working explicitly and implicitly, with the objective of making the design engagement more familiar and therefore less exposed to risk. In simple terms, experienced software designers make the presenting problem look like one they have solved before and alter what they can in the context to make it similarly familiar. Once rearranged, the context supports and facilitates the architect's execution of a personal design process. This observation reinforces the importance of situation in the practice of software design.

## Chapter 10: Conclusions

Inappropriate, expensive and disruptive interventions in all sectors of working life are continually reinvented based on naive and dangerous fantasies of control and order. Many of those who work with or under these systems, or who are involved in attempting to implement them, are dubious of their worth, but often argument against them is characterised as “user resistance”, that is recalcitrance, rather than the articulation of real flaws in their underlying plan-based approach. Frequently problems with these systems are diagnosed as “implementation failures”, or due to inadequate scoping of the system, rather than as due to any problem of principle. (Johnston 1999, p. 146)

### 10.1 Introduction

This final chapter concludes the thesis by resolving the running argument (that is, the hypothesis) for viewing software design as situated action. In the preceding chapter’s discussion, the findings were contrasted with theory to reinforce their significance and relate them to the epistemological base of knowledge from where the research aim and hypothesis originated. This discussion was organised around four philosophical positions. This thesis has proposed that these modes of designerly behaviour collectively constitute the ways that expert software architects design, and that to analyse a designer’s actions or interpret a designer’s behaviour out of context and from only one of these four perspectives is always flawed. This thesis has proposed an alternative characterisation—that of the ‘situated’ software architect—and the investigation and characterisation of this construction has motivated this research.

This chapter resolves the hypothesis—*that software design can only be meaningfully understood when viewed as situated action*. Evidence from the research findings for and against the hypothesis is reconciled to reveal a final position. The thesis concludes with some emergent hypotheses and clarification of the research contributions.

## 10.2 Response to the hypothesis

In Chapter One, a hypothesis was declared that forms the central argument of the thesis. The hypothesis (that *software design can only be meaningfully understood when viewed as situated action*) goes beyond the basic question of the relevance (or meaning) of situatedness to software design to ask whether it is an *essential* perspective for any meaningful account of software design. It challenges a rationalistic account in two ways. Firstly, it insists that situation plays an undeniable role in the most basic act of software design. More significantly, it asserts that a software designer *must* be situated, or must be subject to situational forces that transcend rational process or method as a design driver, in order to design. As a consequence, an understanding of design can only be arrived at when these situational forces and their impact on the design process are acknowledged and understood.

The hypothesis is not intended to represent an *either-or* choice. This thesis has not set out to replace a rational model of software design with a situated one. Rather, this thesis has assumed a plurality of models of design and has set about depicting the situated model in more detail than has previously been attempted for software architects. Suchman (1987) describes the situated approach as studying ‘how people use their circumstances to achieve intelligent action’ (p. 50). To find in favour of the hypothesis is to promote circumstance—in the hands of the expert designer—to (at least) peer status with both rational theory and method as a primary factor in how software structure is created.

### 10.2.1 Revisiting definitions

To take a position on the hypothesis, we need to clarify or restate the working definitions of ‘software design’ and ‘situated action’. Throughout this thesis, ‘software design’ has been taken to refer to those actions used consciously and sub-consciously by software architects to create software architecture and artefacts, be they direct manifestations of architecture in code or indirect manifestations in the form of models. *Sullivan’s* definition of software design—‘putting a design where none previously existed’—was adopted for its emphasis on creativity as a skill that not everyone possesses. It also demarcates software creation *viz a viz* the enhancement or elaboration of pre-existing software structure (although the distinction between the two blurs). ‘Software design’ therefore means the act of software creation, particularly the selection, reification and synthesis of software structure, in the business and industry context defined in the Research Aim in Chapter One (1.4).

The use of the term ‘situated action’ in the hypothesis to describe software design links design and action, and this linkage focuses its definition on what has been referred to as the ‘design act’. The participants were asked to reflect on and describe their experiences of producing their ‘best’ software architectures and designs. These accounts produced the grounded theories on abstraction, personal patterns and archetypes, design episodes within a design engagement, breakdowns at episode boundaries, and aesthetics as a means of reflection. These are typical ‘situated actions’, and their descriptions in Chapters Six (Qualitative Analysis) and Eight (Findings) provide the ‘thick’ contextual descriptions that allow the reader to perform their own interpretation. These ‘situated actions’ also constitute the main evidence for the hypothesis. Given these definitions, we can make a response to the hypothesis on the basis of this study’s findings.

#### 10.2.2 Evaluation of findings against Chapter Five’s design framework

A framework was assembled in Chapter Five (5.3.4, ‘A framework for eliciting descriptions of situated software design’) to characterise rational and situated extremes of designer behaviour. This was used to structure the participant interviews. This framework provides a useful structuring mechanism for assessing the hypothesis. The framework’s *identity* dimension contrasts emergence or appointment of the designer’s role. This is similar to the *control* dimension, which also focuses attention on role self-selection. Design control is also concerned with how designers exert control over the ‘management of meaning’ (Markus and Bjorn-Andersen 1987) in a team’s emergent culture. On the principal designer role, the findings support both self-selection and appointment. *Breuer’s* case study clearly depicts self-selection based on distinct design episodes and each designer’s differing paradigmatic views. However, a number of the participants reported a strong sense of identity and ownership of the design, including *Le Corbusier* (second case study), *Cook*, *McLuhan* and others. Although *Le Corbusier’s* Exchange-centric architecture was flawed in its initial form, it is impossible to draw the conclusion that the designer’s tight control of the design process was to blame. More significant than whether the lead designer’s role at a point in time is self-selected or appointed is the finding that most creative design acts are performed by a designer working in isolation, and that collaboration serves rational design activity more than pragmatic, creative activity in the participant’s reports. The importance of identity and role clarity in situated design activity is therefore as an indicator of the freedom which exists in the design engagement’s context for design expertise to emerge in response to contextual drivers and cues. There is evidence to suggest that designers actively seek out or reject collaboration and sharing of design responsibility.

The framework's *planning* dimension contrasts the designer's belief in plans as predictors of design effort and the unpredictable nature of design. The findings strongly confirm the unpredictable nature of design and the architect's general mistrust of plans and planning regimes. They acknowledge little correlation between design effort and plan content. They report the hiding of design effort in unrelated plan phases as well as *post hoc* rationalisation of design effort. The findings also report assembly of the personal design processes *in situ*, regardless of direction from external methods or plans. This tension is not new, and the suggestion that the rational planning paradigm for creative work be abandoned has surfaced in the Agile methods (see for example the 'planning game' (Highsmith 2002)). The conclusions are that architects are guided minimally if at all by plans and that design advances are mostly unrelated to plans. Conventional plans should therefore be considered as useful only in providing high-level phasing of architectural design effort and activity.

The framework's *generator* dimension uses the designer's choice of generators (concepts and patterns) and the design process they use to elaborate the generators into solution design as an indicator of situatedness. In a situated design scenario, generators are contextual, as is the process by which they drive architectural structure. The findings are divergent around the role of generators in software architecture. On one hand, *Breuer's* Decision Tree story exemplifies how a generator can collapse architectural complexity and breadth across multiple layers of architecture, and *Utzon's* dedication to the discovery of 'self-similar holons' echoes the theme. *Breuer* also enunciates the importance of grasping the conceptual core of the problem, in a mathematical sense. *Le Corbusier's* 'exchange' architecture similarly depicts this approach and reinforces the pivotal importance of getting the primary generator right. On the other hand, others of the participant group did not emphasise the importance of this kind of discovery, instead leaving the impression that their design approach was more routinely methodological, even if process was assembled or heavily customised for the particular design engagement. The concept of generator is therefore both relevant and important in understanding design as practiced, and although it appears not to be essential to the creation of software architecture, it is a distinguishing factor in the creation of high quality software architecture. The role of generators in situated design activity is as encapsulations of a recurring structural theme and as catalyst for elaboration of the architecture.

The framework's *collaboration* dimension contrasts the degree of distribution of design knowledge and the designer's preferences for delegation versus ownership of design



responsibility. Discussion of the findings on both of these points is essentially covered under the *Identity* characteristic. Collaboration alone is not a strong indicator of situatedness on the basis that a rational design approach does not prohibit delegation so long as the roles and design responsibilities are defined and enforced. Collaboration does not exclusively drive design or a design capability, but it does improve its overall quality. As discussed, most accounts of architectural design feature an individual designer acting alone but in context. The *Breuer* case study evidences four distinct design episodes in which each designer performed the creative design act in relative isolation. Through collaboration, these individual designers transcended their personal limitations but at no point was a recognisable act of design performed purely via collaboration. Collaboration therefore is not an essential part of the situated design act. Situated activity is evidenced more by the individual designer's sensitivity to contextual cues during a design episode and not by collaboration with other designers *per se*.

The framework's *process and method* dimension contrasts the claimed versus actual use of public methods to guide the design trajectory. The findings strongly reinforce claims from a wide base of research that methods are not used as expected. Rather, designer intuition plays a significant part of system development, designers 'cherry-pick' methods based upon personal judgement, and experienced designers are more confident in their ability to mix and match techniques from different methods, even those notionally underpinned by different paradigms (functional decomposition versus object orientation, for example). The findings that architects assemble a design process *in situ*, drawing on appropriated tools, techniques and method fragments, collectively build a strong case for the situatedness of design method in practice. The architect's preference for 'tools, not methods' (*Gropius*) explains how this method assembly occurs. One possible explanation—that software designers rely upon rational design methods more often when the designing is routine and less when the designing is conceptual—is not supported by the finding that designers do not distinguish between designing at high or low levels of abstraction. Rather, the designers move flexibly and continuously, cross-cutting both vertical domains and horizontal technology layers, without changing their design approach. This behaviour makes it unlikely that key features of software architecture could be traced to the application of an external or situation-independent design method. It also makes the design product (software architecture) largely dependent on the individual designer(s)—that is, if the designing were to be repeated (by the same or different) designer(s) it is unlikely that the results would be similar. All of these findings strongly support the designer's situation-specific treatment of method, and this in turn demands a situated

interpretation of the role of method in any design scenario or account.

The framework's *reflection* dimension contrasts how the rational and situated models deal with the designer's personal reflection. In all design activity, reflection results in feedback that serves to correct errors in processes and techniques. In the rational model, reflection is recognised within the method's prescriptions, but reflection that would result in significant diversion from, or changes to the method, often causes the designer to question or even abandon the method. In the situated perspective reflection is performed by each individual designer on an almost continuous basis. A reflection-aware design process allows the designer to alter the design's trajectory in both small and substantial ways. The findings make it clear that designers reflect continuously, and that this reflection drives further contextualisation and continuous re-invention of the designer's assembled personal method. Software architects use the notion of 'software aesthetic' as a means of conveying this reflective practice. As a consequence, assembly of the personal design process, abstraction and concept formation, pattern and archetype reification, and other elements of personal design insight are all mutually dependent in the design act. It is impossible to rationally assess one of these dimensions over any other, and all are contextual. Reflection—or 'the software aesthetic' as the participants described it—is therefore an ever-present part of situated software design practice.

### 10.2.3 Conclusions

To conclude, there are findings for and against a situated model of software design. Those that support the situated perspective on action cluster around a common theme—the architect's rejection of externally imposed plans and processes as controllers of the design trajectory. They replace these with a mix of practiced techniques for the situation at hand, including a personalised, discovery-driven design process that they assemble and re-assemble, selection and reification of generators, collaboration with peer designers for evaluation, and corrective feedback from continuous reflection. It can also be concluded that the presence of these phenomena in a given software design engagement does not guarantee a high quality design outcome. The phenomenon of role-emergence illustrates this. As has been discussed, in many successful architectural design engagements, the designer roles are self-selected from, and rotate within the team over a period of time according to skills, level of engagement and the individual's ability to contribute given the phase of design knowledge at the time. However, the presence of this role emergence phenomenon does not guarantee a high quality design outcome. Nor can it be claimed that the opposite behaviour (strong ownership and control of the design process) consistently

impacts design outcomes negatively. The conclusion must therefore be that the phenomena of architectural design practice that are described naturally by the situated model increase the likelihood of good design outcomes but are not reliable predictors of good design outcomes.

This research set out to characterise and describe the design practices of expert software architects. The findings synthesise these rich descriptions such that the reader can interpret them to other designers and design situations. The findings constitute a framework for observation of design practice, for use either by a researcher looking into a design engagement or by the designer himself. The final assessment of the hypothesis is in three parts. Firstly, it is in understanding these phenomena that the situated model of designing is necessary and the rational model of designing misleading. Secondly, the presence of these situated phenomena does not guarantee high quality architectural design or even good design outcomes. Thirdly, the situated model is not predictive, but serves as a means by which to observe and understand the practice of software-architectural design. Thus the rational observer will always fail to account for *a*-rational designer behaviour, whereas the pragmatic (or situated) observer—recognising a full and rich phenomenological picture with consistency and integrity—is able to explain *why* the design trajectory is the way it is, but is unlikely to be able to use this superior observational perspective for prediction.

### 10.3 Generated hypotheses

The process of exploratory research generates hypotheses. Some emergent hypotheses are stated here as natural output of the research process and in the interests of defining future work.

#### 10.3.1 Perspective-shifting

The finding that some experienced software architects are aware of their preferred perspective and are also aware of the effects that changing perspectives can have on their designing opens up a field for further investigation. This thesis has made the following assertions—that designers are aware of their perspective; that perspective is most strongly associated with a preferred paradigm; that designers are prone to becoming locked into a paradigm-based view of a problem and solution when designing and that they find it difficult to break this impasse; and that few if any of the designers evidenced conscious perspective shifting as an explicit design technique. This research also asserts (via the first

case study) the value when different perspectives are brought to bear on a design problem.

The challenge in understanding this phenomenon is to devise research to explore and document it. The phenomenon poses a number of interesting questions. Is there demonstrable value in perspective-shifting? Does the practice lead to better designs as suggested by *Breuer's* case study? Is it realistic to expect practitioners to employ some kind of perspective-shifting technique? Can perspective be separated from paradigm such that other (non-paradigmatic) bases of alternate perspectives could be used? If not, do contemporary software infrastructure platforms (such as J2EE and Microsoft's .NET technologies) lock the architect into a paradigm, thereby eliminating the option and potential value of perspective-shifting? These questions point to a hypothesis such as 'The introduction of alternative perspectives into a software design situation raises design quality (or improves the designed product)'. Adoption of a phenomenological approach would provide descriptions, similar to this research. The question may also be amenable to simulations or observed design exercises involving the collaboration of alternate design paradigm experts.

### 10.3.2 The episodic nature of the design act

The 'design episode' finding—that many of the participants describe a cycle of alternately creating and evaluating an artefact and the descriptions of how and why these modes change—also opens up a potentially rich phenomenological field for further investigation. This thesis makes the following assertions about this phenomenon. Firstly, that designers alternate between creatively proposing or generating designs and rationally evaluating them. Secondly, creation or generation predominantly employs creative design action, evaluation predominantly employs rational design action, and these two modes more or less alternate. Thirdly, changes from the evaluative mode to the creative mode occur as a result of a breakdown, and changes from the creative mode to the evaluative mode occur when the full potential of a creative episode has been explored and exploited in the mind of the designer. A research hypothesis might be 'Software design quality is improved if the designer actively manages the design episode'. This could be tested in protocol analysis, such as is frequently used in design research (Gero and McNeill 1998; Valkenburg and Dorst 1998), in case studies, or *in situ* via the action research paradigm.

### 10.3.3 Methodological support for criticalism and radicalism

The third class of generated hypotheses concerns the findings that describe the role that criticalism and radicalism play in the architect's judgement. A summary of the ways in

which the participants described acting in critical and radical ways (the ‘personas’) was presented in the previous chapter. This research has focused more on the ways that compromised power structures have impacted the software architecture than on describing the nature of the compromise or the situations in detail.

The findings suggest hypotheses that take a phenomenological approach to studying these situations and the designer’s resulting behaviours. With regard to criticalism, questions that would need to be asked include—what situations most frequently compromise the position of the architect? What form does the compromise take? Can patterns be detected in these recurring power structures? How do architects circumvent or mitigate the presenting challenge or compromise? The relevance of this research would be high, particularly as the findings would have direct implication on professional practice.

With regard to radicalism, questions that would need to be asked include ‘Under what circumstances do practitioners adopt a radical approach to design?’ and ‘What practitioner techniques qualify as radical design techniques?’ as well as ‘How can these be incorporated into workable methodology?’.

## 10.4 Reflections on research practice

Finally, several personal reflections on the research process have relevance. A question that concerns qualitative researchers at the commencement of a research project is the number of interviews (or participants) that will be needed before sufficient data will have been collected. Qualitative analysis methods suggest continuing until theoretical saturation occurs—that is, to the point where subsequent interviews produce no new categories but reinforce the existing category model. This heuristic still leaves the likely number of interviews unresolved. In a discussion of this project’s methodological approach, Schauder (2002) suggested that as few as eight interviews should suffice for an IS research question of this type.

This study used twenty four participants (Appendix D) each of whom were screened with a preliminary survey (Appendix C). In general terms, about one third of these interviews turned out to be ‘high yield’ sessions, made so by the participant’s level of expertise and knowledge, their experience, their level of engagement with the interview process and the subject matter (to the degree that they understood it). Their ability to enunciate their thoughts and experience was also significant. Very approximately, another third turned out to be ‘low yield’ sessions which ultimately benefited the research little. A rough measure of the usefulness of each interview can be gauged by the frequency with which the

interviewee's pseudonym appears in the narrative of the qualitative analysis. The chart in Figure 23 was produced by counting the frequency of occurrences of each pseudonym in material prepared for input to Chapter Six. While this metric is dependent on how the prose is written, it provides a rough measure of how each participant contributed in relative terms to the analysis text.

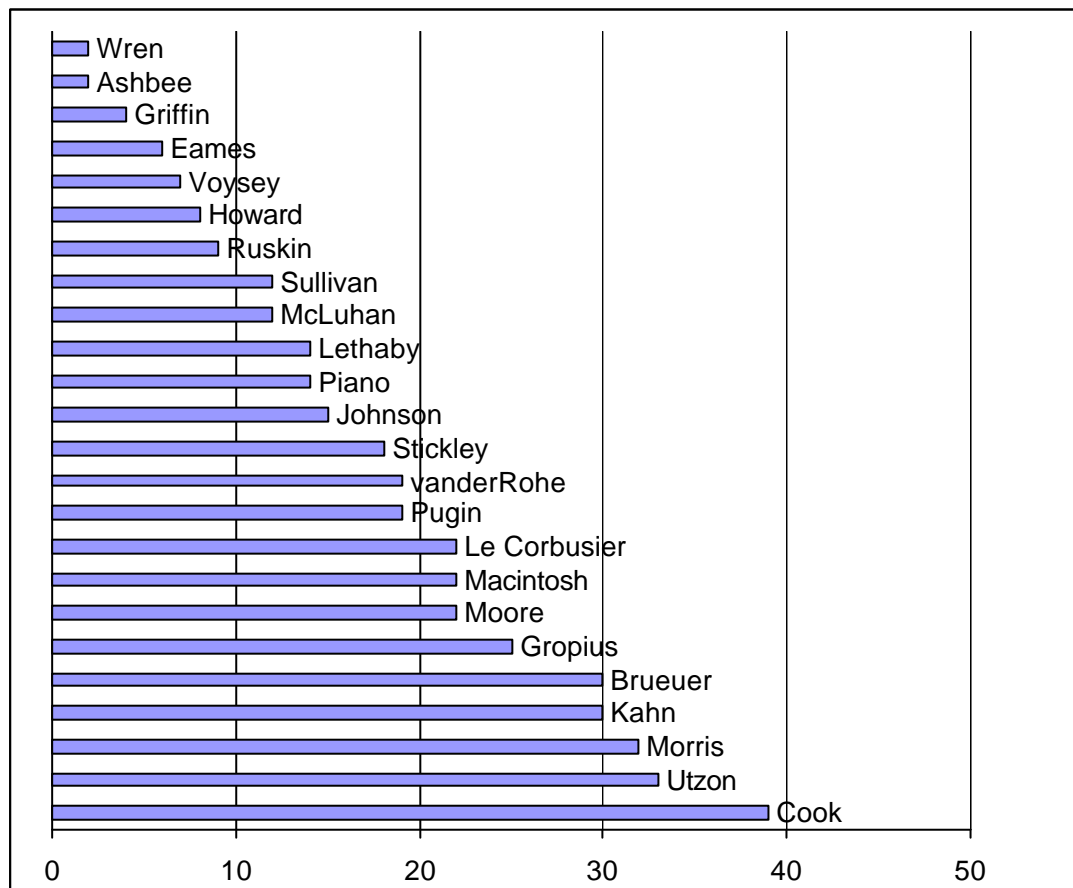


Figure 23: Frequency of participant pseudonym occurrences in this thesis' qualitative analysis chapter.

Figure 23 shows a relatively linear progression from the most cited to the least cited participants. The top eight most cited participants from the group were responsible for 56% of citations in the analysis, while the top 11 realise 70% of citations, which goes close to confirming Schauder's claim. This implies that between half and two thirds of the 24 interviews could have been eliminated with a huge reduction in transcription and analysis effort. The problem for the researcher engaged in a process such as this is to determine *which* 8 of the 24 possible participants to interview and which to reject. Some options exist—for example, the qualification survey could be tightened, and extended with a brief telephone interview. Any measures to increase the effectiveness of the interviews conducted would likely be generously paid back by the savings in laborious transcription

and analysis time.

Having said this, there is also evidence in this chart that *not* interviewing more widely risks missing useful and relevant data. For example, if a tighter pre-qualification process had been used, interviews with *Griffin*, *Eames*, *Sullivan*, *McLuban* and possibly *Lethaby* and *Piano* would never have been conducted, resulting in loss of valuable data and subsequent insights.

The efficacy of using interviews as a data collection technique is most dependent on interview technique—more so than the interview agenda, or even the topics themselves. Each time a new topic is introduced into the conversation, care must be taken not to lead the participant. For example, asking the question ‘tell me about what you do *not* like about the software methods you have used’ can lead to a different discussion than asking ‘tell me about your experience of using software design methods’. Practice improves interview technique as well as refining the category model, and after half a dozen interviews the process becomes second nature.

Interview transcription and analysis must commence early as it provides the feedback loop that enables control of the process. The first substantive analysis effort was commenced after the eighth interview, although note-taking and assembly of the category model was well underway by this stage. Analysis and interviews then continued in parallel for the remaining sixteen interviews and all successive interactions. Qualitative analysis is a deep and continuous exercise in interpretation. The activities of interpreting transcripts, relating an interpretation to other transcript parts and to the emerging topic model, and to theory as well as ‘conventional wisdom’ is intellectually demanding and personally involving. Ultimately, the researcher-analyst is engaged with and bound to the data—engaged to the intellectual core by the search for a final interpretation that provides the sense of resolution, and bound by the personal responsibility to tell the collective story entrusted to the researcher under confidentiality by the participants. This sense of importance and weight of responsibility is a construction in its own right which serves to drive the project to completion.

Another interesting reflection is the way the interview and analysis process generated its own ontology (see Appendix A) including terms such as ‘design act’, ‘design episode’, ‘design engagement’ as well as more esoteric ones such as ‘software aesthetic’ and ‘design trajectory’. Each term has a meaning peculiar to the definitions and categorisations that emerged as a direct result of qualitative analysis of the data. The meanings of some familiar terms were refined during the analysis as well. For example, ‘methodology’ was

initially taken to mean any externalised process to guide the personal and collective processes of designing software architecture. However, it became clear that methodology is an overarching term that encompasses project organisation and roles (of which the designer is one), planning, demarcation of tasks, and design in its various forms. It became necessary to explode the term to ‘public process’ (the publicly visible process), ‘personal process’ (the process assembled and used by the designer *in situ*), and ‘design method’ (the part of a method that attempts to guide design) to more accurately describe what the participants were relating at different times.

## 10.5 Contribution

This research makes new and original contributions to the epistemology of software design, with implications for both further research and design practice.

### 10.5.1 Originality

The central theoretical theme throughout this thesis is the applicability of the opposing rational (or plan-driven) versus situated models of action. In the traditions of qualitative analysis and interpretivist research, the rich descriptions in Chapter Eight (Findings) (which document the accounts and practices of a class of professional designer) constitute one type of theoretical contribution. Beyond these descriptions, this thesis contributes original theory in three main areas—the role of plans as predictors of design activity, the act of software design, and the ways in which expert software designers arrange their context so as to support their designing. These contributions are elaborated in turn.

### 10.5.2 Contribution to design theory

Regarding the role of plans and planning in the practice of software-architectural design (where a ‘plan’ has been depicted as any abstract artefact, including models of software that predict design effort over the period of the engagement), this thesis has argued that plans do not ‘drive’ software design activity in any predictable way. That is not to say that plans are useless or meaningless—plans are undoubtedly useful resources for setting gross-level expectations of work-streams and resource allocation. This thesis has argued that plans do not drive design activity and do not serve as a controller of action where designing is concerned. Plans are best thought of as providing an overarching framework within which design activity occurs. There is no evidence in the findings of any kind of causal relationship between plans and the design trajectory. Similarly, design outcomes are not



causally linked to the existence of plans or models prepared ahead of the designing, and when plans and models are prepared, their relationship and consistency with the final design outcomes and outputs is often dubious. The research findings (Chapter Eight) explain, in considerable detail, why this claim can be made. This research therefore confirms software-architectural design as a domain of human activity that is amenable to a situated model. It also identifies and elaborates the characteristics of situated action in the domain of software-architectural design.

On the act of software design, this thesis has presented a detailed model of the design act—the creative activity by which a software designer puts a design where none previously existed. These findings serve to explain the unpredictable ‘design trajectory’ in terms of a composition of distinct design episodes. While this notion of unpredictability in design is not new in design theory, this thesis’ expositions of a number of characteristics are claimed as original. These characteristics include the notion of discrete and distinct episodes typified by a period of stability and commitment to a design option; description of the part that breakdowns play in bounding design episodes; the relationship between episodes and designer self-selection of roles; the effects of various kinds of bias in design (particularly paradigm and perspective biases); the means by which designers use negotiation with various stakeholders to steer the design trajectory toward familiar ground (in the form of known problem types and solution archetypes); the role that a range of other forces in the designer’s wider social context have on design outcomes; the architect’s relationship with methodology; the architect’s apparently *ad hoc* approach to assembling a personal design process *in situ*. All of these findings have direct relevance to the theory and practice of design in the software fabric.

On the ways in which expert software designers arrange their context so as to support their tacit designing, this contribution results from a selection of findings that portray the architect as negotiator—of scope, requirements, system structure, and other aspects of the design engagement. Negotiation of problems with clients—in order to make them more like problems for which the architect-negotiator has known and trusted solution archetypes—evidences the architect’s potential to influence goals and goal-setting processes. From a design-theoretic perspective, this behaviour can also be interpreted as the architect arranging his or her context, both explicitly and implicitly, with the personal objective of making the design engagement more familiar and therefore less exposed to risk. In simple terms, experienced software designers make the presenting problem look like one they have solved before. This is entirely consistent with other accounts of

situated activity. Suchman's account of photocopier use (1987) confirms how situation cannot be divorced from theories of cognition, and her account of the Trukese navigators confirms the viability of purposeful activity in the absence of formal abstract models or plans. Johnston's (1999) account of situated activity in manufacturing settings confirms that managers structure the environment of operations in order that 'management goals emerge through the robust interaction of simple reactive systems with suitably structured environments' (p. 141). This contrasts dramatically with the dominant position in the manufacturing and operations management domains that asserts that desired system outcomes are achieved through the automated production and implementation of plans or schedules derived by formal manipulations upon abstract representations of the world. The key theoretical contribution is to be found in the rich descriptions of the ways that expert software architects effect this environmental change before and during design activity. There are three dimensions of influence—externally to the client, internally to their team, and personally through reflection. Externally (to their clients) software architects shape their context through problem, requirements and solution negotiation. Internally (to their team) software architects shape their context through collaboration (particularly to achieve design adoption and evaluation) and via the assembly of a design process assembled for the team and its capabilities. Personally (to themselves) software architects shape their context through continuous reflection, which manifests as a sense of software aesthetic. Thus the expert software architect makes his or her practice situated through exertion of influence. This research did not determine the degree to which the individual designer's influence in all three dimensions results in the success or otherwise of a design engagement, but it seems intuitive that higher degrees of controlled influence would correlate with better design outcomes. The *situated* software architect is therefore primarily an influencer, able to change the context so as to make it familiar, which in turn sets the scene for personal, tacit design skills to come to the fore. To the less experienced practitioner, software architecture is the assembly of elements of the solution. To the journeyman, it is assembly of elements of the solution and the shared design process that enables a design capability to evolve. The most experienced designers assemble the holistic context of the design engagement to allow design activity to be performed tacitly. This is how an expert software designer makes the complex and multi-dimensional act of software design look easy.

### 10.5.3 Contribution to software engineering

The thesis' main contribution to software engineering concerns the establishment of a

researched basis for re-conceptualising the management of design activity. From a situated perspective, management is a design process rather than a control process. The participants accounted for the pragmatic design process as one which involves analysis of the environment in which the system is to act, design of the simplest system which will achieve the desired goals in interaction with a properly structured or restructured environment, followed by testing and tweaking. Johnston claims that this may well be the way most successful systems are actually designed in practice, but ‘the lack of theoretical respectability for this approach can lead to it being derided as ‘muddling through’”(1999, p. 146). Intuitively, we expect such a bottom-up system design process to be robust by virtue of its reliance on stable low-level interactions, and also to be entirely compatible with an evolutionary approach to change. This contrasts with the expensive and ossifying effects on design activity often associated with conservative engineering and method-centred approaches.

The thesis’ findings have application in making the principles of how software designers act to restructure environments more explicit in such a way that the environment shares the cognitive burden with the target system. This amounts to a call for the development of ‘principled characterisations of interaction between agents and their environments’ to guide design (Agre 1995). Designers can restructure their environments in the following ways. At the macro level, recognition of the iterative and episodic nature of design activity can be used to separate and promote the creative part of software design. A recognition of the way design activity occurs—via Simon’s Generate/Test cycle and this research’s depiction of design episode—can alert the practitioner to poorly formed or inappropriately used methods, processes and management practices that would constrain or conflict with episodic design. This thesis has not pursued further translation of the findings into either a methodology of its own or to suggested practices for practitioners. However, the general direction of some of these implications is not difficult to identify. With respect to design activity, these might include techniques for taking control of the design episode by recognising and explicating situation, recognising breakdowns during design activity, articulating an aesthetic sense for the particular team and situation, and stimulating intuitive leaps or new design episodes by controlling perspective change. With respect to the engagement, these might include techniques for developing a critical and radical sense of judgement and assessment, and for relating this to designing and design activity. Many of the practices of the Agile methods would have currency in this application of these research findings. For example, the use of short, goal-oriented user stories as a planning mechanism dovetails easily with setting goals and scope of a series of design episodes.

Developing a software aesthetic could possibly be driven by the experience of pair programming. Other associations should be self-evident to the Agile practitioner. There may be some value in completing this exercise on the basis that any results in the form of concrete applications and techniques would be derived from this research's grounded-theoretical foundation.

## 10.6 Conclusion

This thesis commenced by posing a question about the practice of software design. The question was refined to a particular class of professional designers who could demonstrate significant experience in the architecture of object-oriented software products, frameworks and systems. The author's position—that rational planning and design models have failed to accurately reflect software design activity—was declared and argued. A review of design theory and philosophy identified a broad philosophical and epistemological movement away from rationalism and toward theories based on constructivist philosophy. One of these—the situated action model—was selected as an alternative model of design activity.

A thematic review of the literature revealed a foundation of design research based on positivism, interpretivism and ethnography. Most of this research concerned design in non-software fabrics. In the software domain, most design-focussed research concerns the operation of teams engaged in design activity, models of collaboration, decision-making and learning within teams, and the organisational impacts that result from design-focussed projects and initiatives. Where research addresses the individual software designer, it generally focuses on use of methods and related pedagogical themes rather than on eliciting descriptions and models of the individual designer's creative and evaluative design activity and how this activity supports the activity of software design.

An interpretivist research method was then designed which would explore the various phenomena evident in expert's descriptions of software design practice. A framework identifying characteristics of situated practice was assembled to structure the interviews. Twenty-four expert object-oriented software architects were recruited, screened and interviewed, resulting in 33.5 hours of interviews and 342 pages of transcript (Appendix D). The qualitative analysis yielded 5 top-level topic maps, 258 sub-topics and relationships (Appendix F), and 25 distinct narratives (Chapter Eight) on the nature of software design as related by the participants. This set of narratives on the practice of software design constitutes the primary research output and satisfies the research aim.

Of the twenty-five narratives, many represent depictions of different mechanisms and techniques used by the architects to manipulate their context so as to achieve a higher probability of a successful design outcome. For example, when the architect negotiates problems, requirements, constraints and solutions, he does so to make the problem and context both familiar and comfortable. Recognising that situation allows tacit design skills to come to the fore, this contextual manipulation enables the designer to move from predominantly using *explicit* design skills to using *tacit* design skills. Thus the architect-negotiator arranges the workshop furniture and tools in preparation for immersive, tacit designing. The situated software architect's two primary skills are therefore designing and influencing. Arguably, this over-arching observation reveals the true nature of situated software design—the architect responds to the situation by arranging what can be arranged in order to transition into a tacit design mode. In the quote at the head of the previous chapter, Jones expresses the designer's role as 'designing the meta-process, *designing the situation* so that designing collaboratively is possible, so that the interaction flows'. (Jones 1988, p. 225). Apart from Suchman, this relationship between designer, situation and context is well expressed by Johnson (1999) who commented on 'the principles of designing systems for particular environments and of the restructuring of environments in such a way that the environment shares the cognitive burden with the focal system' (p. 146).

The hypothesis proposed situatedness as a model of the participant's depictions of the personal process and activity of software design. A partial fit was found and argued for on the basis of the models expressed by the qualitative analysis. The inability of a rational model of activity to comprehensively explain software design activity was confirmed. The findings did not support adoption of a situated model over a rational model—rather, they explained how these models are complementary and where and how the situated and rational models respectively hold. The software design professional acts in a situated fashion when creating, and in a rational fashion when evaluating. The detail of this summary statement can be found in the findings (Chapter Eight).

The strength of these research results depend primarily on two factors—the content and solidity of the participant's accounts of designing, and the author's choice of interpretation of same. The strongest content came from about one third of the twenty-four participants. Every attempt has been made to accurately reflect the character of these individuals. For the author (and hopefully for the reader) the personas of *Breuer*, *Utzon*, *Le Corbusier*, *Gropius*, *Sullivan*, *Mackintosh* and their counterparts build through the second half of the thesis to a

point where their positions on various design topics become unsurprising and even anticipated. This thesis portrays its participants as actors engaged in a discourse on design practice, with the effect that the convergence of their opinions becomes authoritative.

Ultimately, as with much design research, this thesis is a work of the author's interpretation. The research process progresses from perceptions and early understandings through collection and analysis of data to solidifying concepts and relationships via both iterative cycles and successive layers of interpretation. Iteration serves the refinement of notions to clear and well-defined concepts or relationships. Layering serves the organisation of complexity in this conceptual model. Every action in the analysis process, whether it be the definition of a phenomenon or the drawing of relationships between two such conceptual nodes in a topic map, is based on an interpretation. As such, this thesis is a form of argument that resembles a building, where each 'block' is selected, presented, justified, anchored to epistemology and the research paradigm, and then laid on the previous ones, to provide the foundation for the next. The participant's accounts which evidence these conceptual blocks, and the process by which this analytical structure has been built, have both been kept transparent to the degree that the thesis medium allows. It is up to the reader to determine whether he or she would build the same structure, or if not, where the two would differ.

In the final reflective self-analysis, the researcher starts to recognise the behaviours portrayed by the participants in his or her own research practice. When aspects of the 'research trajectory' start to align with what the analysis has referred to as the 'design trajectory', a Heideggerian truth about design dawns. In a world inanimate of objects, design is ubiquitous. We are all designers. And the continual design and redesign of our environment is our inescapable destiny.

## Appendix A: Glossary

This glossary presents definitions of terms introduced in the qualitative analysis to describe a particular emergent phenomenon.

<i>a</i> -rationality	The collective behaviours of software designers that can be classified as <i>not</i> being rational—that is, behaviours that are based on a pragmatic, radical or critical philosophical position.
Archetype	A high-level heterogeneous solution space pattern, used by an experienced software architect during the early states of a design engagement, to reason about the potential structure of a solution.
Conceptual design	The design of a software system or product expressed in terms of concepts or high-level abstractions.
Design act	The activity, performed by a software architect, act that puts a software artefact where one did not previously exist.
Design engagement	The terms and characteristics of a contract between a client and a software architect, that signals an intention to work together to produce a design for a software system or product.
Design episode	A phenomenon that emerged from the analysis of the participant's accounts of designing that accounts for how a design trajectory moves forward in phases, or distinct episodes, separated in time, each representing a period of commitment to the design as it stands, and demarcated by breakdown events. Each 'design episode' represents the currently preferred design option, and is stable until proven flawed.
Design method	The part of an 'external method' that purports to assist in the creation of a software architecture or artefact.
Emergence	A phenomenon of certain types of design activity whereby features of the solution emerge as a consequence of the designer's manipulation of the solution elements in context.
External method	A published or shared method intended to guide the design process.
Methodology, method	The analysis of the principles of methods, rules, and postulates employed by a discipline, as well as a particular procedure, or set of procedures. Also implies the rationale and the philosophical assumptions that underlie a particular design process or approach.
Pattern	A general repeatable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template

	for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.
Personal method	A synonym for 'Personal process'.
Personal pattern	A phenomenon that emerged from the analysis of the participant's accounts of designing which describes an expert software designer's conception of a design or code pattern. A personal pattern may be the designer's interpretation of a published and possibly widely known and used design pattern, or it may be a knowledge fragment specific to the individual.
Perspective	The choice of a context or a reference (or the result of this choice) from which to sense, categorize, measure or codify experience, cohesively forming a coherent belief, typically for comparing with another. One may further recognize a number of subtly distinctive meanings, close to those of paradigm, point of view, reality tunnel, or weltanschauung. To choose a perspective is to choose a value system and, unavoidably, an associated belief system.
Private method	A phenomenon that emerged from the analysis of the participant's accounts of designing in which software designers distinguish between the external or published method (which they and their project may, on the surface, be following) and the personal method which the individual designer believes he or she is following.
Public method	A phenomenon that emerged from the analysis of the participant's accounts of designing in which  Aka 'external method'
Routine design	A form of design which yields convincingly to rational approaches. Routine design is often contrasted with conceptual design which is defined as requiring a higher degree of interpretation.



## Appendix B: Ethics Clearance



18 April 2001

Assoc. Prof. Christine Miggins  
School of Computer Science & Software Engineering  
Caulfield Campus

Mr. Paul Raymond Taylor  
9 Combarton Street  
Box Hill 3128


### **Project 2000/469 - Architecture evolution and transfer in object-oriented software projects**

Thank you for the information provided relating to the changes as requested by the Standing Committee on Ethics in Research Involving Humans.

This is to advise that the amendments have been approved subject to the following:

- Please provide the Committee with a copy of the revised consent form.

and the project may proceed according to the approval as given on 14 November 2000.

  
Ann Michael  
Human Ethics Officer  
Standing Committee on Ethics  
in Research Involving Humans



RESEARCH GRANTS AND  
ETHICS BRANCH  
PO Box 5A  
Monash University  
Victoria 3800, Australia  
Telephone: +61 3 9905 3022  
Facsimile: +61 3 9905 3831  
E-mail: [ofrest@adm.monash.edu.au](mailto:ofrest@adm.monash.edu.au)

[www.monash.edu.au](http://www.monash.edu.au)  
ABN: 12 377 614 012



15 November 2000

Assoc. Prof. Christine Miggins  
School of Computer Science  
& Software Engineering  
Caulfield Campus

Mr. Paul Raymond  
9 Combarton Street  
Box Hill 3128

**Re: Project 2000/469 - Architecture evolution and transfer in object-oriented software projects**

The above submission was approved by the Standing Committee on Ethics in Research Involving Humans at meeting B7/2000 on 14 November 2000 provided that the following matters are satisfactorily addressed:

- Sects. 10(g) and 10(h). It is said that "early participants" will be asked to nominate "suitable professional colleagues". Confirm that there will be no "dependent or unequal relationship" between these categories. Would any early participant be in a position to put pressure on a colleague?
- Sect. 18: Chief Investigator's signature is required.
- If the Informed Consent Form is to accompany the Explanatory Statement Sect. 11(d) should be amended accordingly, as the consent covers more than just interviews.

The project is approved as submitted for a three year period and this approval is only valid whilst you hold a position at Monash University. You should notify the Committee immediately of any serious or unexpected adverse effects on participants or unforeseen events that might affect continued ethical acceptability of the project. Changes to the existing protocol require the submission and approval of an amendment. Substantial variations may require a new application. Please quote the project number above in any further correspondence and include it in the complaints clause:

*Should you have any complaint concerning the manner in which this research (project number.....) is conducted, please do not hesitate to contact The Standing Committee on Ethics in Research Involving Humans at the following address:*

*The Secretary  
The Standing Committee on Ethics in Research Involving Humans  
PO Box No 3A  
Monash University  
Victoria 3800  
Telephone (03) 9905 2052 Fax (03) 9905 1420  
Email: SCERH@adm.monash.edu.au*

Continued approval of this project is dependent on the submission of annual progress reports and a termination report. Please ensure that the Committee is provided with a report annually, at the conclusion of the project and if the project is discontinued before the expected date of completion. The report form is available at <http://www.monash.edu.au/resgrant/human-ethics/forms-reports/index.html>.



RESEARCH GRANTS AND  
ETHICS BRANCH  
PO Box 3A  
Monash University  
Victoria 3800, Australia  
Telephone: +61 3 9905 5012  
Facsimile: +61 3 9905 3831  
Email: [offres@adm.monash.edu.au](mailto:offres@adm.monash.edu.au)  
[www.monash.edu.au](http://www.monash.edu.au)  
ABN: 12 377 614 012

- 2 -

The Chief Investigators of approved projects are responsible for the storage and retention of original data pertaining to a project for a minimum period of five years. You are requested to comply with this requirement.



Ann Michael  
Human Ethics Officer  
Standing Committee on Ethics  
In Research Involving Humans



18 June 2001

Assoc. Prof. Christine Miggins  
School of Computer Science & Software Engineering  
Caulfield Campus

Mr. Paul Raymond Taylor  
9 Combarton Street  
Box Hill 3128

**Project 2000/469 - Architecture evolution and transfer in object-oriented software projects**

Thank you for the copy of the revised consent form and other information provided as requested by the Standing Committee on Ethics in Research Involving Humans.

This is to advise that the amendments have been approved and the project may proceed according to the approval as given on 14 November 2000.



Ann Michael  
Human Ethics Officer  
Standing Committee on Ethics  
in Research Involving Humans



RESEARCH GRANTS AND  
ETHICS BRANCH  
PO Box 3A  
Monash University  
Victoria 3900, Australia  
Telephone: +61 3 9905 3012  
Facsimile: +61 3 9905 3831  
E-mail: [offices@adms.monash.edu.au](mailto:offices@adms.monash.edu.au)  
[www.monash.edu.au](http://www.monash.edu.au)  
ABN: 12 577 614 912

**SIGNATURES****ALL SECTIONS MUST BE COMPLETED****18. STATUTORY PRIVACY PROTECTION**

*If the data used are held or to be collected by a Commonwealth Agency (see Question 14(c)) AND collection will or might enable identification of any individual (see Question 14(f)), then the Privacy Act (1988) applies. Indicate below whether or not the Privacy Act applies.*

The Privacy Act applies to the proposed data collection

☐

The Privacy Act does not apply to the proposed data collection

☒

Signature of Chief Investigator/Supervisor

Date

**19. DECLARATION**

I/We, the undersigned, accept responsibility for the conduct of the research detailed above in accordance with the principles outlined in the National Statement and any other conditions required by SCERH. If any changes to the protocol are proposed after the approval of the Committee has been obtained then SCERH will be informed immediately.

**Signature of Chief Investigator/or Supervisor**Name: Christine Miggins (please print)Signature:  Date: 23/10/00**Signature/s of Co-Investigator(s)/Student Researcher**1. Name: Paul Taylor (please print)Signature:  Date: 23/10/00

2. Name: \_\_\_\_\_ (please print)

Signature: \_\_\_\_\_ Date: \_\_\_\_\_

**Signature of Head of Department/Acting Head of Department/Director of Centre**

I certify that my department takes responsibility for this research.

Name: Christine Miggins (please print)Signature:  Date: 23/10/00Section: School of Computer Science & Software Engineering

SCERH application form updated 1/9/2000 page 17

# Appendix C: Interview Pack

## Informed Consent Form

### **Informed Consent Form**

**Project Title:** Architecture Evolution and Transfer in Object-Oriented Software Projects (2000-469)

This form allows us to record your consent to participate in this study. Consent forms are standard ethical research practice and a requirement of the university.

#### **1. Participation in this Study**

By signing the consent clause below, you agree to take part, and to be interviewed and recorded.

I agree to take part in the above Monash University research project. I have had the project explained to me, and I have read the Explanatory Statement, which I may keep for my records. I understand that agreeing to take part means that I am willing to:

- complete a survey asking me about my expertise and professional practices in the design and management of software systems and architectures,
- be interviewed by the researcher,
- allow the interview to be recorded.

I understand that any information I provide will be treated as highly confidential, and that no information that could lead to the identification of any individual, product or company will be disclosed in any publication or to any other party,

AND

I understand that I will be given a transcript of my interview for correction and my approval before it is included in the research thesis or any publication.

I also understand that my participation is voluntary, that I can choose not to participate in part or all of the project, and that I can withdraw at any stage without being penalised or disadvantaged in any way.

*Name:* .....  
(please print)

*Signature:* ..... *Date:*  
.....

#### **2. Further use of Data**

The survey and interview has been designed for the aims of the above research project. However, it may be desirable to re-use the data for further analysis by other projects in the future. By signing the consent clause below you agree that the data collected (with your name and any other names removed) may be passed

onto other researchers for further analysis.

I understand that the data collected from my survey and interview may be passed on to other research projects in the future, provided:

- these projects have university ethics approval, and
- my name, and all other names (persons, products, companies) and contact information is removed before it is passed on.

*Name:* .....  
(please print)

*Signature:* ..... *Date:* .....

This form will be kept by the researchers with transcripts of the interview.  
Thankyou for your participation.

## Invitation to Participate

### Invitation to participate in research project (2000/469)

1 June 2001

**Project title:** The Situated Software Architect — Architecture Evolution and Transfer in Object-Oriented Software Projects.

**Chief Investigator:** Associate Professor Christine Mingins.

**Student Researcher:** Paul Taylor.

Thank-you for your interest in this research project, the centre-piece of my PhD research in the Department of Computer Science and Software Engineering (Monash University) under the supervision of Associate Professor Christine Mingins and Professor Richard Mitchell.

The following summary should provide everything you need to know about the project and your involvement:

#### Research Aims

The aim of this research is to understand what approaches, methods and techniques are used by software architects in industry to design, manage and evolve complex object-oriented software architectures and systems. The interview sessions are designed to elicit architect's experiences and opinions on matters of software architecture and design. The research will take a qualitative, interpretivist approach, by focusing on the factors that shape software and system design in this situated context. This information will be used to identify ways in which OO architectures are designed and practically managed. It will also give software engineers and methodologists a current picture of how software architecture and design in business and industrial contexts is understood, and enacted.

#### Selection of participants

Participants in this study must be experienced software architects and designers who have worked on at least three object-oriented systems or software projects in industry. Participants will have worked in an architect role providing technical direction to other software developers and engineers. To ensure that the study's findings are not compromised, it is important that each participant can show evidence of this experience. A Preliminary Survey helps us to understand how relevant your particular experience is to the study's goals, and to prepare for the interview.

#### Preliminary Surveys

By this stage, you should have received a Preliminary Survey by e-mail (a Word 97 document attachment). Please complete the survey (in electronic form) and return it to [ptaylor@csse.monash.edu.au](mailto:ptaylor@csse.monash.edu.au). The survey assesses broad exposure to object-oriented software architecture and design, and will allow us to prepare for, and get best value from the interview session. If we both decide to proceed, an interview will then be scheduled at your convenience.

#### Interviews

The interview will provide the opportunity to interact and discuss some areas of your experience and expertise more deeply. Your interview will take no more than 90 minutes and will be scheduled at a mutually convenient time and place. The venue will be quiet and private. With your permission, interviews will be audio-taped and subsequently transcribed into an interview record that will be e-mailed back to you so that you can validate the transcript, if you wish.



**Confidentiality**

The interviews will discuss generalised techniques and practice, and you will not be asked to state or discuss the names of companies, products or individuals at any time. If you do, these names will be kept completely confidential. When you provide corrections, clarifications or indicate agreement with the interview notes, the audio recording will be erased. Access to all collected data (the interview tapes while they exist, interview notes and transcriptions) to tapes will be restricted to the Student Researcher (Paul Taylor) and the Chief Researcher (Associate Professor Mingins). No information that could identify the participant will be present in the interview transcript, its filename, or its properties (participants are allocated a pseudonym to facilitate this).

**Follow-up interview**

In some cases, there may be some value in a follow-up interview, in which specific points are clarified or further detail discussed. If you are invited to a second interview, again at a time and place that suits you, you may of course refuse without compromising your first interview's data.

**Access to data**

All collected data will be securely stored for 5 years as prescribed by university regulations. You will have an opportunity to indicate whether you agree to the (de-personalised) data from your interview being made available to other researchers in the future.

**Feedback**

You may indicate your interest in seeing copies of any of the reports, thesis sections or published papers that result from this research. If you do, electronic copies will be sent to you.

**Incentive**

We cannot offer monetary or any commercial incentive to reward your participation.

**Withdrawal**

You may withdraw from participation at any time simply by simply informing me (Paul Taylor). You are also free to avoid answering any survey or interview question which you believe is inappropriate or professionally intrusive—such a refusal will not invalidate your other answers nor affect your future involvement in this or any similar Monash University study

If you have any queries, please contact either me (Paul Taylor) on (03) 9617 0202, 0404 819 005 or FAX (03) 9621 1951 (or send e-mail to [ptaylor@csse.monash.edu.au](mailto:ptaylor@csse.monash.edu.au)) or Associate Professor Mingins on (03) 9903 2078 ([cmingins@csse.monash.edu.au](mailto:cmingins@csse.monash.edu.au)). Should you have any complaint concerning the manner in which this research (2000/469) is conducted, please do not hesitate to contact The Standing Committee on Ethics in Research Involving Humans at the following address:

The Secretary, The Standing Committee on Ethics in Research Involving Humans, PO Box 3A, Monash University, Victoria 3800. Telephone (03) 9905 2052 Fax (03) 9905 1420  
Email: [SCERH@adm.monash.edu.au](mailto:SCERH@adm.monash.edu.au)

Thankyou for your interest in this project. Paul Taylor.

## Interview outline

### The Situated Software Architect

Architecture Evolution and Transfer in Object-Oriented Software Projects

Principal Researcher: Christine Miggins

Student Researcher: Paul Taylor

Associate Supervisor: Richard Mitchell

Cleared by Monash University Standing Committee for Ethical Research into Humans—project number 2000-469



### About Software Design Practice

(~20 mins ? +20)

1	Software architecture	What do you understand 'software architecture' to be?
2	Software design	What do you understand 'software design' to be? How does this differ from software architecture?
3	Software methodology	What do you understand 'software methodology' to be? How does it relate to software architecture and software design?
4	Perceptions of success	Describe your most successful piece of software architecture and design. Describe the circumstances in which you did this work. What made this architecture or system successful?
5	Collaborative design capability	What things most significantly contribute to a team's ability to deliver high quality software architecture and design?  What things compromise it?
6	Contextual effects	How do contextual (non-technical) factors impact the shape of a software system's architecture or design?
7	Compromise	What factors most compromise system architectures developed in business and industry contexts? What causes or drives these compromises?

### Performing Software Design

(~25 mins ? +45)

8	Role definition	What distinguishes between a skilled software architect and a skilled programmer?
9	Conceptual vs. detailed design	Is there a difference between conceptual and detailed software design?
10	Abstraction skills	How important are abstraction skills to software design?  How does methodology help you in forming abstractions?  How does your experience with past solutions help you in forming abstractions?
11	Creativity	What role does creativity play in software design and architecture?

12	Assumption validation	How do you validate or cross-check your assumptions when designing?
13	Generators	How do you go about producing the initial architecture of a system? Describe how you have approached this in the past.
14	Option assessment	Do you ever develop several candidate solutions and then select the most suitable one? Why or why not?
15	Synthesis	How do you go about composing or synthesising solution fragments or components into an emerging design?
16	Design feedback	In the design of software architecture, what sort of feedback is important to you?  How do you incorporate feedback into the way you design?
17	Architecture preservation	What sorts of things do you do to preserve critical architectural structure?  At the system architecture level?  At the class, package or sub-system level?
<b>Design and Knowledge</b>		<b>(~15 mins ? +60)</b>
18	Design process	Describe the process you take to the design of a system's software architecture.  How does this relate to the design processes espoused by popular methodologies?
19	Repeatability	In your experience, is the design of viable software architectures a repeatable process? Do you think it should be?
20	Planning	What sort of planning works best for periods of system and software design activity?
21	Knowledge media	Where should design knowledge be embedded for maximum benefit? in the product (the software architecture), <ul style="list-style-type: none"><li>• in the process,</li><li>• in the heads of people,</li><li>• somewhere else (a repository, for example)?</li><li>• some combination of the above? What combination would you recommend and why?</li><li>•</li></ul>
22	Patterns	How much do you draw upon published architectural and design patterns?
23	Learning design	How did you learn to design software architectures and systems? How would you teach someone else?
24	Metaphor	How relevant do you think the engineering metaphor is to what you do?

<b>Other relevant experiences</b>		<b>(~5 minutes ? +65)</b>
25	Etc	Are there any other relevant stories or experiences about your experience with OO software architectures and designs that come to mind?
26	Referrals	Can you suggest others with a background similar to yours who would make valuable participants in this study?

## Preliminary Survey Form

### Preliminary Survey

Project Title: Architecture Evolution and Transfer in Object-Oriented Software Projects (2000-469)

Overtyping the free format text fields, and type an 'X' next to (or over) the most appropriate box symbols.

- 1 Name: .....
- 2 Phone(s): .....
- 3 E-mail: .....

### Architectural experience

- 4 In what architectural or technical leadership roles have you worked? Architect ☐  
 Technical Manager ☐  
 Project Manager ☐  
 Consultant ☐  
 Lead Software Engineer ☐  
 Other title(s): .....
- 5 Approximately how many years in total have you worked in these positions? < 12 months ☐  
 1-2 years ☐  
 2-5 years ☐  
 5 or more years ☐
- 6 How many distinct object-oriented software systems or products have you worked on during this time? 1 ☐ 2 ☐  
 3-5 ☐ More than 5 ☐
- 7 How large was the largest system you worked on in this time? <50 ☐ 50-100 ☐  
 100-250 ☐ 250-500 ☐  
 Indicate the number of manually 500-1000 ☐ 1000-2000 ☐  
 designed and coded classes. > 2000 ☐ Please specify the number .....
- 8 What is the longest period that you have spent working with any one object-oriented system or product? < 3 months ☐ 3-6 months ☐  
 6-12 months ☐ 12-24 months ☐  
 More than 2 years ☐

### Responsibilities

- 9 What aspects of software architecture have you held responsibility for during your involvement with these projects? (tick as many as are applicable) The entire product ☐  
 The software architecture ☐  
 A subsystem of the software architecture ☐  
 OO methodology ☐  
 Design and coding processes ☐  
 Code quality ☐  
 Other responsibilities: .....

- 10 What types of work did you perform in your role as architect? (tick as many as are applicable)
- Requirements and functional specification ☐  
 System architecture and design ☐  
 Domain modeling ☐  
 Software architecture and design ☐  
 Object modeling ☐  
 Data or database modeling ☐  
 Coding ☐  
 Code reviewing ☐  
 System testing ☐  
 Debugging ☐  
 Other activities: .....

### Specific object technologies

- 11 What object-oriented software technologies have you used on these projects?

Analysis/Design Methods	Programming Languages	Object databases and persistence
UML <input type="checkbox"/>	Java <input type="checkbox"/>	ObjectStore <input type="checkbox"/>
Open <input type="checkbox"/>	C++ <input type="checkbox"/>	Versant <input type="checkbox"/>
FUSION <input type="checkbox"/>	VisualBasic <input type="checkbox"/>	Persistence <input type="checkbox"/>
MeNtOr <input type="checkbox"/>	Delphi <input type="checkbox"/>	
Others: .....	Others: .....	Others: .....
Class libraries and frameworks	Distributed Objects and Components	CASE
MFC <input type="checkbox"/>	COM/DCOM <input type="checkbox"/>	Rational ROSE <input type="checkbox"/>
Java libraries <input type="checkbox"/>	CORBA <input type="checkbox"/>	StP <input type="checkbox"/>
RogueWave <input type="checkbox"/>	J2EE/EJB <input type="checkbox"/>	System Architect
Others: .....	.net <input type="checkbox"/>	
	Others: .....	Others: .....

### Other comments

- 12 Anything else we should know to help us prepare for the interview?

.....

.....

.....

.....

Please e-mail this completed survey to [ptaylor@csse.monash.edu.au](mailto:ptaylor@csse.monash.edu.au) and you will be contacted to arrange an interview time.

# Appendix D: Participant Profile

## Introduction

The preliminary survey results comprise a profile of the participants that provides important background to the reading of the analysis.

## Interview Process

Between June 2000 and July 2002 24 software architects were interviewed in Melbourne (21), Sydney (2) and Canberra (1). These participants were recruited in one of three ways. Two participants responded to an invitation at the end of an article published in the Australian Computer Society's monthly magazine (Taylor 2001a). One participant responded to a verbal invitation made at the end of a paper presentation at the Australian Software Engineering Conference (Taylor 2001c). Nine participants responded to email invitations directly from the author. The remaining 11 participants were 'snowball' referees from these 12 initial participants.

Each participant completed a preliminary survey (reproduced in Appendix C) to screen their suitability, the results of which are presented in the next section. This survey was always intended to profile the participants rather than have any statistical significance. At the interview, each participant was asked to sign a consent form. All interviews followed the questions and topics in the framework described in Chapter 4, some more rigidly than others.

Every word of each interview was transcribed for subsequent analysis. Participants were allocated a pseudonym (the name of a 'famous' architect) which was used from that point forward in the handling of the transcript. Each participant's transcript was then sent back to him (with all information identifying any parties or products stripped) to provide an opportunity to comment. Table 4 summarises some metrics from the interview phase.

Characteristic	
Number of interviews	24
Total number of interview hours	33.5 hours
Average interview period	84 minutes
Transcript total word count	219,426 words
Transcript total page count	342 pages
Average transcript page count	14 pages

Table 4: Interview metrics.

Transcripts were imported into QSR's NVivo 2.0 (a qualitative analysis tool) for coding and analysis. Coding and analysis commenced after the seventh interview and continued in parallel with the remainder of the interviews. The complete analysis is presented in Chapter Six.

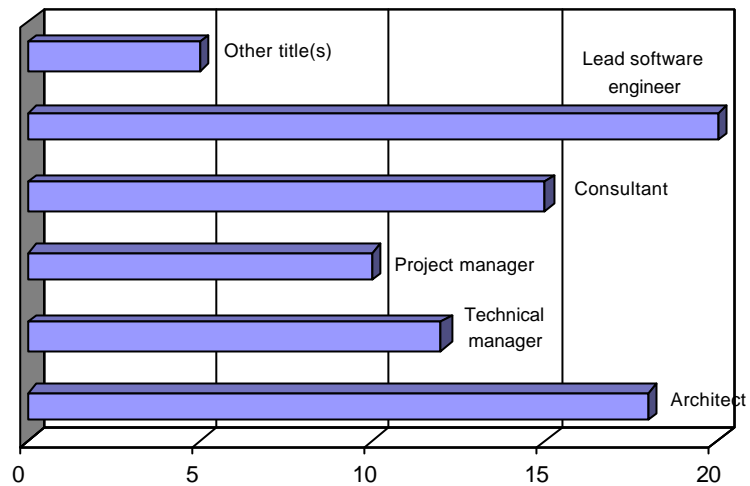
## Participant Profile

This section presents a simple frequency analysis of the participant's responses to the preliminary survey.

### *Professional roles*

Question 4—In what architectural or technical leadership roles have you worked?



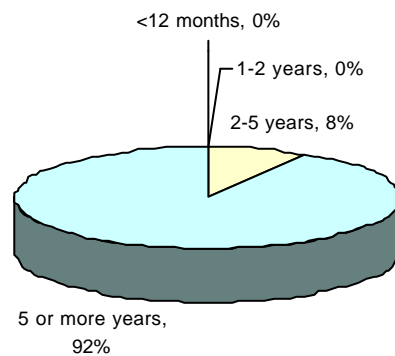


**Figure 24: Participant's professional roles.**

A participant's claim to having worked in an industry-sanctioned role or title is a rough indicator of their experience level. When asked about what architectural or technical leadership roles they had worked in, the participants identified 'lead software engineer' and 'architect' most often, with a spread across other technical roles.

#### *Years in roles*

Question 5—Approximately how many years in total have you worked in these positions?

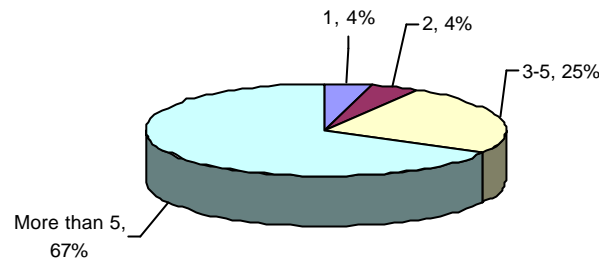


**Figure 25: Participant's years in roles.**

Time spent in industry-sanctioned roles is also an indicator of expertise. Most (92%) of the participants had held one or more lead technical roles for 5 or more years, the remainder 2 to 5 years.

*Number of systems*

Question 6—How many distinct object-oriented software systems or products have you worked on in this time?

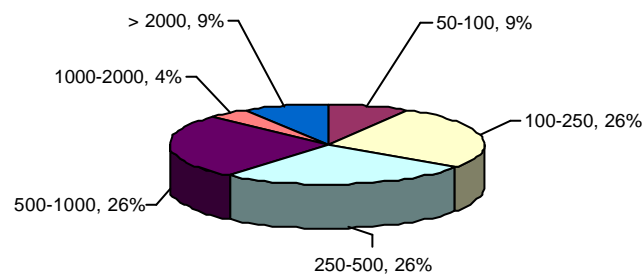


**Figure 26: Participant's number of object-oriented architectures or systems.**

Experience in software design increases with the number of projects and architectures an individual has worked with. Most (67%) participants had worked on more than 5 distinct object-oriented systems, with most of the remainder claiming 3 to 5 systems.

*Largest system*

Question 7—How large was the largest system you worked on in this time? Indicate the number of manually designed and coded classes.



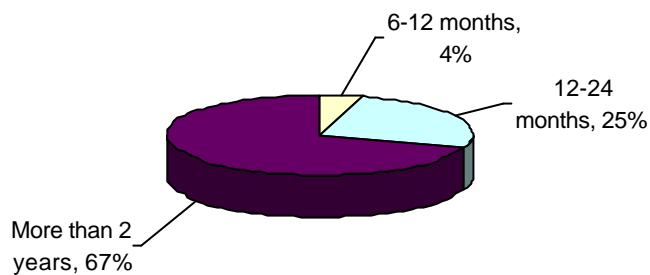
**Figure 27: Participant's largest architecture or system (classes).**

In general, large object-oriented systems involve more complexity, scale and coordination problems than small systems. Time spent with large systems is therefore an indicator of experience. A gross metric for object-oriented systems is the number of classes. Most of the participants were evenly distributed between systems of 100-250 classes (26%), 250-500 classes (26%) and 500-1000 classes (26%). Three participants reported having

worked with the architectures of very large systems (one reported 1000-2000 classes and two reported >2000 classes).

### *Longest time with one system*

Question 8—What is the longest period that you have spent working with any one object-oriented system or product?

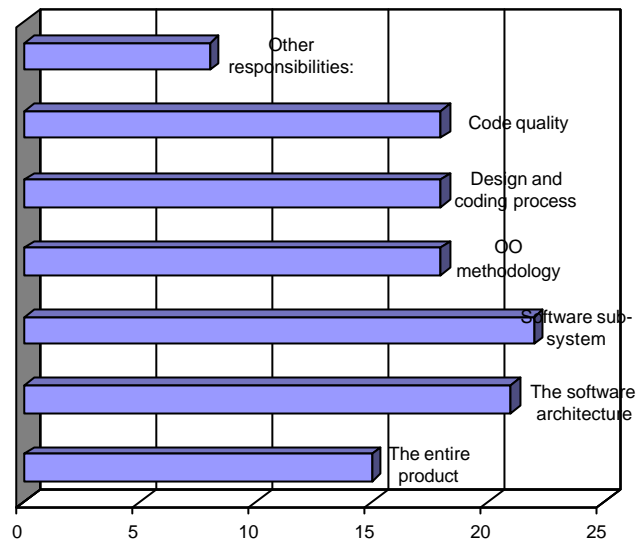


**Figure 28: Participant's longest time with one architecture or system.**

Some architectural insights are thought to emerge over the lifetime of an object-oriented system. Time spent working and evolving one system is therefore an indicator of exposure to the consequences of certain design choices. More than half of the participants (58%) reported having worked with an object-oriented system for more than 2 years. Most of the remainder reported 1 to 2 years.

### *Responsibility*

Question 9—What aspects of software architecture have you held responsibility for during your involvement with these projects?

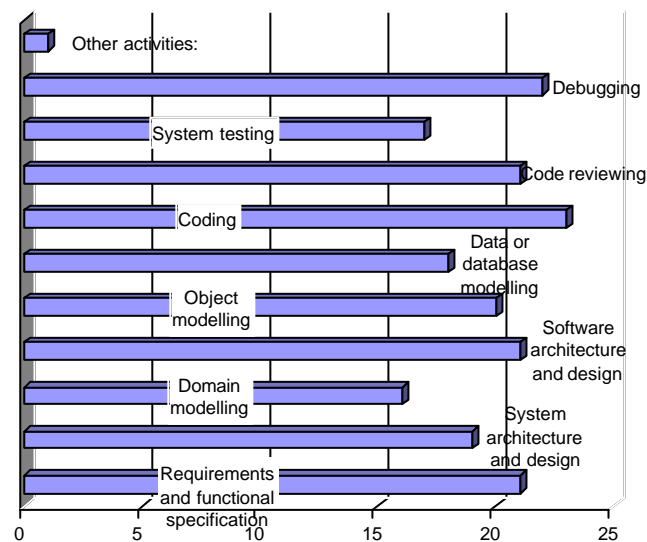


**Figure 29: Participant's architecture and design responsibilities held.**

The responsibilities of project architects and technical leaders can vary widely. This question assessed what responsibilities the participants had held across a typical system development lifecycle. The participants responded almost uniformly across the set of responsibilities.

#### *Points of engagement*

Question 10—What types of work did you perform in your role as architect?



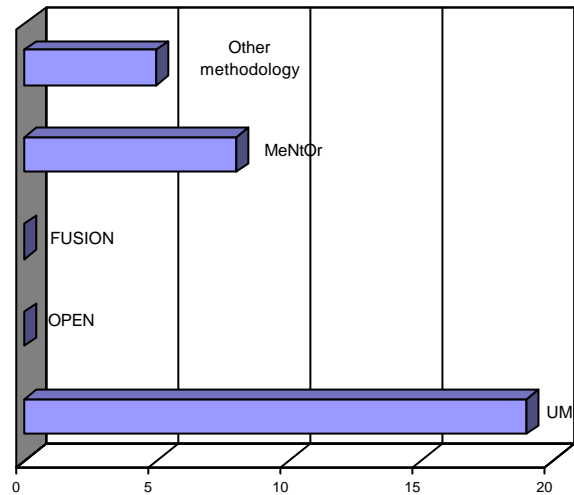
**Figure 30: Participant's points of engagement with the development process.**

Another indicator of experience is the kinds of tasks that the architect actually performed.

Again, the participants responded almost uniformly across the set of tasks.

### *Methodology use*

Question 11a—What object-oriented methodologies have you used on these projects?

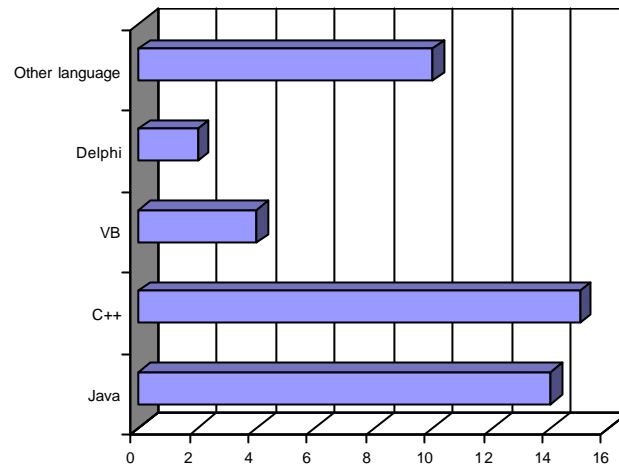


**Figure 31: Participant's use of methodology.**

Architects would be expected to have used methodologies in their design work. Most participants (79%) reported using UML. MeNtOr was also reported (33%). A small number of other methodologies (Booch, Schlaer/Mellor) were noted in the 'Other' category. In total, 88% of architects reported working with one or more methodologies.

### *OO languages*

Question 11b—What object-oriented languages have you used on these projects?

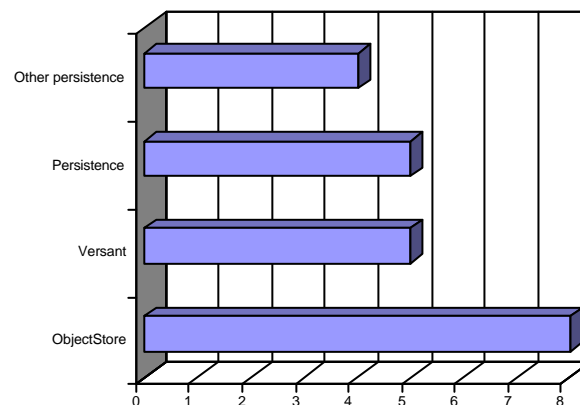


**Figure 32: Participant's use of object-oriented languages.**

Architects would be expected to be expert in at least one object-oriented language. The choice of language is driven by industry demands at the time of the survey. C++ (50%) and Java (54%) were predictable first choices. 71% of the architects reported using 2 or more object-oriented languages professionally.

### *Persistence*

Question 11c—What object-oriented persistence technologies have you used on these projects?



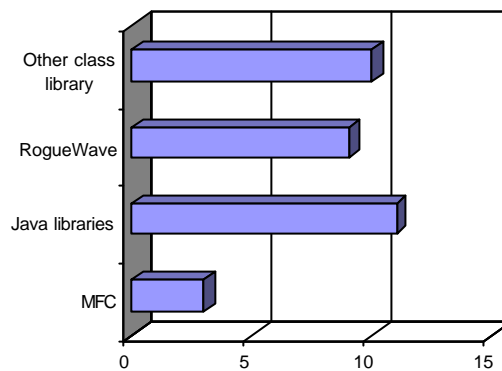
**Figure 33: Participant's use of persistence frameworks and products.**

Most object-oriented systems use a persistence service. At the time, the most popular

options were two object databases (Versant and ObjectStore), and an object-relational mapping product (Persistence). The participants reported relatively even exposure to these products. 96% of the architects reported using at least one persistence product.

### *Class libraries*

Question 11d—What object-oriented class libraries or frameworks have you used on these projects?

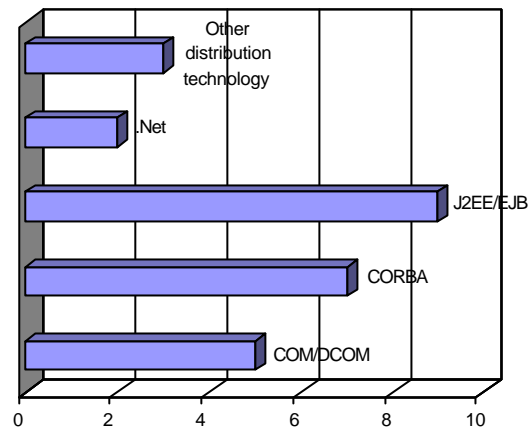


**Figure 34: Participant's use of object-oriented class libraries.**

Object-oriented languages rely heavily on class libraries. At the time, the three most popular options were Microsoft's MFC, the Java libraries and a C++ and Java product (RogueWave). Again, the participants reported relatively even exposure to these products.

### *Distribution technologies*

Question 11e—What object-oriented distribution technologies have you used on these projects?

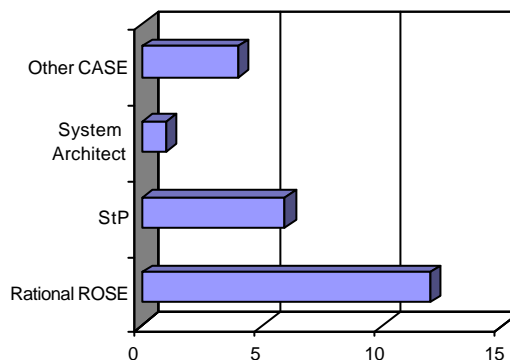


**Figure 35: Participant's use of object-oriented distribution technologies.**

Most object-oriented systems rely on some form of distribution. At the time, the options were COM/DCOM, .Net (which was immature at the time), CORBA and J2EE/EJB (also immature). Again, the participants reported relatively even exposure to these technologies. 71% of the architects reported using one or more distribution technologies.

### *CASE*

Question 11f—What Computer-Aided Software Engineering products have you used on these projects?



**Figure 36: Participant's use of CASE products.**

In conjunction with their use of methodologies, architects would be expected to have used Computer-Aided Software Engineering tools for their models. Half of the participants reported using Rational ROSE, with the remainder reporting Software thru Pictures, System Architect and others (including Modelmaker and Teamwork). 92% of the architects reported using CASE tools of some kind.



## Conclusion

The results of the preliminary survey paint a picture of a class of professional software technologists. Most have worked across the industry-accepted roles for software architects and with a few exceptions, have been responsible for (and have performed) design and development tasks across the typical object-oriented system's scope and lifecycle. They are very experienced in object-oriented technologies—almost all (92%) have 5 or more years of practice, two thirds (67%) have worked on 5 or more distinct object-oriented systems, nearly half (39%) have worked on medium sized systems (> 500 classes) with 3 having direct experience of large systems (> 1000 classes). More than half (67%) have worked with a specific object-oriented system for more than 2 years.

Their exposure to object-oriented technologies and products reflects availability and industry demand at the time of the interviews. Almost all have used a methodology (88%) with CASE support (92%), most are professionally multi-lingual (71%) with C++ and Java predominant, almost all have dealt with persistence issues and products in their architectures (96%), class libraries (96%) and distribution technologies (71%).

The 24 interviews that followed the preliminary survey instrument were transcribed and coded in parallel. The interview process was terminated when the categories appeared to be saturating.

## Appendix E: Example of Topic Analysis

Aesthetic

-----

< Listening to the code. >

Morris> So for me, doing emergent design is doing as much design as I am confident of, and then trying it out, and then modifying what happens based on the feedback that I get from the code. So things then get a little bit new-agey, because you start to talk about 'listening to the code' and listening to what the system is telling you.

-----

Sullivan> I know you were asking about architecture, but obviously architecture and design are related aren't they? It's different kind of ideas that are pointing at the same area. If you look at -- let's say -- a motor car or an aeroplane or something, you say "what a great design". When I was a student I had a Mark 7 Jag, I can still see in the Workshop Manual a picture of -- I think it might have been a cross section...no it wasn't... it was just a picture of the front of the engine... with it's two cam shafts ... oh, no, it had timing chains on it as well... And I mean, it has just left an indelible impression on me, what a thing of beauty! So, is that design? Where actually is the design?

-----

Cook> If I think back to the problem... to the project that I think was my most successful, one of the great things that we managed to do, David and I, in that, was to continually take problems, and re- evaluate them in the context of what we'd already done, and without fitting square pegs in round holes, actually say 'well OK, that is really just a special case of that'. 'Why doesn't that already handle that?' 'Oh look, if we re-factored that to use another buzz-word in a certain way it would still do all the stuff it was still doing before and it would handle that one special case. Great! Fantastic! Alright, that's good -- and to me that's an indication of elegant software, that is, with very little change, you can incorporate new concepts or new solutions or solutions to new problems.

-----

< Aesthetic universally involves reduction, optimisation, removal of duplication. >

Breuer> When I saw the ‘commissions’ package specifications -- I don’t think, you weren’t hear for that were you? I walked out of the meeting. I ended up saying... it was 124 pages -- please turn it into 15.

- - - - -

Ruskin> PRT> And also you can aim to reduce rather than expand. That is, the sign of the emergence of an elegant solution is that it starts to reduce in complexity and size -- and I take it that that’s how you regard object structure as well?

Architect> Indeed.

- - - - -

< Morris seems to suggest that software aesthetics are socially constructed in the context of a team. >

Morris> So one of the things I try to do is teach people more about the social stuff, and teach people to be able to express themselves in an assertive, but not aggressive way, and to respect other people’s opinion and to listen actively, and to do those sorts of things, so that we can come up with a better design overall. And when I go out to teach, yes I try to teach people to make technical decisions and listen to the code and appreciate the software aesthetics, but that is what they can do as individuals, but I try to place as much emphasis on teaching them to interact positively with the rest of the team, as I do on learning to appreciate the software aesthetics, because one is important in the solo environment and the other is critical for the team environment.

- - - - -

< A fundamental question is whether the kind of software aesthetic talked about by many of the architects is in any way objective. Can it be shared, or taught? Can it be expressed in patterns, for example? >

Cook> PRT> But presumably opinion converges, I mean you can see that in patterns that are being documented, you can see that in the work of the methodologists...

Architect> Oh, most definitely it does. And I think that’s because those people have experienced similar things and they all come to the agreement that yes, this is probably the most... I like to use the word ‘elegant’ more than simple. I think for people like me, I don’t know about other people, but I have... I look at something and I either have an intuitive feel that this is an elegant solution or it’s not. And when it’s elegant I think that’s... whether it’s considered simple by one person or not, it’s the right solution.

-----

< The software aesthetic is described as being essentially subjective and experiential. >

Utzon> It is very subjective, I know. And I mean I've taken basically the same architectural principles to one client and you know, almost been burned at the stake and taken them somewhere else and you know, lauded as a absolute genius. Still pretty much the same ideas, just a matter of whether or not there's that shared vision of what it is you're trying to bring across.

-----.< There is an analogy between occupying a physical designed space (an architect's building) and occupying a designed conceptual space (a software architect's system architecture). It is also related to a perception of 'goodness'. >

Utzon> PRT> Now you mentioned aesthetics before, obviously aesthetics plays a big part in real world architecture, and you could argue that real world architects are obsessed with aesthetic, what part do you think it plays in your work?

Architect> Umm, I think it's very important but extremely hard to define. Let's have a think... I have a... I put a lot of stock in Alexander's ideas about objective beauty, all the stuff about 'Turkish carpets and stuff like that. I think that, when people walk into a building that really works they know. And, not sure how you necessarily explain that. And I think people know that about software as well, is that when they start walking around a software design, they know.

PRT> There's a very interesting question there, it's to do with what it is that you're perceiving and what others are perceiving. I mean don't let me put words in your mouth but you might have worked with people who just don't perceive the same type of appeal.

Architect> Yeah... I have...

PRT> But you might have also worked with people with whom you share ideas with in a very fluid fashion. And you can see the same sorts of aesthetic appeal.

-----

< Johnson described a very tangible sense of code occupancy. >

Johnson> I have come up with designs that I don't want to ever go back to, I don't want to touch the code, I am scared to go there, I am scared to touch anything. I don't have the Quality Without A Name, the experience for me as the programmer is poor. I don't want to occupy that space. On my good designs, I love that space, I love going back, I can go there and I can read it, I can make changes and I know what I am doing.

-----

< Not everyone has a sense of software aesthetic. >

Morris> Architect> It is exactly in line with what I perceive, with an extra qualifier - we all have an innate sense of what makes us comfortable in a living space. We do not all have that sense of what is comfortable and liveable in software, inside a piece of software. Have you read Richard Gabriel's stuff on this? So I agree entirely with the sort of things he says, and that is one of another things that I see when I say some people aren't suited to software development -- they do not, and I don't believe that it is because they can't, or I am not sure if it is can't/won't or which axis it is on, but for whatever reason, they don't have that sense of what I call 'software aesthetics'.

And one of my challenges is to try to convey to people what is aesthetically pleasing code, what is going to work and what is not.

-----

< Unlike Morris, Lethaby thinks that a software aesthetic is not uncommon. >

Lethaby> I think generally there are enough programmers who have a good sense of those things. If you are in an environment where hardly any programmers have a sense of an aesthetic sense of their code, or any idea of elegance or efficiency, or design sense, then you would be in a difficult situation that you would probably need to come up with some sort of heuristic metrics for 'is this nice or is this not nice?' But fortunately, in environments that I have worked in at any rate, there have been always plenty of programmers who have had a very good sense of 'nice code versus nasty code' - and it shows that I am optimistic that this is a general condition, that is a human condition where you naturally respond to some kind of elegance in the code we are writing, and the machines we are making in our day to day routine, whatever it is.

-----

< Preservation of the essential theme, or a lack of its corruption, is an element of 'goodness'. >

Utzon> What are the measurements of aesthetics?

PRT> Is it the ability to share concepts? Or to jointly see concepts in the code with others, or in the structure of the system? The architecture of the system?

Architect> Yeah I guess so, I think it's just the ability to be able to express... it's about goodness you know, there's the idea that this architecture, the way this thing's designed is

‘good’. And obviously anything I do I’m going to try to make the best quality thing I can and when somebody else looks at that and goes ‘oh yeah, that is good’ and I understand that, and you can see the kinds of benefits that you’ve tried to build into it actually accruing, through other people actually taking that into design and not trying to corrupt it that as they go along...That’s actually a good yardstick now that I think about it - the less the designers and the programmers corrupt the theme that you’re actually trying to put together probably the better, the more successful your architecture is and the team that is that’s coming together.

-----

< Although it is subjective, Utzon thinks its principles can be distilled and documented. >.

PRT> So do you think you could -- I’m not going to ask you to do this because it could take along time but do you think you could express those principles of good architecture across domains?

Utzon > Yes. Have done. Just finished a document, actually. Yeah, I think so.

-----

< It is intuitive. >

Cook> I think for people like me, I don’t know about other people, but I have... I look at something and I either have an intuitive feel that this is an elegant solution or it’s not. And when it’s elegant I think that’s... whether it’s considered simple by one person or not, it’s the right solution.

-----

< Not being able to distil it into a set of statements or principles suggests to Cook that he does not understand it sufficiently well. >

Cook> PRT> So could you reduce that sense of elegance to a bunch of principles that you could then teach others? Or do you think this is just something that you develop over time?

Architect> I think it’s probably experience, but I think, again, given my previous criteria that probably means I don’t understand it at all well enough. I think also that it may not be that my definition of what seems like an elegant solution and someone else’s are all that different, but my willingness to compromise on certain issues may be different. I may consider some things to be more important than

others, and that will bias my decisions. But yeah, probably... the fact that I use the term 'elegance', that's quite an emotive word, and that probably indicates that I don't understand it.

-----

< The audience (or inhabitants) live with the consequences of good and not-so-good design. It is they who should arbitrate a software aesthetic. >

Cook> PRT> That's right. But we don't have that same sense of aesthetic in software architecture, that, if you like, 'real architects' do, but it seems to me that we do have something similar?

Architect> I think you do, because architect's peers, or the people who they perceive as judging, if you like, are, maybe other architects, also the general public, the people who are going to pay for the building and what-have-you. I think, speaking from a purely technical perspective, I think it's other developers, and its ability to withstand not sure withstand is such a good word to weather change I guess, its extensibility I guess. In my experience things that are elegant tend to be extensible. They may not... it may be completely different in day 5 to day 1, but the design allows you to change quite smoothly from one to the other.. So I don't know that it isn't necessarily... I don't know that it isn't any different, it's just who...

PRT> ...who the audience is?

Architect> who the audience is, audience is a good word, yes.

-----

< Cook describes a sceanrio in which he was able to repeatedly accommodate a raft of new requirements with his existing architecture by re- interpreting the problem to a degree. This genericity of his solution architecture and its demonstrated extensibility constituted an elegance. >

...take problems, and re-evaluate them in the context of what we'd already done, and (without fitting square pegs in round holes) actually say 'well OK, that is really just a special case of that'. 'Why doesn't that already handle that?' 'Oh look, if we re-factored that in a certain way it would still do all the stuff it was still doing before and it would handle that one special case. Great! Fantastic!

Cook> ...and to me that's an indication of elegant software, that is, with very little change, you can incorporate new concepts or new solutions or solutions to new problems.

-----

< Elegance is defined by usefulness. >

McLuhan> PRT> What about elegance?

Architect> That's very important...

PRT> Mathematicians, apparently, so I've read, have valued elegance in mathematical proofs and so forth? Do you perceive a degree of elegance?

Architect> I think that's the part of making everything fit properly. In an elegant solution, things fit better. It might not look elegant when you see the pieces on paper, but that's not the point. Paper is not where it happens, it happens inside the machine. Not in 2 dimensions, but in a thousand dimensions.

PRT> So when you're talking about things fitting together... in an elegant solution things fit together well, you're really talking about your model of how the objects are interacting in the executing solution as you see it in your mind's eye?

Architect> And the elegance of being able to say here I have an abstract data structure and these are its properties. And yet if I don't understand the structure of this object I've got, so what are its properties? Looking for the elegancies which are those properties -- that's the mathematical view of it. And so there's a degree of elegance associated with use and usefulness of it.

- - - - -Howard> Yes and particularly I think the more generic, and I will call it elegant the model, by almost definition the more abstract it is, the more flexible adaptive and all the rest

-----

< Clean, simple mappings between domain concepts and model concepts. >

Johnson> ...so pretty much everything in the UML and in the classes had a nice clean correlation in reality.

PRT> So the mapping from real world things to classes was clean and intuitive?

Architect> Yeah, it was pretty clean and intuitive, we had to introduce extra classes to manage other classes and all those sort of workhorse type things, and we had a report manager that handled reports and it all worked out really quite nicely.

-----

< Models have an elegance that is kept pure by including only the essential elements of the



design. >

Mackintosh> We have industrial designers who put together the overall shape, the look, the feel, and then we have coders who put the models together. And its much more like what they do in those design houses than what they do in an engineering plant...Can you name a tolerance in software? The language is so different whereas model-builders that's much more like what we are. Because we never build a physical real thing its much more like building models, getting the aesthetics right. It doesn't need engineering. Phillipe Starck, much more like that. And it can be just as silly, or it can be beautifully elegant. But he's not an engineer.

-----

< Listening to the code can be likened to Schon's (Heidegger's) notion of 'breakdown'. Listening suggests catching subtleties before they reach the point of breakdown. Not actual breakdown, but rather a resistance, or jarring. >

Lethaby> Yes, I am also thinking perhaps not so much about simply the clear breakdown event, but may be your arm is getting sore, because you are holding the hammer slightly the wrong way and your arm shouldn't be getting sore, I am just hammering -- why is my arm getting sore?

PRT> I am just coding, why is this tedious it is not breaking, it is not bouncing off the nail bit but it is not comfortable, it doesn't feel right. It is very hard to put qualitative measures on all those ideas of feeling, the code feeling right and listening to the code, does that bother you at all.

-----

Sullivan> When I was a student I had a Mark 7 Jag, I can still see in the Workshop Manual a picture of -- I think it might have been a cross section... no it. wasn't... it was just a picture of the front of the engine... with it's two cam shafts ... oh, no, it had timing chains on it as well...

And I mean, it has just left an indelible impression on me, what a thing of beauty!

-----

< Breuer associates overall bulk, and its reduction or lack of reduction as indicators of elegance. >

Breuer> When I saw the 'commissions' package specifications -- I don't think, you weren't hear for that were you? I walked out of the meeting. I ended up saying... it was 124 pages -

- please turn it into 15.

-----

< Symmetry is an indicator of both elegance and simplicity. The designer should strive to uncover and exploit the natural symmetries in the structures of both problems and solutions. Most problems have an underlying structure. The designer must strive to identify, uncover this structure. >

Breuer> P> So really you have, I suppose there was an underlying, there was a view there that someone, by taking a view from a particular perspective of the system, of the business domains, of the problems, the structure of the problems that you were trying to solve, you were able to come up with a solution that collapsed down into a grammatically simple structure.

Architect> Elegant and simple, yes. The fact that it did collapse into something so elegant meant that you have to be right. This has to be the way to go.

P> Yeah. Elegance confirms correctness.

Architect> Yes, yes, exactly. And that is a very strong touch-stone. I think that, yeah, symmetry is, I suppose, one of my guiding lines here is that if something is not symmetrical then I have probably got something wrong. As we know when you compose symmetric objects together, they lead to a great deal of asymmetry and of course, when Jeff presented me with this 'miracle happens here' type document where they had all the added parts of Product sketched except for the big hole in the middle, ... it appeared to me that this all too complicated, this just can't be right. It is too complex. We will never finish implementing it and as soon as somebody changes something this is going to be hopeless to undo. Because we don't understand all the inter- connections and so on.

So what is the simplifying first principle that we can pull this down to?. And the simplifying first principle was 'business rules are a tree'.

P> Yeah.

-----

< Symmetry is important because it reduces work. >

< If a software architecture does not exhibit elegance, than it may be because the design lacks symmetry, or has failed to uncover and exploit symmetries in the structure of the problem that it is trying to solve. >

Breuer> ...and, yeah, symmetry is a key issue in the elegance of the solution – is a key.

P> And also from what you have said that you need, if you've come up with an elegant solution, you should drive for that. Elegance in the solution suggests correctness...

Architect> And also means that you are going to less work in the future. And it is going to be less work. Essentially it is all about doing less work.

## **Appendix F: Topic Maps**

### **Purpose**

The topic maps presented in this appendix represent the analysis of categories around topics, and the relationships between categories, within the five broad domains.

## What is 'Software Architecture'?

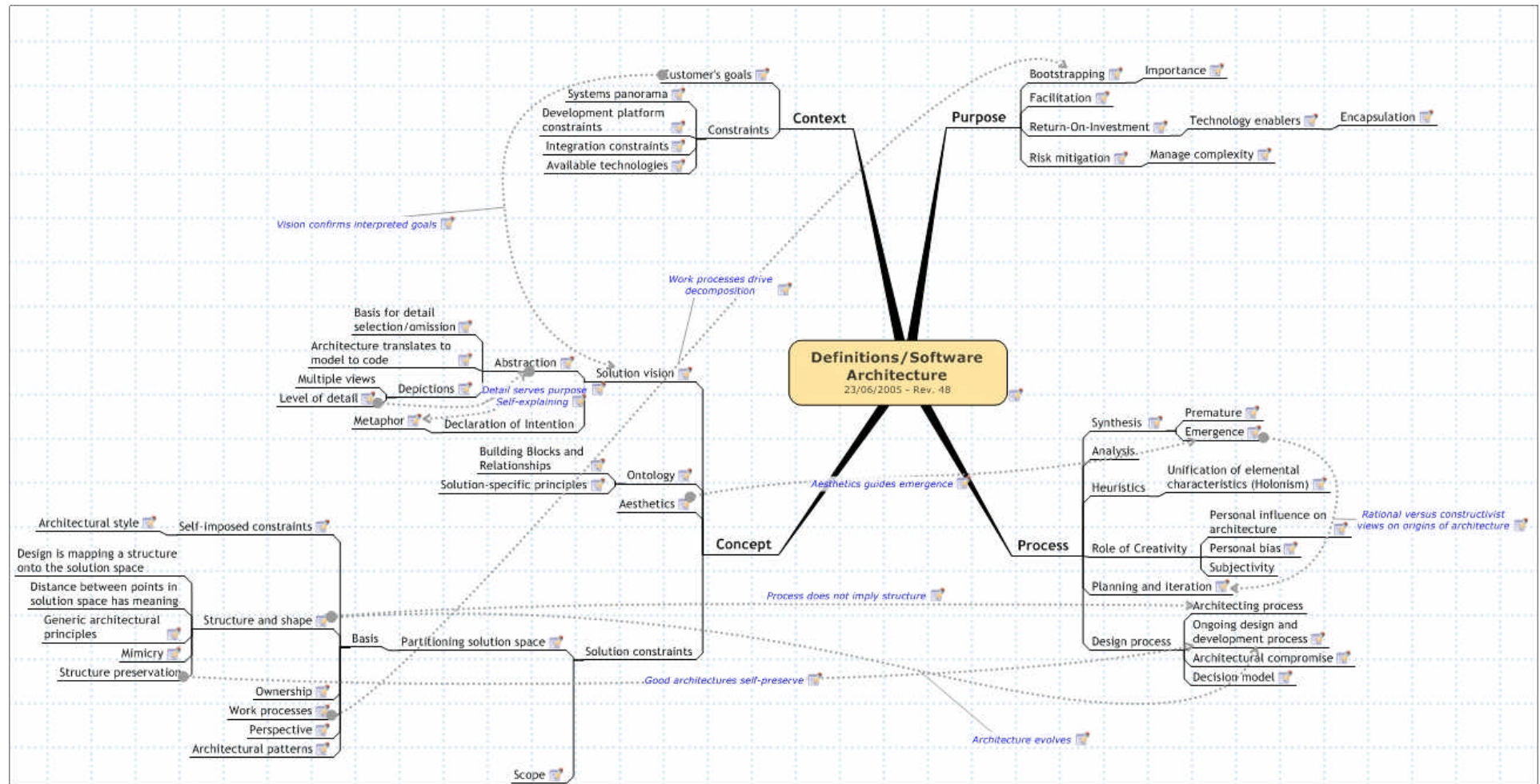


Figure 37: Topic map for 'software architecture'.

## What is 'Software Design'

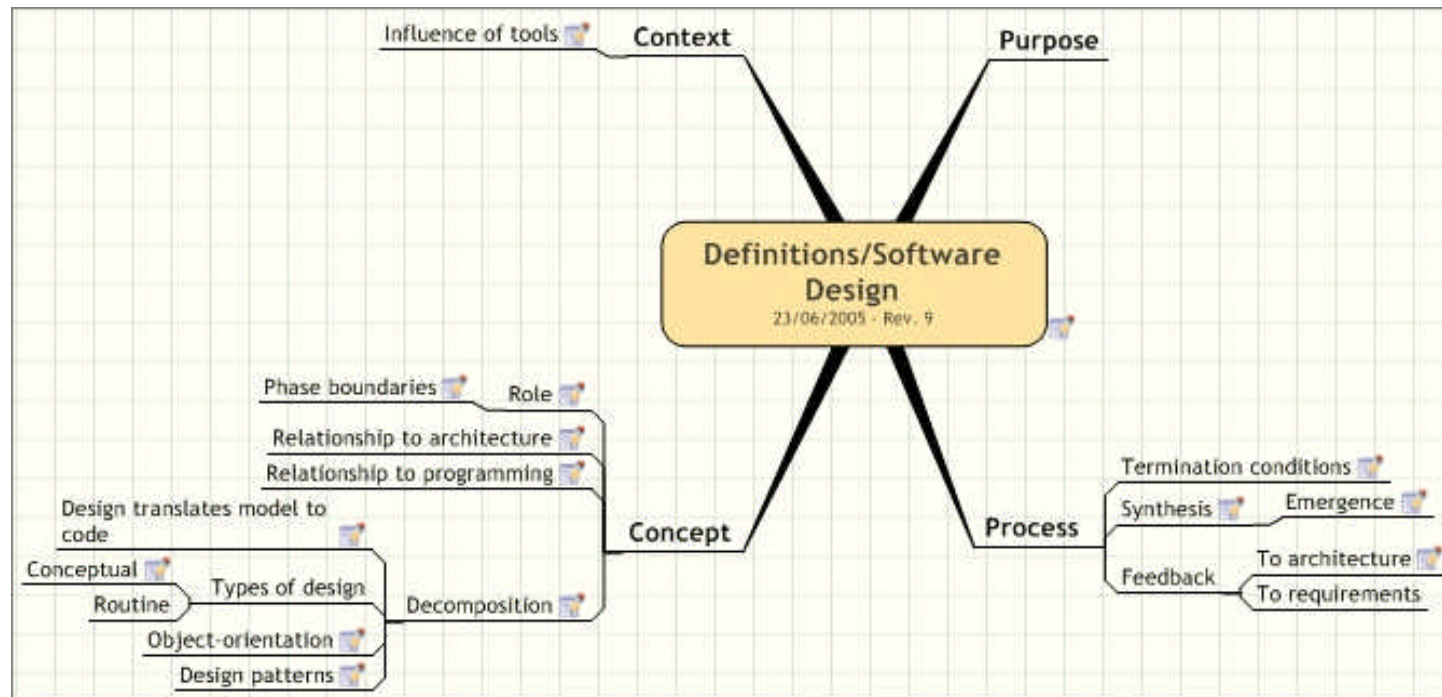


Figure 38: Topic map for 'software design'.

## The ‘Software Architect’ Role

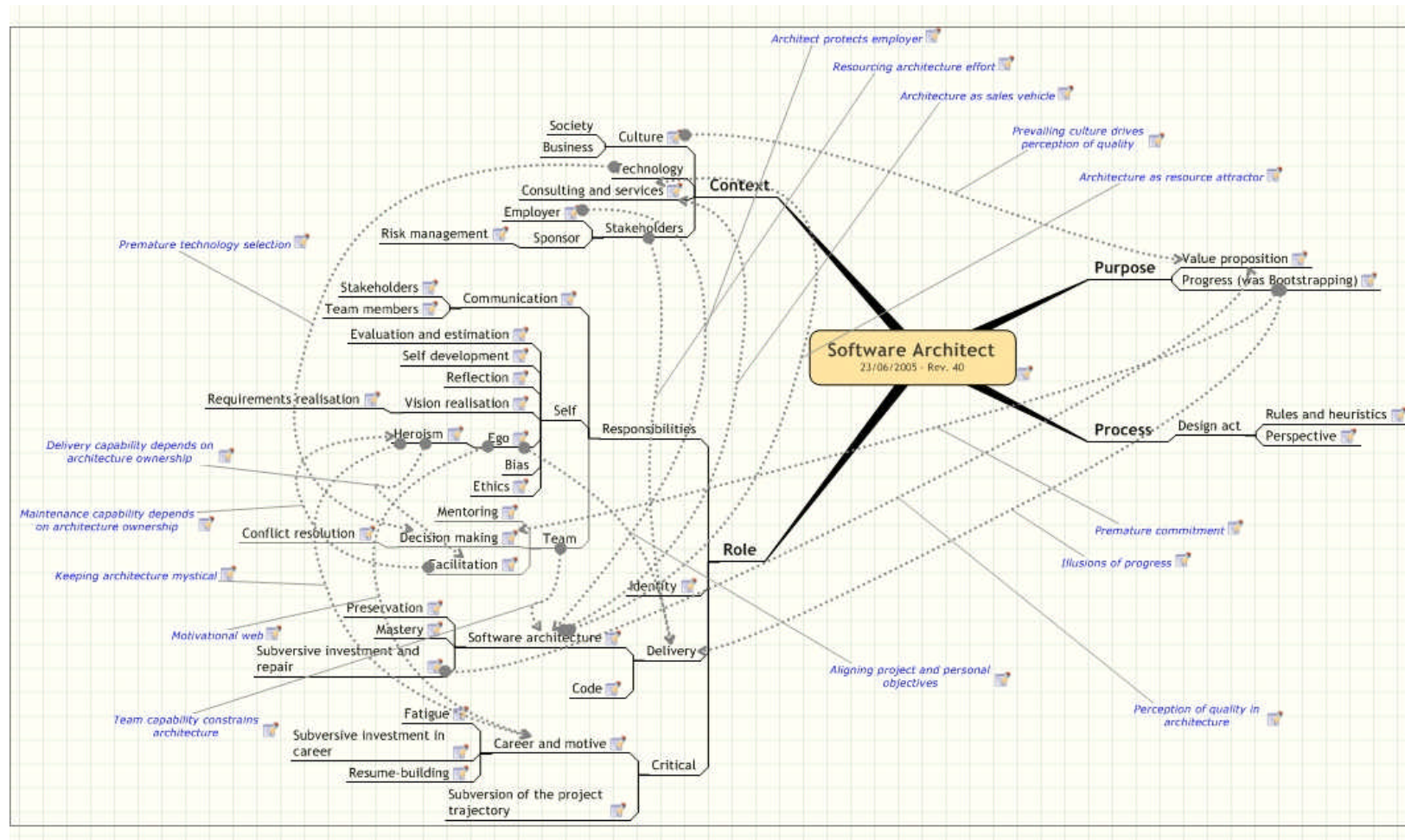


Figure 39: Topic map for ‘the role of the software architect’.



## Methodology

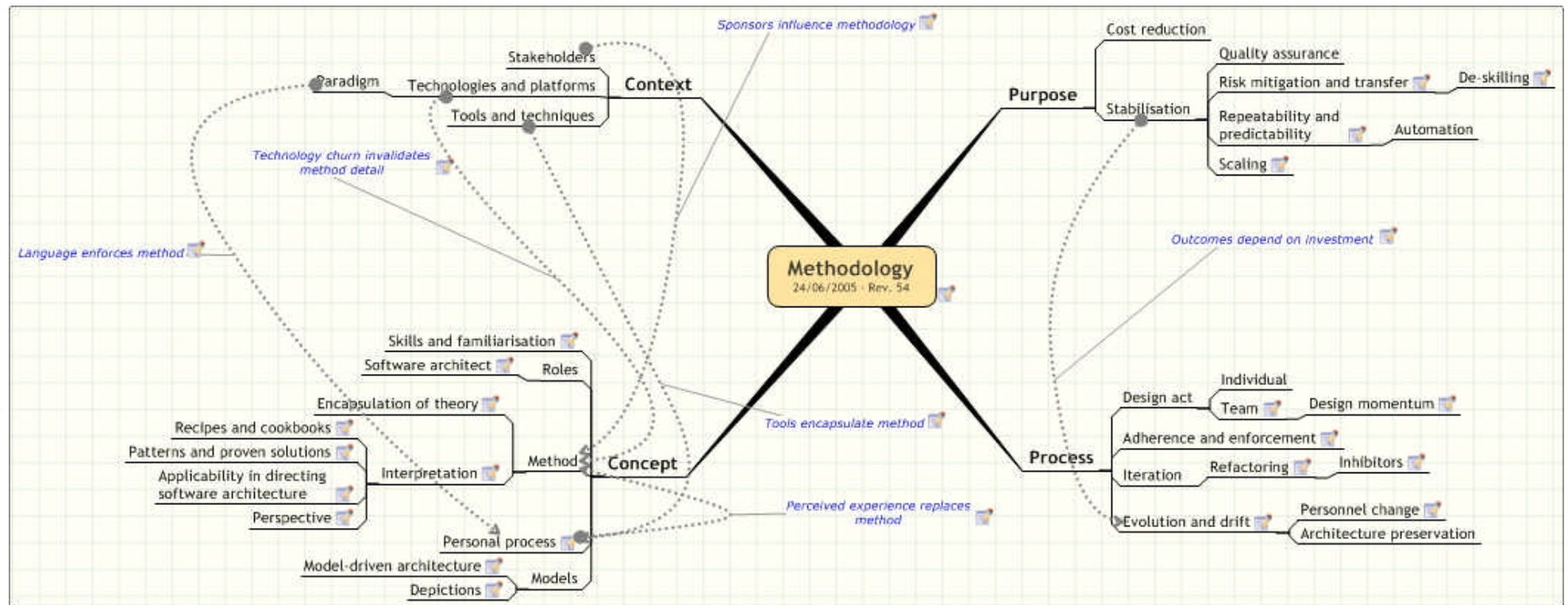


Figure 40: Topic map for 'methodology'.



# The 'Design Act'

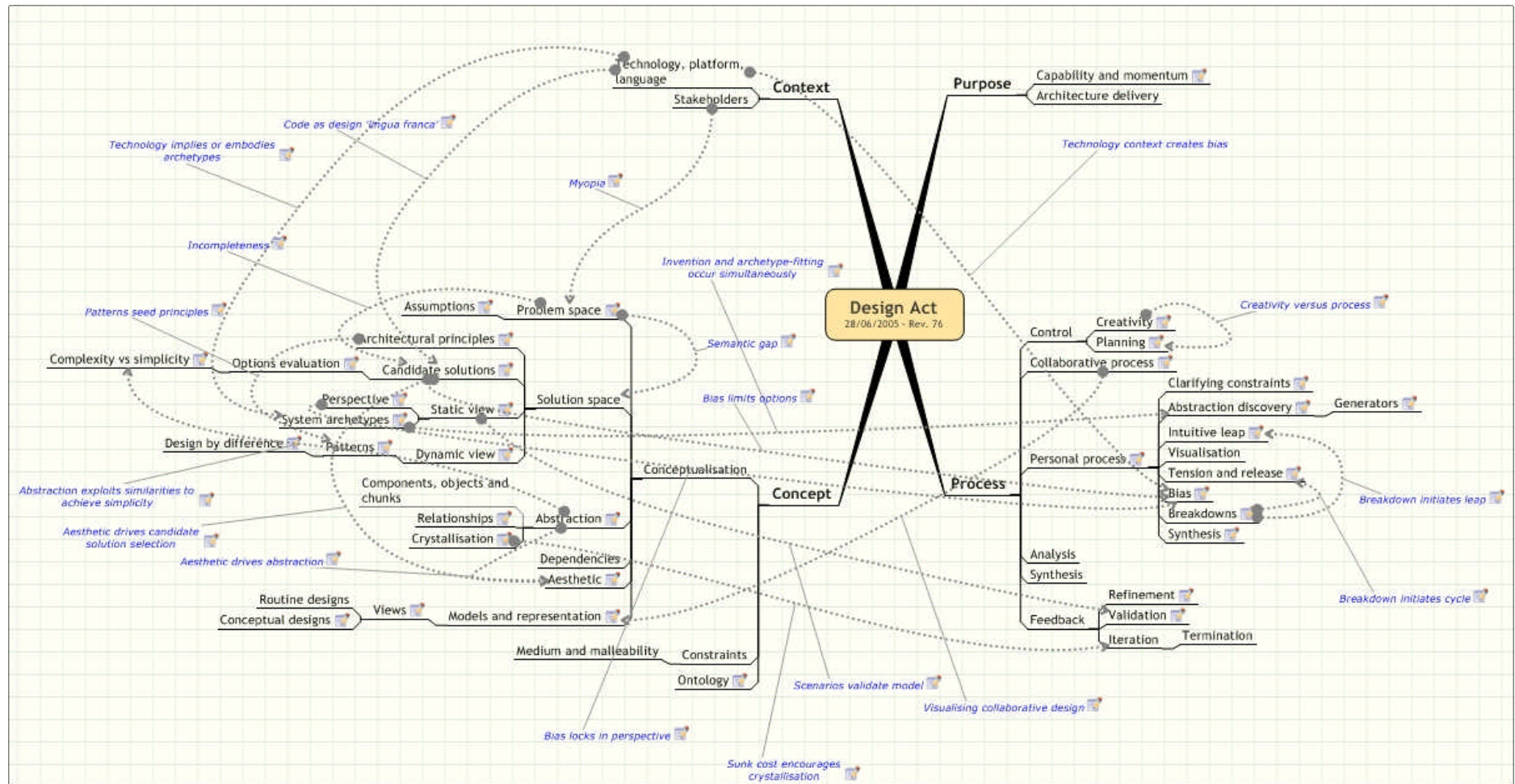


Figure 41: Topic map for 'the design act'.

## Appendix G: Project Website

The project website was created in June 2000 to advertise for participants and explain the project goals as well as the process to be followed by participants. It was deployed at <http://www.csse.monash.edu.au/~ptaylor/> where it has been served continuously for six years. This Appendix presents the site's content.

## Front page



## **'How do expert software architects balance forward planning...' teaser page**

Le Corbusier's master-plans for everything from entire cities to modernist public housing complexes elevated architecture and planning beyond concrete, steel and glass to a blueprint for economic reform and social revolution...



Unfortunately, Le Corbusier's functionalist structures could not be filled with life. They were ultimately judged as impractical and unmalleable. People preferred bungalows and shacks that they could mould to their changing needs over living in a modernist monument to its designer.

System and software designers need architectures. They must ensure that their teams implement a consistent vision and structure. So how do experienced software architects enforce structure whilst preserving flexibility? What criteria do they use to determine this balance?

## **'...and piecemeal growth?' teaser page**

Christopher Alexander has a lot to say about planning and emergence in design...

"In the present way of thinking about architecture, one is supposed to design the building completely, and use the description



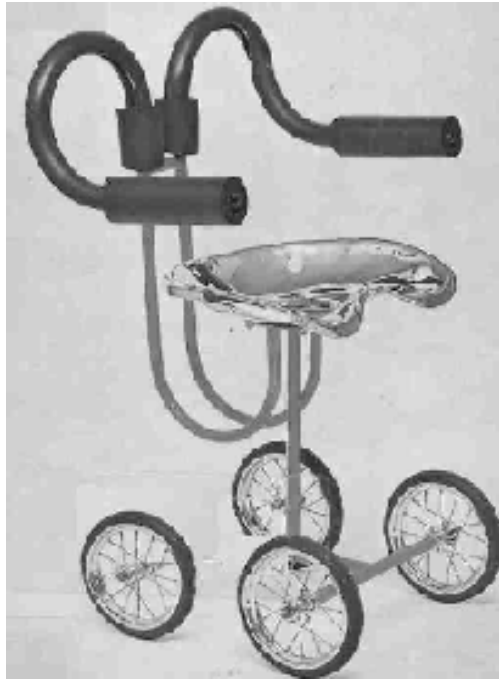
(design, plan, drawing) as a specification from which to build. But it is precisely in this that architecture has gone wrong, and it is because of this that living structure no longer appears in our buildings. Instead of using plans, designs, and so on, we must use generative processes which tell us what to do, rather than detailed drawings which tell us what the end result is supposed to be. That procedure allows us to create living structure..."



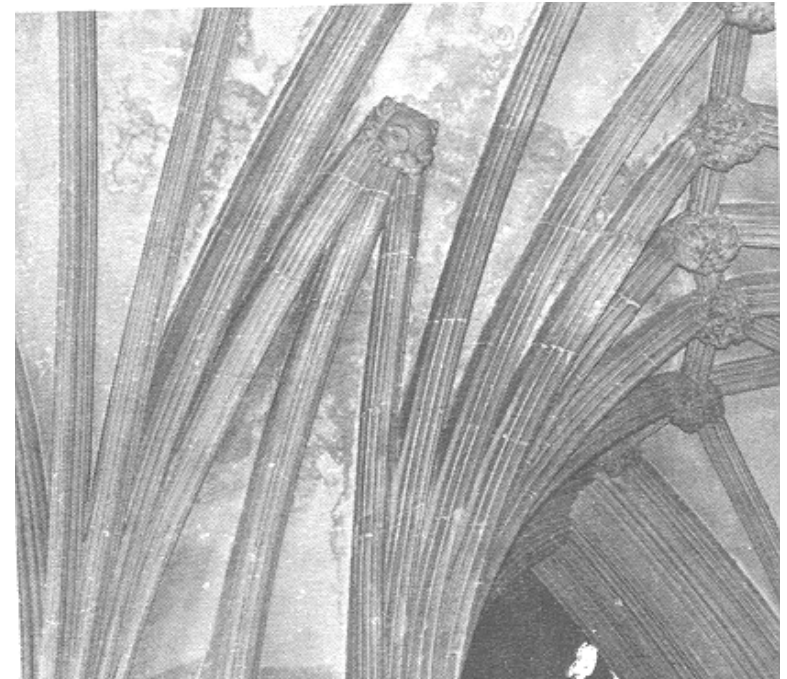
Pattern languages and other generative techniques might produce highly habitable designs over time, but most software architects don't have the time or the right pattern language to realise such 'beauty' in their conceptual and detailed object architectures. How do expert software designers initiate, manage and control piecemeal growth and repair in their software structures?

### **'Is ad hoc software design...unprofessional or purposeful?' teaser page**

Ad hoc is bad... right? Process maturity models (like CMM) have undoubtedly increased software quality substantially around the world. But what do most process models have to say about the way conceptual and architectural software design is actually performed? And do software architects follow them when doing design? If not, why not?



*Ad hoc* office chair by Charles Jencks and Nathan Silver. (exact location unknown).



*Ad hoc* resolution of cathedral roof tracery.

Software processes tend to shun unrepeatable, one-off, ad hoc design and development, because it is not repeatable and often not of high quality. But all design has a creative element which cannot be dictated by processes, there are times when we do not require repeatability, and most design comes down to an individual engineer or architect anyway. So what is the difference between bad adhocism -- the kind software process people despise -- and good adhocism, the kind that delivers pragmatic, working designs quickly and efficiently?

### **'How do experienced software architects articulate their designs...' teaser page**

One of the most acute criticisms of functionalist design was the lack or articulation, or signs and detail, that serve to distinguish a

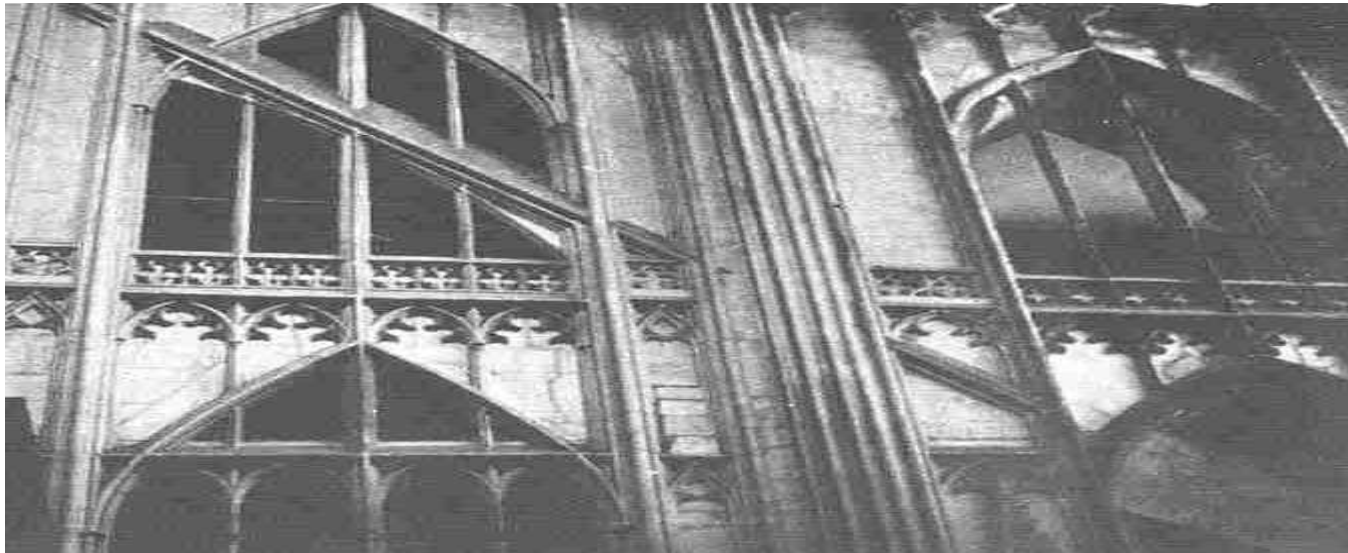
structure and announce its presence or use. Design has historically oscillated on the question of semiotics -- the science of signs and their reading -- from the ridiculous detailing of Victorian buildings and artefacts to faceless, modernist blandness...



Semiotics in software architecture has addressed critical issues such as cognitive perception and habitability in the interests of driving down maintenance costs and increasing extensibility. So how do expert software architects choose their abstractions and compose their structures to provide articulate designs? Do software designers worry about design self-documentation at all?

### **'...and ensure essential structures are preserved?' teaser page**

Even supposedly perfect structures are subject to erosion over time, and can require buttressing against forces unforeseen at design time...



A diagonal buttress added post-hoc to Gloucester cathedral.

What measures do experienced software architects and designers take to ensure that the conceptual clarity of their designs are preserved? What design preservation techniques work, and under what circumstances are software architectures long-lived? Do designers worry about architecture preservation and longevity at all?

### **'About software design' page**

#### *The design 'onion'*

Design is a difficult thing to describe, according to IEEE Software columnist Robert L. Glass (March 1999, p104): Design is one of the most elusive yet fascinating topics in the software field. It is elusive because, no matter how thoroughly academics try to shape it into a teachable, testable, fact-based topic, it just doesn't fit. It is fascinating because design holds the key to the success of most software projects.

Clive Dym (IEEE Spectrum, June 1996, p10) suggests that design can be thought of as an 'onion', comprised of many distinct but interdependent layers:



- design is experiential -- it is about learning by doing;
- design is mathematical -- it only achieves valid academic status when it is framed in mathematical terms;
- design is cognitive -- it is about the functioning of the human mind in creative activity;
- design is social -- it is about process activities, especially in team settings.

Glass concludes that software design is, at heart a trial-and-error, creative process, in which proposed solutions to problems are iterated and enhanced until they are powerful enough to be a complete solution to the problem at hand. This is true of design in other domains as well -- architecture in the built world, artistic design, and some types of engineering design. Glass concludes with the observation that not much work has been done to further our understanding of design across the disciplines, and that:

It is my belief that in the mainstream of the field today, too many still see design as an objective topic. We teach people about design methodologies. We teach them about design representations. 'Turn the methodology crank', we say, 'write down the result in a representation, and you are a skilled designer'. That's just not enough.

Taken collectively, these layers represent a formidable yet fascinating barrier capable of taxing the finest software minds for years to come.

## **'About software architects' page**

### *The software architect--some defining characteristics*

Software architects are experienced software developers who have developed design skills in systems, software architectures and detailed component and module design.

They are usually strong conceptual thinkers, who understand the value of getting the conceptual model absolutely right. They are prepared to defend and preserve the conceptual clarity and purity of their architectures and models.

Their knowledge extends beyond software technology to include systems thinking, communication and negotiation. They may be able to work outside of the Information Technology organisation to resolve issues and deliver solutions. They often have a deep knowledge of several or more business domains, such as insurance, patient administration, or telecommunications fault management, that has been developed while developing systems in these business areas.

They may sport a few scars and a healthy pragmatism from the experience of deploying business-critical systems under tight

deadline pressures, and they have a sense of how easily or otherwise certain characteristics of a proposed system will be to design, implement and deploy.

Some, perhaps even a lot of their knowledge is tacit. Like most experts, they know that they have knowledge and a sense of what is right, or what will work, but they cannot always elucidate their know-how, their methods or approaches.

### *The situated software architect*

The concept of situated action is based on the fact that all actions are performed in a context and are chosen by the actor in response to a whole range of stimuli, and that it really doesn't make much sense to try to interpret action out of context, or to expect action to follow a prescriptive plan.

'Situatedness' has been taken seriously by design researchers, who regard it as a useful way of explaining some of the unpredictability of the act of design. Design is the difficult process of synthesising functional, efficient and elegant structures and artefacts from ill-defined problems whilst dealing with multiple dimensions of interrelated constraints. The messiness of the real-world contexts in which designers work, and the pragmatic effects of non-technical factors such as deadlines, people, politics and should be highly familiar to most experienced software architects.

We know that these 'messy' environmental factors shape software architectures almost as much as methodologies, plans and processes do -- we just don't know how much, how, or why. A model of situated software design action will be useful to illuminate the reality of software architecture and design.

The situated software architect is a product of education, experience and context. He or she is part architect, part engineer, part craftsperson -- a rational designer today, a professional expert tomorrow, a pragmatic hands-on doer the day after. Situated designers do not use theory and methods per se, they filter and use bits of them selectively in response to their expert perceptions of contextual needs.

## **'Related research' page**

### *Alternate Metaphors and Models for Software Production*

The applicability of the engineering and architecture metaphors for software production have been questioned since their adoption. Some alternative metaphors have included Dance, Craft, and Theatre.

Some authors are starting to interpret what a post-modernist software paradigm (or paradigms) might look like. The prospect of replacing the engineering and architecture metaphors is fuelled by recent insights into the software making process that do not sit comfortably with forward engineering -- the ability to select different methods and match them selectively to problem types (see Michael Jackson's problem frames -- the methodologist, not the singer!), heterogeneous components and object-oriented programming languages, multi-design paradigms, and continuous user involvement. The true extent of the inherent flexibility of software fabric is finally being recognised and is feeding back into methods, metaphors and paradigms.

Early results from Richard Gabriel's Feyeraabend Project suggest this 'edge of a paradigm shift' perspective. The ongoing project is investigating recasting the foundations of software engineering in the light of current technology and reported experience. The work includes envisioning, reflection and narrative, and deconstruction of some important software engineering value-frameworks and norms.

Jim Coplien has explored multi-paradigm design and the parallels between design history and software. Jason Baragry has described the movement from misfitting metaphors to a paradigm. Bruce Blum's book (*Beyond Programming*) comprehensively argues that the time has come to abandon engineering metaphors grounded in 'technical rationality' in favour of a kind of adaptive design paradigm. Meanwhile, journals like *Design Studies* have been publishing high quality design research in non-software domains for years -- the tussle between 'design science' and 'design art' appears constantly in its pages.

Research projects that explore the nature of situated design (or situated software design like this one) become exercises in assessing the underlying paradigm when practitioner's reports are generalised and the common themes compared and contrasted with conventional wisdom or 'theory'.

### *Past and Present Design Theory*

Design in its own right -- independent of any realisation media -- has a strong theoretical basis. Journals like *Design Studies* have chronicled design theory for several decades. Foundations were laid by design theorists including Nigel Cross, John Thackara and John Chris Jones. Contemporary researchers like John Gero at the Key Center for Design Computing are researching and publishing situatedness, constructive memory and emergent behaviours in design.

Authorities like the Design Research Society, and national design councils (UK) oversee design promotion and provide a view of current research issues.

### *Pattern Theory*

Christopher Alexander's theory of pattern languages provides one view of how complex designs can evolve over time, with a decentralised form of design control. The situated software architect balances local, regional and corporate design effort in parallel. Lots of work has been done since the early nineties to document 'best practice' software design in a wide range of domains in the

form of patterns. The situated software architect uses patterns extensively, both explicitly and tacitly, and may even write patterns to transfer design knowledge.

### *Contextual Software Design*

The 'situatedness' of software design has been recognised for some time. The Scandinavian School pioneered the application of participative design techniques and other approaches borrowed from industrial design to software. Voluminous work has been done within the HCI community to exploit participative design. The impact of the particular designer's skills and abilities on software productivity rates and software quality are also well recognised. The perceptions, attitudes and values of software architects therefore have a critical effect on the shape and form of software architecture in industry and business.

### *Design Ethnography*

Some researchers are adapting ethnographic techniques to the study of industrial design and software development. This typically involves 'going native' and living 'with the tribe' for a period of time, observing and then generalising the observations back to theory.

## **'Papers' page**

The following papers have been published on software design topics. Send email to request an electronic copy.

Seen, M., Taylor, P., and Dick, M. "Applying a Crystal Ball to Design Pattern Adoption." TOOLS Europe (33), Mont-Saint-Michel, France, 443-454.

Taylor, P. "Adhocism in Software Architecture - Perspectives from Design Theory." International Conference of Software Methods and Tools (2000), Wollongong, 41-50.

Taylor, P. (2000b). "Capable, Productive and Satisfied: Patterns for Productive People." Pattern Languages of Program Design 4, N. Harrison, B. Foote, and H. Rohnert, eds., Addison-Wesley, Reading, Massachusetts, 611-637.

Taylor, P. "Designerly Thinking: What Software Methodology can learn from Design Theory." International Conference on Software Methods and Tools (2000), Wollongong, 107-118.

Taylor, P. "Dynamic Team Structures for Supporting Software Design Episodes." 37th International Conference on Technology of

Object-Oriented Languages and Systems (TOOLS Pacific) 2000, Sydney, 290-301.

Taylor, P. "Evolution of Software Design Knowledge." Australian Conference on Knowledge Management and Intelligent Decision Support (2000), Melbourne.

Taylor, P. "Problem Frames and Object-Oriented Software Architecture." 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific) 2000, Sydney, 70-81.

Taylor, P. (2001a). "The Great Meta-Skills Shortage." Information Age, 34-35.

Taylor, P. "A Software Enterprise is not a Tree." First Australian Conference of Pattern Languages and Programs (KoalaPLoP 2000), Melbourne.

Taylor, P. "Eternal Triangle: Design in Three Discourses." Second Australian Conference of Pattern Languages and Programs (KoalaPLoP 2001), Melbourne.

Taylor, P. "Mayall's Ten Principles in Design: A Software Engineering Response." 2001 Australian Software Engineering Conference (ASWEC 2001), Canberra.

Taylor, P. "Patterns of Software Craft." Second Australian Conference of Pattern Languages and Programs (KoalaPLoP 2001), Melbourne.

## **'FAQ' page**

*What are the project's goals?*

The research project aims to build a model of software design practice, in industry and/or business contexts. It aims to describe the personal design processes, rationale and values adopted by software architects in industry to design, manage and evolve complex (principally object-oriented) software architectures and systems. The research will take a qualitative, interpretivist approach, by focusing on the factors that shape software design in this context. This information will give software engineers and methodologists a current picture of how software architecture and design in the business and industrial contexts is understood and enacted, and how this differs from the 'theoretical' models of design that underly popular methodologies and process improvement models. The model is expected to confirm many aspects of conventional wisdom and software engineering theory, and it will almost certainly diverge from some others. Findings from the project's participants will be compared and contrasted with each other and with both

design theory (the kind of theory that architecture and industrial design is based upon) and software engineering theory. The areas of divergence are expected to become a source of insight into the differences between planned and actual software design and development practice.

*What does 'situated' mean?*

Research into design performed in industry is motivated by the need to understand the effect of these contexts on individual and collective design processes. Many researchers have adopted in situ action-oriented perspectives to interpret action in different ways. In contrast to causal research based on controlled observation of expert designers, the in situ approach takes the researcher out of the laboratory and into a setting where causal and predictive factors frequently become masked and muddled beyond recognition amidst the complexity of the business environment. Researching in situ requires the use of methods based on a paradigm in which engineering facts are considered alongside socially constructed perceptions. Software design, being a form of design that is both highly conceptual and deeply contextual, is an ideal candidate for this kind of research.

The notion of 'situatedness' originated from artificial intelligence research in cognitive models of planning, which moved from forward planning to contextual planning many years ago. Lucy Suchman described action as being understandable and explainable only in its immediate context, coining the term 'situated action' to distinguish this view from the entrenched one of action conforming to a plan, description or specification. Suchman claims:

...the term situated action...underscores the view that every course of action depends in essential ways upon its material and social circumstances (Suchman, "Plans and Situated Actions: The problem of human machine communication", Cambridge University Press, 1987, p. 50).

Rather than to attempt to abstract away action from its circumstances and represent it as a rational plan, the situated approach studies "how people use their circumstances to achieve intelligent action". Rather than build a theory of action out of a theory of plans, the situated approach aims to investigate how people produce and find evidence for plans in the midst of their situated actions. (Others have called this common phenomena 'Posthoc Rationalisation'). Suchman's work paralleled the emergence of interpretivist and ethnographic research techniques in information systems research, and a general recognition of the importance of contextual forces in systems design, development and deployment success.

*Do contextual factors really influence Software Architects all that much?*

Situated action's claim that all actions are meaningless if interpreted out of context undoubtedly applies to some forms of group or social behaviours, but should it -- and does it -- apply to an engineering design activity like software development? This question strikes at the paradigmatic foundation of what it is we do... is software development art, craft, science, engineering, or a hybrid of

some or all of these disciplines?

The question 'should software be art or science' has been debated endlessly and it is not the goal of this project to stir that particular can of worms. This project focuses on the act of design in context, and asks the second question -- does it depend on contextual rather than context-independent factors, and if so, how, and why. The answers, as evidenced in the responses, thoughts and stories of a cohort of expert software architects will build evidence for a particular form of situated design action. It may be profoundly embedded in context, or it may be tightly coupled with context-independent prescriptive techniques and methods -- time will tell. Either way, we will know more about expert software design practice.

Contextual factors that might prove significant in architect's accounts of design include risk management strategies, political and project-related factors, available skill levels, deadline and budget constraints, other contingent factors, and many others. The project aims to understand the designer's values that underpin how these kinds of factors are interpreted and dealt with in creating and evolving complex system and software designs.

*Do good design practices really differ from popular methods?*

At the coarse level, almost certainly not. There is no question that the feedback loop between conventional wisdom and contemporary methods is working reasonably well. This project focuses specifically on how individuals perform complex software design in situ. Contemporary software development methods provide coarse level orientation, particularly for beginners, but not (on the whole) plans, processes or schemas for detailed architectural design.

Neither can they. Methods provide project and design infrastructure, which can orient but cannot dictate design. For example, when less experienced designers follow methods literally, their designs are often naive, immature, and inflexible, and as a result not long-lived. Their accounts of design are of almost no value to this project.

The best architectural designs are produced by experienced designers who draw upon the infrastructure provided by languages and methods, libraries and frameworks, knowledge of the business domain, past experience of architectural design, and other intangible factors to synthesise enduring system and software structures. It is these design acts that are of great interest, because they are borne of valuable expertise and have lasting impact.

Of equal interest are the perceptions of methods that these experienced designers carry, their experience of methods in the past, and the ways that they use methods.

*If software architects are experts, how can you mine their tacit knowledge in an interview?*

This insightful question must be asked of any research that uses interviews, questionnaires, fora, or other means of soliciting insights from experts in any field. What an expert says he does and what he actually does can be different things. Almost by definition, a large component of an expert's knowledge is tacit -- that is, experts internalise certain behaviours over time to the point where they can't even recognise them, let alone explain them to a researcher.

The knowledge management community has suggested lots of techniques that can be used to make tacit knowledge explicit, but these often take time and considerable effort. Getting software architects to write design patterns from their experience of system and software design is one such knowledge mining technique. Other approaches to understanding expert practice include ethnographic techniques such as 'living with the leader of the tribe' for a period to unobtrusively observe his or her behaviours, and controlled protocol analysis in which designers narrate their thoughts whilst performing a design task.

Doing ethnographic research with a development team engaged on large-scale business-critical systems development is both difficult to arrange, for commercial reasons. It constitutes a form of case-study research which risks unpredictable results, and over-influence from the people in the particular project. In protocol analysis, the expert practice is taken out of context and into a 'laboratory' setting, which defeats the purpose of any study of situated action.

The chosen research method -- that of conducting interviews with experienced software architects -- has the advantage of allowing longitudinal experience over many years and many projects to be reported by the architects themselves. Of course it suffers from the selective filtering that these experts will unknowingly apply as they recount their life's work. But there are simple interview techniques that can mitigate some of these risks.

Experts also do not always distinguish between fact and interpretation, or between commonly understood principles and personal values. This is a problem for all interpretivist research. As interpretivist researchers gain experience, they:

- interpret the participant's responses in the light of their understanding of the problem domain, and
- recognise that the interview process produces the expert's perceptions, not necessarily absolute facts.

Interpretivist research methods provide techniques and tools that transform the (literal) interview transcripts and perform word-by-word, phrase-by-phrase textual analysis on the data. The resultant qualitative analyses can be overlaid to construct some hypotheses and deconstruct others. The end result is a form of conceptual model that accounts for individual biases and individual's perspectives.

*Why go to individual software architects to find out about design?*



To research situated software design, an important research question must be addressed -- what or who provides the best access to useful knowledge of situated design, and how should the researcher attempt to access this knowledge? The obvious research target is the software designer him/herself. This choice puts an individual (rather than an organisation or team) at the focus of the study. The hosting organisation, while it provides a critical business, economic and social context for software is not the source of the design decisions that shape the software artifact directly, and is generally inaccessible to intrusive studies, anyway.

The software development team is another viable source of knowledge on situated software design, particularly as all developers perform software design to some degree. Many studies have been performed by design researchers who have 'lived with' and observed a design team. Performing this kind of research with software development teams is becoming more difficult due to confidentiality and commercial sensitivities. Team-based ethnographic research is a must where the research question concerns collective rather than individual design and behaviours. Software architecture design, however, is often an individual and even personal activity. Software development team collaboration tends to support the design and development infrastructure with the minutiae of individual design acts and 'pieces' of software design, rather than system and architectural design.

Team-based research can only capture particular design practices if they emerge or are recognised and observed. By focussing on the expert designer, his or her collective knowledge of many architecture-related roles on many projects and development contexts (the designer's longitudinal experience) can be probed. The most useful knowledge will come from the most experienced designers who have had the opportunity to reflect on practices over many projects and software design contexts.

*Won't the participants answer defensively rather than honestly?*

Designers or developers with little confidence in their ability to choose a design approach, deliver a design and understand its merits and shortcomings might be tempted to cling to conventional wisdom, or the dictates of a particular methodology, when questioned about what they did and why they did it that way.

We believe that the software architects we intend to interview will be experienced enough to admit mistakes, to perform rational assessments of their past designs, architectures and projects, so as not to compromise the study with tactical answers or overtly biased accounts. The confidentiality of the interviews and data ensure a highly secure environment in which to reflect and assess critically.

What happens at the interview?

The interview will provide an opportunity to interact and discuss some areas of your experience and expertise. The interview is run using a structured outline that contains about 30 questions that probe your experience and perceptions of the practice of (object-oriented) software design in industry. Interesting or relevant themes that emerge in the discussion will be explored further with additional unscheduled questions.

*What kinds of questions are asked?*

The questions are organised into six groups that address individual and group software design experience, design of software, the design act, design and time, design knowledge management, and design values. In all sections, the questions are designed to draw out personal experience and expertise.

*How long will the interview take?*

The interview will take 60 minutes, and no more than 90 minutes.

*Where is the interview conducted?*

The interview will be scheduled and conducted at a mutually convenient time and place. The venue will be quiet and private. How is confidentiality ensured?

The interviews will discuss generalised techniques and practice, and you will not be asked to state or discuss the names of companies, products or individuals at any time. If you do, these names will be kept completely confidential.

*What do I have to sign?*

You will need to sign a Consent form. This form allows us to record your consent to participate in this study. Consent forms are standard ethical research practice and a requirement of the University.

*Why is the interview recorded?*

With your permission, the interview will be audio-taped so that everything that is said can be subsequently transcribed into an interview record. This is standard practice amongst qualitative researchers. It is important that the actual words of the interview are captured exactly, so that the interview recording process does not filter or introduce bias.

*What happens to the recording afterwards?*

The interview transcript will be e-mailed back to you so that you can validate the notes, if you wish. The university requires that the audio recording and transcripts are stored in a safe place for a period of five years after the date of the interview. This is standard research practice. After this time they will be destroyed.

*Can I retract something I said in the interview?*

Yes -- if you say something that you are not sure of for any reason, you can retract it there and then. Or, when you receive the transcript, you can request that any part of the text be deleted. In this case, the retracted statement(s) will not be considered further during the analysis.

*Can I choose not to answer some questions?*

Yes -- you can provide as much or as little as you feel is appropriate in response to any of the questions. You may elect not to answer any of the questions without compromising your answers to the other questions.

*How is the interview data analysed?*

The data from a number of interviews will be analysed using standard qualitative analysis techniques, that include conceptual clustering, categorisation, conceptual modelling using qualitative categories and relationships. This will deliver a 'conceptual model' that identifies recurrent themes, concepts, categories, and the relationships between them. Qualitative research results explain but do not predict or forecast. Qualitative data and analysis provide an extremely rich view of complex, real-world domains. Qualitative research approaches are particularly suitable for explaining human culture, practice, the interplay between practice and culture, and expertise.

*What does 'Ethics Committee approval' mean?*

Monash University requires that any research performed 'on any animal' be cleared by an Ethics Committee. This ensures that all research activity has been vetted for ethical content and practice. This project has been cleared by the committee (clearance number 2000/469). This means that the Ethics Committee has reviewed and approved every page of explanatory text, including the interview outline and the consent form. Ethics Committee approval also means that if anyone has any issue with the way the study is being conducted, they can directly contact the Monash University Ethics Committee for clarification or to have a concern addressed.

*What are the implications of situated design?*

This study has the potential to uncover two broad themes. Either the majority of the interviewed participants will tell us that their best architectures and designs resulted from a methodology-driven design process, in which contextual factors had minimal influence and their critical design decisions could be traced transparently through forward-planned rational decision-making on the basis of

common-sense engineering trade-offs. Alternately, we may find a high degree of contextual factors evident in the emergence of practical, workable software architectures. We cannot predict which of these themes will emerge, and the study has been designed not to prejudice either potential outcome.

The former finding will reinforce the use of plans, processes and methods as the drivers of design, according to the reported experiences and perceptions of a cohort of highly experienced designers. This will be good for software engineering. We can then retire from academic studies such as this to less arcane pursuits such as teaching industry about methodologies and process maturity models.

The later finding will reinforce the impact of contextual factors, the individual's ability to work in and through context, to synthesise, and to draw upon expertise in a way that aligns with designers who work in other (non-software) media. This will also be good for software engineering, because it will shed light on the nature of complex software design as practiced, in a form that methodology can learn from. As for us, we might just have to devise even more arcane research projects to discover yet more about the Situated Software Architect.

## **'People' page**

*Assoc. Prof. Christine Miggins*

Christine Miggins is the Principal Researcher and supervisor. Christine is an Associate Professor and Head of the School of Computer Science and Software Engineering - Monash University (Caulfield).

*Prof. Richard Mitchell*

Richard Mitchell is an associate supervisor. Richard provides input to the research design, and ongoing review of the project's progress and deliverables. Richard is past Professor of Computing at the University of Brighton and a Principal Consultant with Inferdata.

*Paul Taylor*

Paul Taylor is the project's chief researcher. Paul is currently performing the indepth interviews and subsequent analysis, and will draw the project's findings together into a research thesis, feedback for the participants, and subsequent publications.

## **'Participate' page**

*Participation*

Are you an experienced software architect? Do you like reflecting on your experiences with software design, methods, and teams?  
...then please consider being a participant...

You must be:

- in Australia -- we have to interview you face-to-face;
- a software engineer, software architect, team leader or system developer who has been responsible for software architecture and design;
- experienced with object-oriented and/or component technologies;
- experienced in the design and ongoing development of one or more non-trivial OO/component software systems.

We'll send you a simple preliminary survey form to help us to understand your background. If you fit the profile of the people we are seeking, we'll schedule a private interview (about 1.5 hours) at your convenience.

In return, we'll keep you informed of the study's progress and findings.

To register your interest as a participant, send an email to Paul Taylor.

### **'Disclaimer' page**

This is a personal page published by the author. The ideas and information expressed on it have not been approved or authorised by Monash University either explicitly or implicitly.

In no event shall Monash University be liable for any damages whatsoever resulting from any action arising in connection with the use of this information or its publication, including any action for infringement of copyright or defamation. Any complaints about the contents of home pages on this particular server ([www.csse.monash.edu.au](http://www.csse.monash.edu.au)) should be directed to the Webmaster, who will fully investigate the complaint and take appropriate action.

## Appendix H: Author's Publications

The author published the following publications both as a result of this research, and in the period during this research was conducted.

- Seen, M., Taylor, P., and Dick, M. (2000). "Applying a Crystal Ball to Design Pattern Adoption", Proceedings of the Thirty-third International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 33), Mont-Saint-Michel, France. R. Mitchell, J.-M. Jezequel, J. Bosch, B. Meyer, A. Cameron Wills, and M. Woodman, eds. IEEE Computer Society, 443-454.
- Taylor, P. (2000). "Adhocism in Software Architecture: Perspectives from Design Theory", International Conference of Software Methods and Tools (2000), Wollongong. J. Gray and P. Croll, eds. IEEE Computer Press, 41-50.
- Taylor, P. (2000). "Capable, Productive and Satisfied: Patterns for Productive People." Pattern Languages of Program Design IV, N. Harrison, B. Foote, and H. Rohnert, eds., Addison-Wesley, Reading, Massachusetts, 611-637.
- Taylor, P. (2000). "Designerly Thinking: What Software Methodology can learn from Design Theory", International Conference on Software Methods and Tools (2000), Wollongong. J. G. a. P. Croll, ed. IEEE Computer Press, 107-118.
- Taylor, P. (2000). "Dynamic Team Structures for Supporting Software Design Episodes", Proceedings of the Thirty-seventh International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 37), Sydney. B. Henderson-Sellers and B. Meyer, eds. IEEE, 290-301.
- Taylor, P. (2000). "Evolution as a Model of Software Design Knowledge Formation and Propagation", Proceedings of the Australian Conference on Knowledge Management and Intelligent Decision Support (ACKMIDS 2000), Melbourne. F. Burnstein and H. Linger, eds. Monash University, 182-198.
- Taylor, P. (2000). "Problem Frames and Object-Oriented Software Architecture", Proceedings of the Thirty-seventh International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 37), Sydney. B. Henderson-Sellers and B. Meyer, eds. IEEE, 70-81.

- Taylor, P. (2000). "The Great Meta-Skills Shortage", ACS Information Age (Feb-Mar 2001), 34-35.
- Taylor, P. (2000). "A Software Enterprise is not a Tree", Proceedings of the First Asia-Pacific Conference of Pattern Languages and Programs (KoalaPLoP 2000), Melbourne. J. Coplien and L. Xhao, eds. RMIT University.
- Taylor, P. (2000). "Eternal Triangle: Design in Three Discourses", Proceedings of the Second Asia-Pacific Conference of Pattern Languages and Programs (KoalaPLoP 2001), Melbourne. J. Noble and N. Harrison, eds. University of Wellington.
- Taylor, P. (2001). "Interpreting Mayall's Principles in Design", Proceedings of the Australian Software Engineering Conference (ASWEC 2001), Canberra. D. Grant and L. Stirling, eds. IEEE Computer Press, 297-306.
- Taylor, P. (2001). "Patterns of Software Craft", Proceedings of the Second Asia-Pacific Conference of Pattern Languages and Programs (KoalaPLoP 2001), Melbourne. J. Noble and N. Harrison, eds. University of Wellington.
- Taylor, P. R. (2001). "Designing Philosophers", Proceedings of the Twelfth Australasian Conference on Information Systems (ACIS 2001), Coffs Harbour. G. Finnie, D. Cecez-Kecmanovic, and B. Lo, eds. Southern Cross University, 653-660.
- Taylor, P. R. (2001). "Patterns as Software Design Canon", Proceedings of the Twelfth Australasian Conference on Information Systems (ACIS 2001), Coffs Harbour. G. Finnie, D. Cecez-Kecmanovic, and B. Lo, eds. Southern Cross University, 661-670.
- Taylor, P. R. (2001). "Researching the 'Situated Software Architect': Describing the Effects of Context on the Design Practice of Experienced Software Designers", Proceedings of the Third International Workshop on Strategic Knowledge and Concept Formation (SKCF 2001), Sydney. J. S. Gero and K. Hori, eds. Key Centre of Design Computing and Cognition, University of Sydney, 261-276.
- Taylor, P. R. (2003). "Vernacularism in Software Design Practice", Proceedings of the Twelfth International Conference on Information Systems Development: Constructing the Infrastructure for the Knowledge Economy (ISD2003), Melbourne.
- Taylor, P. R. (2004). "Vernacularism in Software Design Practice: Does Craftsmanship have a Place in Software Engineering?", Australasian Journal of Information Systems(Special Issue 2003/4), 14-25.

## Appendix I: Bibliography

- AgileAlliance. (2005). "The Agile Manifesto", accessed 23 March 2005, <http://www.agilemanifesto.org>
- Agre, P. E. (1995). "Computational Research on Interaction and Agency", *Artificial Intelligence*, 72(1), 1-52.
- Agresti, W. W. (1986). "What are the New Paradigms?", New Paradigms for Software Development, W. Agresti, ed., IEEE Computer Society Press, New York.
- Alexander, C. (1964). *Notes on the Synthesis of Form*, Harvard University Press, New York.
- Alexander, C. (1977). *A Pattern Language*, Oxford University Press, New York.
- Alexander, C. (1979). *The Timeless Way of Building*, Oxford University Press, New York.
- Alexander, C. (1988). "A City is not a Tree", Design After Modernism, J. Thackara, ed., Thames and Hudson, London, 67-84.
- Allen, T. J. (1977). *Managing the Flow of Technology: Technology Transfer and the Dissemination of Technological Information within the R&D Organization*, Massachusetts Institute of Technology, Boston.
- Amann, K. (1992). "Scientific Expertise as a Social Process", Software Development and Reality Construction, C. Floyd, R. Budde, H. Zullighoven, and R. Keil-Slawik, eds., Springer-Verlag, Berlin, 131-139.
- Baird, F., Moore, C. J., and Jagodzinski, A. P. (2000). "An ethnographic study of engineering design teams at Rolls-Royce Aerospace", *Design Studies*, 21(4), 333-355.
- Ball, L. J., and Ormerod, T. C. (2000). "Applying ethnography in the analysis and support of expertise in engineering design", *Design Studies*, 21(4), 403-421.
- Bamberger, J. (1991). *The mind behind the musical ear*, Harvard University Press, Cambridge, MA.
- Bansler, J. P., and Bodker, K. (1993). "A reappraisal of structured analysis: design in an organisational context", *ACM Transactions on Information Systems*, 11(2), 165-193.
- Baragry, J., and Reed, K. (2001). "Why we need a Different View of Software



- Architecture”, *Working IEEE/IFIP Conference on Software Architecture*, Amsterdam, The Netherlands.
- Bardram, J. E. (1997). “Plans as Situated Action: An Activity Theory Approach to Workflow Systems”, *Proceedings of the Fifth European Conference on Computer Supported Cooperative Work*, Dordrecht. J. A. Hughes, W. Prinz, T. Rodden, and K. Schmidt, eds. Kluwer Academic Publishers, 17-32.
- Barwise, J., and Perry, J. (1983). *Situations and Attitudes*, MIT Press, Cambridge, MA.
- Bateson, G. (1980). *Mind and Nature: A Necessary Unity*, Bantam Books, New York.
- Beck, K. (2000). *Embracing Change: Extreme Programming Explained*, Cambridge University Press, Cambridge.
- Beedle, M., Devos, M., Sharon, Y., Schwaber, K., and Sutherland, J. (2000). “SCRUM: A Pattern Language for Hyperproductive Software Development”, *Pattern Languages of Program Design IV*, N. Harrison, B. Foote, and H. Rohnert, eds., Addison-Wesley, Reading, Massachusetts, 637-653.
- Benbasat, I., and Zmud, R. W. (1999). “Empirical research in information systems: the practice of relevance”, *MIS Quarterly*, 23(1), 3-16.
- Beyer, H., and Holtzblatt, K. (1994). “Calling Down the Lightning”, *IEEE Software*, 11(5), 106-113.
- Blackmore, S. (1999). *The Meme Machine*, Oxford University Press, Oxford.
- Blum, B. (1996). *Beyond Programming: To a New Era of Design*, Oxford University Press, Oxford.
- Boehm, B. W. (1976). “Software Engineering”, *IEEE Transactions on Computers*, 25(12), 1226-1241.
- Boehm, B. W. (1988). “A Spiral Model of Software Development and Enhancement”, *IEEE Computer*, 21(5), 61-72.
- Boland, R. (1979). “Control, Causality, and Information System Requirements”, *Accounting, Organizations and Society*, 4(4), 259-272.
- Boland, R. J., Tenkasi, R. V., and Te'eni, D. (1994). “Designing Information Technology to Support Distributed Cognition”, *Organization Science*, 5(3), 456-475.
- Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*, Benjamin Cummings, Redwood City, California.

- Borenstein, N. S. (1991). *Programming as if People Mattered: Friendly Programs, Software Engineering and Other Noble Delusions*, Princeton University Press, Princeton, New Jersey.
- bpmn.org. (2007). "Business Process Modelling Notation", accessed 2 Feb 2007, <http://www.bpmn.org/>
- Brand, S. (1994). *How Buildings Learn: What Happens to Them after They're Built*, Penguin, New York.
- Broadbent, G. (1973). *Design in architecture; architecture and the human sciences*, John Wiley & Sons, London.
- Bryman, A. (1988). *Quantity and Quality in Social Research*, Unwin Hyman, London.
- Bucciarelli, L. L. (1988). "An ethnographic perspective on engineering design", *Design Studies*, 9(4), 159-168.
- Bucciarelli, L. L. (1994). *Designing Engineers*, MIT Press, Cambridge, Massachusetts.
- Budde, R., and Zullighoven, H. (1992). "Software Tools in a Programming Workshop", *Software Development and Reality Construction*, C. Floyd, R. Budde, H. Zullighoven, and R. Keil-Slawik, eds., Springer-Verlag, Berlin, 252-268.
- Budgen, D. (1994). *Software Design*, Addison-Wesley, Reading, Massachusetts.
- Budgen, D. (1995). "Design models from software design methods", *Design Studies*, 16(3), 293-325.
- Bullock, A., and Woodings, R. B. (1983). *The Fontana Dictionary of Modern Thinkers*, Fontana, London.
- Burrell, G., and Morgan, G. (1979). *Sociological paradigms and organisational analysis: elements of the sociology of corporate life*, Heinemann Educational, London.
- Busby, J. S. (1998). "The neglect of feedback in engineering design organisation", *Design Studies*, 19(1), 103-117.
- Capurro, R. (1992). "Informatics and Hermeneutics", *Software Development and Reality Construction*, C. Floyd, R. Budde, H. Zullighoven, and R. Keil-Slawik, eds., Springer-Verlag, Berlin, 363-375.
- Carroll, J. (2000). "Examining Methodology Adoption and Use: Building Understanding from Process Research", *Proceedings of 11th Australasian Conference on Information Systems (ACIS 2000)*, Brisbane. G. G. Gable and M. Vitali, eds., CD-ROM. 12

pages.

- Carroll, J., and Swatman, P. A. (2001). "Structured-case: A methodological framework for building theory in information systems research", *European Journal of Information Systems*, 9(4), 235-242.
- Checkland, P. (1981). *Systems Thinking, Systems Practice*, Wiley, Chichester, UK.
- Checkland, P., and Scholes, J. (1990). *Soft systems methodology in action*, Wiley, Chichester, UK.
- Churchman, C. W. (1968). *The Systems Approach*, Dell Publishing Co., New York.
- Clancey, W. J. (1993). "Situated action: A neuropsychological interpretation (Response to Vera and Simon)", *Cognitive Science*, 17(1), pp.87-107.
- Clegg, C. (1994). "Psychology and information technology: the study of cognition in organizations", *British Journal of Psychology*, Vol 85, 449-475.
- Cockburn, A. (2002). *Agile Methods*, Addison-Wesley, Reading, Massachusetts.
- Coplien, J. O. (1995). "A Generative Organisational Pattern Language", Pattern Languages of Program Design I, J. O. Coplien and D. C. Schmidt, eds., Addison-Wesley, Reading, Massachusetts.
- Coplien, J. O. (1996). *Software Patterns*, Lucent Technologies, Bell Labs Innovations, New York.
- Coyne, R. (1999). *Technoromanticism: Digital Narrative, Holism, and the Romance of the Real*, MIT Press, Cambridge, Massachusetts.
- Coyne, R. D. (1991). "Is designing mysterious? Challenging the dual knowledge thesis", *Design Studies*, 12(3), 124-131.
- Coyne, R. D. (1995). *Designing Information Technology in the Postmodern Age: From Method to Metaphor*, MIT Press, Cambridge, Massachusetts.
- Crellin, J., Horn, T., and Preece, J. (1990). "Evaluating evaluation: a case study of the use of novel and conventional evaluation techniques in a small company", *Human Computer Interaction (INTERACT 90)*, Amsterdam. D. Diaper, D. Gilmore, G. Cockton, and B. Shackel, eds. Elsevier, 329-335.
- Cross, N. (1977). *The Automated Architect*, Pion Limited, London.
- Cross, N. (1993). "Science and design methodology", *Research in Engineering Design*, Vol 5, 63-69.

- Curtis, B., Krasner, H., and Iscoe, N. (1988). "A Field Study of the Software Design Process for Large Systems", *Communications of the ACM*, 31(11), 1268-1287.
- Dahlbom, B. (1992). "The Idea that Reality is Socially Constructed", *Software Development and Reality Construction*, C. Floyd, R. Budde, H. Zullighoven, and R. Keil-Slawik, eds., Springer-Verlag, Berlin, 101-126.
- Dahlbom, B., and Mathiassen, L. (1993). *Computers in Context: The Philosophy and Practice of Systems Design*, Blackwell, Cambridge, MA.
- Daly, J., Kellehear, A., and Gliksman, M. (1997). *The Public Health Researcher: A Methodological Guide*, Oxford University Press, Melbourne.
- Davies, S. P. (1991). "Characterising the program design activity: neither strictly top-down nor globally opportunistic", *Behaviour & Information Technology*, 10(3), 173-190.
- Dawkins, R. (1976). *The Selfish Gene*, Oxford University Press, Oxford.
- Dawkins, R. (1996). *Climbing Mount Improbable*, Penguin, London.
- DeGrace, P., and Stahl, L. H. (1990). *Wicked Problems, Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms*, Yourdon Press, Englewood Cliffs, New Jersey.
- Dekleva, S. M. (1992). "The Influence of the Information Systems Development Approach", *MIS Quarterly*, 16(3), 355-373.
- DeMarco, T. (1978). *Structured Analysis and System Specification*, Prentice-Hall, Englewood Cliffs, New Jersey.
- Dennett, D. (1995). *Darwin's Dangerous Idea*, Penguin, London.
- Dewey, J. (1916). *Democracy and Education: An Introduction to the Philosophy of Education*, Free Press, New York.
- Dewey, J. (1958). *Experience and Nature*, Dover, New York.
- Dietrich, G. B., Walz, D. B., and Wynekoop, J. L. (1997). "The failure of software development technology diffusion: a case for mass customization", *IEEE Transactions on Engineering Management*, 44(4), 390-399.
- Dijkstra, E. W. (2002). "Edsger Wybe Dijkstra: 1930-2002", accessed 8th August 2002, <http://www.cs.utexas.edu/users/UTCS/notices/dijkstra/ewdobit.html>
- Dilthey, W. (1931). *Gesammelte Schriften, Vol VIII, Weltanschauungslehre*, B. G. Teubner, Stuttgart.

- Dym, C. L. (1995). "Peeling the Design Onion", *IEEE Spectrum*, (June 1995), 10-12.
- Eaves, D. (1992a). "Nolan's Stage Model(s): The Rage for Order", *Technical Report 1/92*, Department of Information Systems, Monash University, Working Paper Series.
- Eaves, D. (1992b). "The Prospects of a Formal Discipline of Software Engineering", *Technical Report 2/92*, Department of Information Systems, Monash University, Working Paper Series.
- Edwards, J. (2006). "State of the Art vs. State of the Practice: A Personal Perspective on the Changes in the Australian Software Engineering Landscape", *Australian Software Engineering Conference (ASWEC 2006)*, Sydney, IEEE Computer Press, 4-13.
- Fergusson, M., and Shaw, G. (2004). "Information Systems Research: A Question of Relevance", *Seventh Australasian Conference on Information Systems (ACIS 2004)*, Hobart. C. D. Keen, C. Urquhart, and J. Lamp, eds. University of Tasmania, 219-230.
- Feyerabend, P. (1993). *Against Method*, Verso, London.
- Fitzgerald, B. (1997). "The use of systems development methodologies in practice: a field study", *Information Systems Journal*, 7(3), 201-212.
- Floyd, C. (1992a). "Human Questions in Computer Science", *Software Development and Reality Construction*, C. Floyd, R. Budde, H. Zullighoven, and R. Keil-Slawik, eds., Springer-Verlag, Berlin, 15-27.
- Floyd, C. (1992b). "Software Development as Reality Construction", *Software Development and Reality Construction*, C. Floyd, R. Budde, H. Zullighoven, and R. Keil-Slawik, eds., Springer-Verlag, Berlin, 86-100.
- Foote, B. (2000). "Deploy People along the Grain of the Domain", *Seventh Conference on Pattern Languages of Programs*, Honolulu, Hawaii  
<http://www.laputan.org/patterns/grain.html>.
- Foote, B., and Opdyke, W. F. (1995). "Lifecycle and Refactoring Patterns that Support Evolution and Reuse", *Pattern Languages of Program Design I*, J. O. Coplien and D. C. Schmidt, eds., Addison-Wesley, Boston.
- Frampton, K. (1988). "Place-Form and Cultural Identity", *Design After Modernism*, J. Thackara, ed., Thames and Hudson, London.
- Frampton, K., Barrow, R., Hamilton, M., and Grossman, B. (2005). "A Study of the In-

- Practice Application of a Commercial Software Architecture”, *Proceedings of the Australian Software Engineering Conference (ASWEC 2005)*, Brisbane, IEEE Computer Society, 292-301.
- Gabriel, R. (2002). "Dreamsongs", accessed 23 May 2002, <http://www.dreamsongs.org>
- Gabriel, R. P. (1996). *Patterns of Software: Tales from the Software Community*, Oxford University Press, New York.
- Gadamer, H. (1976). *Philosophical Hermeneutics*, University of California Press, Berkeley, California.
- Gallagher, S. (1991). *Hermeneutics and Education*, SUNY Press, Albany, N.Y.
- Galliers, R. D. (1992). "Choosing Information Systems Research Approaches", *Information Systems Research: Issues, Methods and Guidelines*, R. Galliers, ed., Blackwell Scientific Publishers, London, 144-162.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software Architecture*, Addison Wesley, Reading, Massachusetts.
- Gane, C., and Sarson, T. (1979). *Structured Systems Analysis: Tools and Techniques*, Prentice-Hall, New York.
- Gans, H. J. (1967). *The Levittowners*, Penguin Press, London.
- Gasson, S. (1999). "A social action model of situated information systems design", *ACM SIGMIS Database*, 30(2), 82-97.
- Gero, J. S. (1996). "Creativity, emergence and evolution in design: concepts and framework", *Knowledge Based Systems*, 9(7), 435-448.
- Gero, J. S. (1998a). "Conceptual Designing as a Sequence of Situated Acts", *Artificial Intelligence in Structural Engineering*, I. Smith, ed., Springer, Berlin, 165-177.
- Gero, J. S. (1998b). "Towards a model of designing which includes its situatedness", *Universal Design Theory*, H. Grabowski, S. Rude, and G. Grein, eds., Shaker Verlag, Aachen, 47-56.
- Gero, J. S., and McNeill, T. (1998). "An approach to the analysis of design protocols", *Design Studies*, 19(1), 21-61.
- Gilchrist, T. (1989). "Incremental System Development: A Tutorial", *DOC #BCS-G2850*, Boeing Computer Services, Seattle, WA.

- Glaser, B. G., and Strauss, A. L. (1967). *The Discovery of Grounded Theory*, Aldine Publishing Company, New York.
- Glass, R. L. (1999). "On Design", *IEEE Software*, Mar/Apr 1999, 104-103.
- Glass, R. L. (2006). "The Standish report: does it really describe a software crisis?", *Communications of the ACM*, 49(8), 15-16.
- Goguen, J. A. (1992). "The Denial of Error", Software Development and Reality Construction, C. Floyd, R. Budde, H. Zullighoven, and R. Keil-Slawik, eds., Springer-Verlag, Berlin, 193-202.
- Grabow, S. (1983). *Christopher Alexander: The Search for a New Paradigm in Architecture*, University of Chicago Press, Chicago.
- Habermas, J. (1997). "Modern and Postmodern Architecture", Rethinking Architecture: A Reader in Cultural Theory, N. Leach, ed., Routledge, London, 227-235.
- Hall, D., and Hall, I. (1996). *Practical Social Research: Project Work in the Community*, MacMillan Press, Hampshire.
- Hamilton, A. (1992). "Hermeneutics in Contemporary Computer Research", *Report No. 13/92*, Department of Information Systems, Monash University, Melbourne.
- Hammersley, M. (1997). *Reading Ethnographic Research*, Longman, New York.
- Heidegger, M. (1962). *Being and Time*, J. Macquarie and E. Robinson, translators, Blackwell, Oxford.
- Hesse, R., Woolsey, G., and Swanson, H. S. (1980). *Applied Management Science: A Quick and Dirty Approach*, Science Research Associates, Chicago.
- Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004). "Design science in information systems research", *MIS Quarterly*, 28(1), 75-105.
- Hickman, L. A. (1992). *John Dewey's Pragmatic Technology*, Indiana University Press, Bloomington.
- Highsmith, J. A. (2002). *Agile software development ecosystems*, Addison-Wesley, Boston.
- Hirschheim, R. (2001). "Information Systems Epistemology: An Historical Perspective", London School of Economics, London.
- Hirschheim, R., and Klein, H. (1989). *The Emergence of Pluralism in Information Systems Development: Stories, Consequences and Implications for the Legitimation of Systems Objectives*, Templeton College, Oxford.

- Hubbard, B. (1996). *A Theory for Practice: Architecture in Three Discourses*, MIT Press, Cambridge, Massachusetts.
- Hutchins, E. (1983). "Understanding Micronesian Navigation", Mental Models, D. Gentner and A. Stevens, eds., Erlbaum, Hillsdale, New Jersey.
- Jackson, M. (1995). *Software Requirements and Specification: a Lexicon of Practice, Principles and Prejudices*, Addison-Wesley, Reading.
- Jacobs, J. (1964). *The Death and Life of Great American Cities*, Penguin Books, Middlesex, England.
- Jacobson, I. (1992). *Object Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, New York.
- Jagodzinski, P., Reid, F., and Culverhouse, P. (2000a). "Design Studies Special Issue on Ethnography: Editorial", *Design Studies*, 21(4), 315-317.
- Jagodzinski, P., Reid, F. J. M., Culverhouse, P., Parsons, R., and Phillips, I. (2000b). "A study of electronics engineering design teams", *Design Studies*, 21(4), 375-402.
- Jayaratna, N. (1994). *Understanding and Evaluating Methodologies: A Systemic Framework*, McGraw-Hill, London.
- Jencks, C., and Silver, N. (1973). *Adbocism: The Case for Improvisation*, Anchor Books, New York.
- Johnston, R. B. (1999). "The Problem with Planning: The Significance of Theories of Activity for Operations Management", PhD thesis, School of Business Systems, Monash University.
- Jones, B. O. (1980). "The Social Impact of Microcomputers", *The Impact of Microcomputers on Industry, Education and Society*, Canberra. J. D. Morrison, ed. Australian Academy of Science, 80-96.
- Jones, J. C. (1988). "Softecnica", Design After Modernism, J. Thackara, ed., Thames and Hudson, London, 216-226.
- Kaplan, S. M. (2000). "Co-Evolution in Socio-Technical Systems", *Proceedings of Computer Supported Cooperative Work (CSCW 2000)*, Philadelphia, ACM Press, New York.
- Kauffman, S. A. (1995). *At home in the universe: the search for laws of self-organization and complexity*, Oxford University Press, New York.
- Keen, P. G. W. (1991). "Relevance and rigour in information systems research: improving



- quality, confidence, cohesion and impact”, *Information Systems Research: Contemporary Approaches and Emergent Traditions*, H. A. Nissen, H. K. Klein, and R. A. Hirschheim, eds., Elsevier Science.
- Keil-Slawik, R. (1992). “Artefacts in Software Design”, *Software Development and Reality Construction*, C. Floyd, R. Budde, H. Zullighoven, and R. Keil-Slawik, eds., Springer-Verlag, Berlin, 168-188.
- Khushalani, A., Smith, R., and Howard, S. (1994). “What Happens when Designers Don't Play by the Rules: Towards a Model of Opportunistic Behaviour in Design”, Department of Information Systems, Monash University, Working Paper Series.
- Kirsh, D. (1995). “The intelligent use of space”, *Artificial Intelligence*, Vol 72, 1-52.
- Klein, H. K., and Lyytinen, K. (1992). “Towards a New Understanding of Data Modelling”, *Software Development and Reality Construction*, C. Floyd, R. Budde, H. Zullighoven, and R. Keil-Slawik, eds., Springer-Verlag, Berlin, 203-219.
- Kluback, W., and Weinbaum, M. (1957). *Dilthey's Philosophy of Existence: Introduction to Weltanschauungslehre*, Vision Press, London.
- Kotre, J. (1995). *White Gloves: How We Create Ourselves Through Memory*, The Free Press, New York.
- Kruchten, P. (2000). *The Rational Unified Process: An Introduction*, Addison-Wesley, Reading, Massachusetts.
- Lammers, S. (1986). *Programmers at Work*, Microsoft Press, Redmond.
- Laurel, B. (1993). *Computers as Theatre*, Addison-Wesley, Reading, Massachusetts.
- Lave, J. (1988). *Cognition in practice: mind, mathematics, and culture in everyday life*, Cambridge University Press, Cambridge.
- Lave, J., and Wenger, E. (1991). *Situated Learning: Legitimate Peripheral Participation*, Cambridge University Press, Cambridge.
- Lawson, B. (1997). *How Designers Think*, Architectural Press, Oxford.
- Lehman, M. M., and Belady, L. A. (1985). *Program Evolution: Processes of Software Change*, Academic Press, San Diego.
- Lloyd, P. (2000). “Storytelling and the development of discourse in the engineering design process”, *Design Studies*, 21(4), 357-373.
- Loftus, E., and Palmer, J. (1974). “Reconstruction of automobile destruction: An example

- of the interaction between language and memory”, *Journal of Verbal Learning and Behaviour*, Vol 13, 585-589.
- Louridas, P. (1999). “Design as bricolage: anthropology meets design thinking”, *Design Studies*, 20(6), 517-535.
- Love, T. (2000). “Philosophy of design: a meta-theoretical structure for design theory”, *Design Studies*, 21(3), 293-313.
- Lovgren, J. (1994). “How to Choose Good Metaphors”, *IEEE Software*, May 1994, 86-88.
- Lueg, C. (1998). “Supporting Situated Actions in High Volume Conversational Data Situations”, *SIGCHI Conference on Human Factors in Computing*, Los Angeles, ACM Press, New York, 472-479.
- Lycett, M., and Paul, R. J. (1998). “Information Systems Development: The Challenge of Evolutionary Complexity”, *Proceedings of the Sixth European Conference on Information Systems (ECIS 98)*, Aix-en-Provence, France. W. R. J. Baets, ed. Euro-Arab Management School, Granada, Spain, 1-15.
- Lyytinen, K. (1987). “A Taxonomic Perspective of Information Systems Development Theory: Theoretical Constructs and Recommendations”, *Critical Issues in Information Systems Research*, R. Boland and R. A. Hirschheim, eds., Wiley, New York.
- Markus, M. L., and Bjorn-Andersen, N. (1987). “Power over Users: Its Exercise by System Professionals”, *Communications of the ACM*, 30(6), 498-504.
- Martin, J. (1991). *Rapid Application Development*, Macmillan, Indianapolis.
- Maturana, H. R., and Varela, F. J. (1980). *Autopoiesis and Cognition: The Realization of the Living*, D. Reidel Pub. Co., Dordrecht, Holland.
- Mayall, W. H. (1979). *Principles in Design*, Van Nostrand Rienhold, New York.
- McBreen, P. (2002). *Software Craftsmanship: The New Imperative*, Addison-Wesley, Boston.
- McFarland, G. (1986). “The Benefits of Bottom Up Design”, *ACM SIGSOFT Software Engineering Notes*, 11(5), 43-51.
- McIlroy, M. D. (1968). “Mass Produced Software Components”, *Proceedings of the First NATO Conference on Software Engineering*, Garmisch, Germany, 138-155.
- McLuhan, M. (1962). *The Gutenberg Galaxy: The Making of Typographic Man*, University of Toronto Press, Toronto.

- McLuhan, M. (1964). *Understanding Media: The Extensions of Man*, Routledge and Kegan Paul, London.
- McLuhan, M., and Fiore, Q. (1967). *The Medium is the Massage*, Bantam Books, New York.
- McPhee, K. (1996). "Design Theory and Software Design", *Technical Report TR 96-26*, University of Alberta, Edmonton, Alberta.
- Meredith, R. (2002). "On the Philosophies of Rationality and the Nature of Decision Support Systems", PhD thesis, School of Information Management & Systems, Monash University, Melbourne.
- Meyer, B. (1988). *Object-Oriented Software Construction*, Prentice Hall International (UK), New York.
- Miles, M. B., and Huberman, M. A. (1994). *Qualitative Data Analysis*, SAGE Publications, Thousand Oaks, California.
- Mitchell, T. (1988). "The Product as Illusion", *Design After Modernism*, J. Thackara, ed., Thames and Hudson, London, 44-51.
- Mitchell, W. J. (1977). *Computer-Aided Architectural Design*, Petrocelli Charter, New York.
- Myerson, J. (1993). "Design renaissance: selected papers from the International Design Congress", *International Design Congress*, Glasgow. J. Myerson, ed. Open Eye Publishing.
- Nardi, B. A. (1996). *Context and Consciousness: Activity Theory and Human-Computer Interaction*, MIT Press, Cambridge, Massachusetts.
- Naur, P. (1991). *Computing: A Human Activity*, Addison-Wesley, Reading, Massachusetts.
- Nonaka, I. (1991). "The Knowledge-Creating Company", *Harvard Business Review*, (Nov-Dec), 96-104.
- Nonaka, I., and Takeuchi, H. (1995). *The Knowledge-Creating Company*, Oxford University Press, Oxford.
- Norman, D. A. (1988). *The Design of Everyday Things*, Doubleday Currency, New York.
- Norman, D. A. (1991). "Cognitive Artifacts", *Designing Interaction: Psychology at the Human-Computer Interface*, J. M. Carroll, ed., Cambridge University Press, Cambridge, 17-38.
- Nygaard, K. (1986). "Program development as social activity", *Proceedings of the Tenth World Computer Congress (IFIP 86)*, Amsterdam. H. G. Kugler, ed. North-Holland, 189-198.

- Oxman, R. (1999). "Educating the designerly thinker", *Design Studies*, 20(2), 105-122.
- Page-Jones, M. (1980). *The Practical Guide to Structured Systems Design*, Prentice-Hall, New Jersey.
- Palmer, R. (1969). *Hermeneutics*, Northwestern University Press, Evanston, Illinois.
- Palmer, S. R., and Felsing, J. M. (2002). *A Practical Guide to Feature-Driven Development*, Prentice Hall, New Jersey.
- Parnas, D. L. (1985). "Software aspects of strategic defense systems", *American Scientist*, (Sep-Oct), 432-440.
- Parnas, D. L., and Clements, P. C. (1986). "A rational design process: how and why to fake it", *IEEE Transactions on Software Engineering*, SE-12(2), 251-257.
- Petroski, H. (1992). *To Engineer is Human*, Vintage Books (Random House), New York.
- Pirsig, R. M. (1974). *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values*, The Bodeley Head Ltd., London.
- Popper, K. R. (1969). *Conjectures and refutations: the growth of scientific knowledge*, Routledge and K. Paul, London.
- Raymond, E. S. (1999). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly.
- Redmond-Pyle, D. (1996). "Software development methods and tools: some trends and issues", *IEEE Software Engineering Journal*, Mar 1996, 99-103.
- Rittel, H. J., and Weber, M. M. (1984). "Planning problems are wicked problems", *Developments in Design Methodology*, N. Cross, ed., Wiley, Chichester, 135-144.
- Roberts Jr., T. L., Gibson, M. L., Fields, K. T., and Rainer Jr., R. K. (1998). "Factors that Impact Implementing a System Development Methodology", *IEEE Transactions on Software Engineering*, 24(8), 640-649.
- Robertson, T. (2004). "Doing technology design and being an information architect", *Proceedings of Understanding of Socio Technical Action (USTA 2004)*, Edinburgh, 40-44.
- Robertson, T., and Hewlett, C. (2004). "HCI Practices and the Work of Information Architects", *Proceedings of the Australia-Pacific Conference on Computer Human Interaction (APCHI 2004)*, Rotorua, New Zealand, 369-378.
- Robertson, T., Hewlett, C., Harvey, S., and Edwards, J. (2003). "A Role with No Edges: The Work Practices of Information Architects", *Proceedings of HCI International (HCII*

2003), Crete, 396-400.

- Rowe, P. G. (1987). *Design Thinking*, MIT Press, Cambridge, Massachusetts.
- Rumbaugh, J., Jacobson, I., and Booch, G. (2004). *The Unified Modelling Language Reference Manual (2nd Edition)*, Addison-Wesley Professional, Essex.
- Salingaros, N. A. (2006). "Life and Complexity in Architecture from a Thermodynamic Analogy", *A Theory of Architecture*, Umbau-Verlag, Solingen, Germany.
- Sargent, P. (1994). "Design science or nonscience", *Design Studies*, 15(4), 389-402.
- Saule, S. (2000). "Ethnography", *Research methods for students and professionals*, K. Williamson, ed., Centre for Information Studies, Charles Sturt University, Wagga Wagga, 159-176.
- Schank, R. C., and Abelson, R. P. (1977). *Scripts, plans, goals and understanding: an inquiry into human knowledge structures*, Wiley, Hillsdale, New Jersey.
- Schauder, D. (2000). "Seven questions for information management and systems researchers", *Research methods for students and professionals*, K. Williamson, ed., Centre for Information Studies, Charles Sturt University, Wagga Wagga, 305-312.
- Schauder, D. (2002). Discussion of the number of in-depth interviews required for a typical interpretivist IS research project, Department of Information Systems, Monash University, June 2002.
- Schirmbeck, E. (1987). *Idea, Form, and Architecture*, Van Nostrand Reinhold, New York.
- Schon, D. A. (1983). *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, New York.
- Schon, D. A. (1987). *Educating the Reflective Practitioner: Toward a New Design for Teaching and Learning in the Professions*, Jossey-Bass Publishers, San Francisco.
- Seaman, C. B. (1999). "Qualitative Methods in Empirical Studies of Software Engineering", *IEEE Transactions on Software Engineering*, 25(4), 557-572.
- Searle, J. R. (1969). *Speech Acts*, Cambridge University Press, Cambridge.
- Senge, P. M. (1992). *The Fifth Discipline: The Art and Science of the Learning Organization*, Random House Australia, Sydney.
- Shanks, G., Rouse, A., and Arnott, D. (1993). "A review of approaches to research and scholarship in information systems", *Proceedings of the Fourth Australasian Conference on Information Systems (ACIS 93)*, Brisbane. P. Ledington, ed., 29-44.

- Shaw, M., and Garlan, F. (1996). *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, New Jersey.
- Simon, H. A. (1983). *Reason in Human Affairs*, Basil Blackwell, Oxford.
- Simon, H. A. (1985). *The Sciences of the Artificial*, MIT Press, Cambridge, Massachusetts.
- Simsion, G. C. (2005). "Data Modeling: Description or Design?", PhD thesis, School of Information Systems, University of Melbourne.
- Snow, C. P., and Collini, S. (1993). *The Two Cultures and the Scientific Revolution*, Cambridge University Press, Cambridge.
- Stanfill, C., and Waltz, D. (1986). "Toward memory-based reasoning", *Communications of the ACM*, 29(12), 1213-1228.
- Steadman, P. (1979). *The Evolution of Designs: Biological Analogy in Architecture and the Applied Arts*, Cambridge University Press, Cambridge.
- Stebbins, G. L. (1971). *Processes of Organic Evolution*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.
- Strauss, A., and Corbin, J. (1998). *Basics of Qualitative Research: Techniques and procedures for developing grounded theory*, SAGE Publications, Newbury Park.
- Suchman, L. A. (1987). *Plans and Situated Actions: The problem of human machine communication*, Cambridge University Press, Cambridge.
- Suchman, L. A. (1994). "Do categories have politics? The language/action perspective reconsidered", *Computer Supported Cooperative Work*, 2(3), 177-190.
- Sutcliffe, A. (1988). *Jackson Systems Development*, Prentice-Hall, New Jersey.
- Takeuchi, H., and Nonaka, I. (1986). "The New Product Development Game", *Harvard Business Review*, Jan-Feb 1986.
- Talukdar, S., Rehg, J., and Elfes, A. (1988). "Descriptive Models for Design Projects", *Artificial Intelligence in Engineering Design*, J. S. Gero, ed., Computational Mechanics Publications, Avon, UK.
- Taylor, F. W. (1911). *The Principles of Scientific Management*, Harper and Row, New York.
- Taylor, P. (1992). "Experiences with Object Technology", *Proceedings of the Ninth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 9)*, Sydney. C. Mingins, W. Haebich, J. Potter, and B. Meyer, eds., 507-519.

- Taylor, P. (1993). "Towards a Reuse Policy", *Proceedings of the Twelfth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 12)*, Sydney. C. Mingins, W. Haebich, J. Potter, and B. Meyer, eds., 49-60.
- Taylor, P. (1995). "Documenting a Framework's User Interface", *Proceedings of the Eighteenth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 18)*. C. Mingins, R. Duke, and B. Meyer, eds., 197-210.
- Taylor, P. (1997). "Patterns: An Introduction to a Movement", *Systems Magazine*, Auscom Publishing, 59-63.
- Taylor, P. (2000a). "Capable, Productive and Satisfied: Patterns for Productive People", *Pattern Languages of Program Design IV*, N. Harrison, B. Foote, and H. Rohnert, eds., Addison-Wesley, Reading, Massachusetts, 611-637.
- Taylor, P. (2000b). "Evolution as a Model of Software Design Knowledge Formation and Propagation", *Proceedings of the Australian Conference on Knowledge Management and Intelligent Decision Support (ACKMIDS 2000)*, Melbourne. F. Burnstein and H. Linger, eds. Monash University, 182-198.
- Taylor, P. (2001a). "The Great Meta-Skills Shortage", *ACS Information Age* (Feb-Mar 2001), 34-35.
- Taylor, P. R. (2001b). "Designing Philosophers", *Proceedings of the Twelfth Australasian Conference on Information Systems (ACIS 2001)*, Coffs Harbour. G. Finnie, D. Cecez-Kecmanovic, and B. Lo, eds. Southern Cross University, 653-660.
- Taylor, P. R. (2001c). "Interpreting Mayall's Principles in Design", *Proceedings of the Australian Software Engineering Conference (ASWEC 2001)*, Canberra. D. Grant and L. Stirling, eds. IEEE Computer Press, 297-306.
- Taylor, P. R. (2001d). "Patterns as Software Design Canon", *Proceedings of the Twelfth Australasian Conference on Information Systems (ACIS 2001)*, Coffs Harbour. G. Finnie, D. Cecez-Kecmanovic, and B. Lo, eds. Southern Cross University, 661-670.
- Taylor, P. R. (2001e). "Patterns of Software Craft", *Proceedings of the Second Asia-Pacific Conference of Pattern Languages and Programs (KoalaPLoP 2001)*, Melbourne. J. Noble and N. Harrison, eds. University of Wellington.
- Taylor, P. R. (2001f). "Researching the 'Situated Software Architect': Describing the Effects of Context on the Design Practice of Experienced Software Designers", *Proceedings of the Third International Workshop on Strategic Knowledge and Concept*

- Formation (SKCF 2001)*, Sydney. J. S. Gero and K. Hori, eds. Key Centre of Design Computing and Cognition, University of Sydney, 261-276.
- Taylor, P. R. (2003). "Vernacularism in Software Design Practice", *Proceedings of the Twelfth International Conference on Information Systems Development: Constructing the Infrastructure for the Knowledge Economy (ISD2003)*, Melbourne.
- Taylor, P. R. (2004). "Vernacularism in Software Design Practice: Does Craftsmanship have a Place in Software Engineering?", *Australasian Journal of Information Systems*(Special Issue 2003/4), 14-25.
- Thackara, J. (1986). *New British Design*, Thames and Hudson, London.
- Thackara, J. (1988). "Beyond the Object in Design", *Design After Modernism*, J. Thackara, ed., Thames and Hudson, London, 11-34.
- Truex, D. P., Baskerville, R., and Travis, J. (2000). "Amethodical systems development: the deferred meaning of systems development methods", *Management and Information Technology (Pergamon)*, 2000(10), 53-79.
- Turner, J. A. (1987). "Understanding the elements of system design", *Critical Issues in Information Systems Research*, R. J. Boland and R. A. Hirschheim, eds., John Wiley & Sons, Chichester, UK, 97-111.
- Valkenburg, R., and Dorst, K. (1998). "The reflective practice of design teams", *Design Studies*, 19(3), 249-271.
- van Manen, M. (1990). *Researching Lived Experience: Human Science for an Action-Sensitive Pedagogy*, State University of New York Press, New York.
- Venturi, R. (1970). "Learning from Levittown", Studio sessions, Yale University, Venturi, Scott Brown & Associates, Yale.
- Venturi, R., Scott Brown, D., and Izenour, S. (1977). *Learning from Las Vegas*, MIT Press, Cambridge, Massachusetts.
- Walker, D., and Cross, N. (1976). *Design: The man-made object*, The Open University Press.
- Walz, D. B., Elam, J. J., and Curtis, B. (1993). "Inside a Software Design Team: Knowledge Acquisition, Sharing and Integration", *Communications of the ACM*, 36(10), 63-76.
- Warfield, J. N. (1994). *A Science of Generic Design: Managing Complexity through Systems Design*, Iowa State University Press, Iowa.



- Weatherall, M. (1979). *Scientific Method*, The English Universities Press Ltd., London.
- Williamson, K., Burstein, F., and McKemmish, S. (2000). "The two major traditions of research", *Research Methods for Students and Professionals*, K. Williamson, ed., Centre for Information Studies, Charles Sturt University, Wagga Wagga, 25-47.
- Winograd, T., and Flores, F. (1986). *Understanding Computers and Cognition: A New Foundation for Design*, Addison-Wesley, Reading, Massachusetts.
- Wirfs-Brock, R. J. (1993). *Responsibility-Driven Design*, Prentice Hall, New Jersey.
- Wirfs-Brock, R. J. (2006). "Explaining Your Design", *IEEE Software*, 23(6), 96-98.
- Yin, R. K. (1994). *Case Study Research: Design and Methods*, SAGE Publications, Thousand Oaks, California.
- Yourdon, E., and Constantine, E. (1979). *Structured Design*, Prentice-Hall, New Jersey.
- Zave, P. (1984). "The Operational versus the Conventional Approach to Software Development", *Communications of the ACM*, 27(2), 104-118.
- Zelkowitz, M. (1988). "Resource Utilization During Software Development", *Journal of Systems and Software*, (No. 8), 331-336.
- Zikmund, W. G. (1994). *Business Research Methods*, Dryden Press, Fort Worth, Texas.

## Appendix J: Index

- ‘Almost alright’, 26
- ‘Design engagement’, 142, 213, 223, 224, 231, 235, 239, 240, 244, 245, 248, 249, 250, 251, 252, 253, 255, 259, 260, 262, 263, 267, 269, 275
- ‘Design episode’, 57, 96, 105, 168, 171, 182, 185, 186, 187, 188, 189, 191, 193, 196, 198, 202, 204, 209, 218, 223, 224, 225, 237, 238, 243, 244, 246, 247, 259, 261, 264, 267, 269, 271, 275
- ‘Design trajectory’, 81, 171, 172, 209, 229, 234, 236, 239, 247, 255, 261, 262, 263, 267, 268, 269, 274, 275
- Abstraction, 4, 24, 27, 30, 34, 50, 77, 95, 109, 122, 123, 128, 129, 136, 140, 148, 150, 154, 155, 156, 157, 158, 159, 165, 172, 176, 177, 178, 186, 189, 194, 200, 213, 214, 215, 216, 220, 223, 237, 239, 241, 244, 245, 259, 261, 262, 286
- Abstractionist, 213
- Action-present, 80
- aesthetic-in-opposition, 26
- Agile, 11, 15, 19, 86, 235, 248, 260, 271
- All-at-once, 82, 85, 86
- Analysis, 68, 259, 267, 290
- a*-rationality, iii, 4, 5, 74, 230, 246, 263, 275
- Archetype, 161, 163, 164, 165, 166, 167, 178, 194, 200, 209, 210, 216, 219, 220, 226, 246, 262, 275
- Architect-technician, 228
- Architecture, software, 118, 121, 126, 210, 286, 290, 333
- Ashbee*, 250
- Associative memory, 219
- Autopoiesis, 28, 59, 65
- Bounded rationality, 38, 229
- Breuer*, 117, 121, 122, 124, 131, 132, 147, 149, 150, 154, 156, 160, 161, 169, 171, 175, 182, 188, 189, 190, 191, 192, 193, 194, 202, 203, 204, 206, 208, 213, 215, 216, 217, 218, 220, 223, 225, 226, 231, 232, 236, 237, 238, 241, 243, 247, 249, 251, 253, 255, 259, 260, 261, 264, 273, 303, 309, 310, 311
- Bricolage, 17, 68, 242
- Bricoleur, 17, 68
- Catalogue-order pattern, 188, 189, 191, 194, 220
- Chen*, 183, 185, 187, 188, 189, 190, 191, 193, 194, 195, 203, 209, 217, 218, 220, 224, 226, 241, 243, 247, 253

- Cherry-picking, 37, 137, 177, 212, 261
- Conceptual design, 4, 7, 59, 60, 73, 124, 133, 149, 152, 202, 203, 206, 210, 221, 237, 275, 276
- Concernful activity, 55
- Constructivism, iii, 5, 20, 38, 56, 57, 58, 61, 75, 79, 94, 97, 234, 240, 272
- Contexts of activity, 78
- Conversation-maker, 82
- Cook*, 119, 122, 125, 128, 142, 143, 152, 156, 157, 158, 166, 167, 168, 172, 173, 174, 236, 243, 244, 254, 259, 302, 303, 306, 307
- Criticalism, iii, 44, 229, 251, 252, 264, 265
- Decision Tree*, 182, 190, 191, 192, 193, 202, 203, 206, 215, 216, 225, 247, 251, 260
- Decision-contract, 241
- Decision-maker, 82
- Deconstruction, 20, 37, 45, 82, 327
- Design act, 28, 30, 35, 42, 45, 57, 62, 70, 74, 79, 81, 85, 90, 91, 95, 98, 107, 115, 116, 122, 123, 124, 126, 140, 141, 170, 174, 177, 179, 180, 202, 203, 204, 209, 212, 215, 221, 222, 230, 231, 232, 233, 237, 240, 242, 243, 244, 245, 247, 248, 259, 260, 261, 262, 264, 267, 268, 269, 270, 271, 272, 273, 275, 287, 317, 326, 330, 331, 333, 334
- Design engagement, 275
- Design episode, 196, 224, 275
- Design method, iii, 1, 4, 5, 15, 17, 18, 30, 37, 38, 39, 45, 68, 98, 138, 139, 140, 149, 195, 203, 212, 228, 233, 253, 261, 267, 268, 275, 325
- Design momentum, 203, 234
- Design, software, 58, 94, 130, 238, 258, 264, 286, 330
- Designer- scientist, 74
- Designer-artist, 74
- Dialectically constituted activity, 78
- Discourse, 9, 17, 24, 35, 45, 60, 92, 94, 209, 253, 254, 274
- Eames*, 116, 133, 134, 207, 209, 232, 236, 237, 250, 251, 267
- Ego, 146
- Emergence, 19, 35, 65, 69, 86, 97, 111, 130, 147, 148, 159, 163, 221, 237, 259, 262, 275, 303, 320, 330
- Enlightenment, 20, 24, 48, 49, 52, 53, 74, 99
- Epistemology, 17, 47, 61, 76, 87, 98, 268, 274
- Ethnography, 75, 88, 207, 272, 328
- Exchange model, 239
- External method, 260, 275, 276
- Extreme programming, 130
- Feature-Driven Development, 248

- First case study, 182, 209, 211, 213, 217, 218, 220, 223, 224, 225, 237, 241, 251, 264
- Gang of Four, 168, 219
- Goal-directed, 33, 48, 80
- Goal-forming, 80
- Goldberg*, 183, 185, 187, 189, 191, 194, 195, 203, 218, 220, 224, 241, 247
- Griffin*, 116, 130, 145, 236, 267
- Gropius*, 118, 119, 120, 121, 129, 134, 136, 150, 157, 158, 206, 233, 234, 236, 245, 261, 273
- Grounded theory, 1, 14, 88, 99, 103, 116, 180, 182, 195
- Hammer, 'hammeriness', 55, 170, 245, 309
- Hermeneutics, 49, 78, 99, 112
- Howard*, 130, 149, 153, 157, 158, 172, 236, 308
- Interpretivism, 101
- Introduced complexity, 146, 194, 214, 226
- Johnson*, 117, 158, 165, 171, 172, 174, 236, 273, 304, 308
- Kahn*, 115, 117, 127, 131, 132, 135, 136, 140, 141, 143, 160, 233, 236
- Knowing-in-action, 80
- Knowledge container, 22
- Knowledge conveyor, 22
- Layering, 12, 16, 71, 127, 144, 154, 155, 160, 164, 167, 260, 261, 274, 324, 325
- Le Corbusier*, 25, 27, 63, 117, 121, 131, 134, 140, 146, 153, 157, 167, 195, 196, 197, 198, 200, 201, 202, 203, 204, 208, 209, 211, 216, 217, 218, 225, 236, 239, 249, 259, 260, 273, 320
- Lethaby*, 117, 128, 131, 132, 160, 163, 170, 208, 216, 244, 247, 249, 267, 305, 309
- Levittown, 25, 26, 57
- Mackintosh*, 116, 119, 127, 128, 131, 133, 134, 136, 137, 146, 153, 155, 163, 165, 166, 195, 196, 197, 198, 199, 200, 201, 203, 208, 218, 231, 236, 249, 273, 309
- McLuhan*, 20, 34, 35, 36, 39, 129, 146, 151, 154, 170, 171, 172, 175, 259, 267, 308
- Method- assembler, 234
- Method-author, 40, 231, 234
- Method-follower, 231
- Methodology, method, 275
- Methodology-in-action, 95
- Moore*, 117, 123, 127, 128, 134, 144, 149, 154, 159, 167, 168, 236
- Morris*, 123, 128, 131, 147, 148, 150, 151, 153, 155, 159, 170, 171, 173, 174, 215, 235, 236, 239, 247, 302, 303, 305
- Necessary complexity, 146, 150, 214
- Negotiation, 119, 143, 157, 176, 178, 184, 208, 209, 210, 211, 220, 229, 230, 231, 240,

- 243, 253, 254, 269, 325
- Nygaard*, 94, 183, 185, 187, 189, 191, 194, 195, 203, 218, 220, 224, 241, 247
- Object-oriented, 6, 7, 9, 11, 24, 30, 46, 50, 97, 104, 107, 113, 121, 122, 125, 127, 131, 139, 140, 147, 148, 153, 154, 163, 165, 183, 195, 196, 197, 198, 199, 200, 201, 208, 209, 211, 212, 220, 221, 226, 245, 272, 275, 284, 289, 290, 294, 295, 297, 298, 299, 300, 301, 327, 329, 333, 337
- Ontology, 17, 57, 60, 79, 88, 91, 142, 147, 148, 170, 178, 209, 221, 267
- Open system, 64, 65, 230
- Paradigm bias, 152, 153, 154, 194, 201, 217
- Pattern, 68, 275, 321, 327, 328, 329, 338
- Pattern language, 30, 85, 321, 327
- Pedagogy, 98, 272
- Personal method, 262, 276
- Personal pattern, 167, 168, 169, 178, 216, 218, 219, 220, 221, 242, 245, 246, 247, 259, 276
- Persons-acting, 78
- Perspective, 161, 253, 276
- Perspective-bias, 195
- Perspective-shifting, 152, 171, 217, 218, 254, 263, 264
- Perspectivism, 54, 97
- Phenomenology, 55, 101
- Piano*, 117, 119, 122, 145, 154, 155, 160, 171, 236, 267
- Positivism, 34, 49, 51, 79, 100, 125, 272
- Post hoc* rationalisation, 33, 38, 44, 104, 251, 260
- Postmodernism, 17, 26, 28, 35, 39, 45, 49, 78, 97
- Pragmatism, iii, 21, 30, 31, 34, 38, 43, 76, 203, 240, 241, 242, 244, 245, 247, 248, 325
- Private method, 276
- Problem-solution transparency, 178, 214, 217
- Pruitt-Igoe, 26, 27, 28, 57, 83
- Public method, 261, 276
- Pugin*, 122, 125, 128, 134, 137, 141, 145, 154, 156, 157
- Rationalism, 14, 21, 22, 39, 229, 230, 231, 232, 233, 237, 239
- Refactoring, 171, 225
- Renaissance, 238
- Romanticism, 26, 48, 49, 53, 58, 74
- Routine design, 123, 124, 126, 149, 204, 222, 230, 239, 276
- Ruskin*, 117, 147, 156, 236, 303
- Sashimi, 84, 85

Scrum, 85, 86  
 Second case study, 195, 203, 208, 209, 211, 217, 218, 239, 259  
 Situated action, 11, 31, 32, 33, 34, 40, 75, 79, 80, 81, 226, 234, 257, 258, 259, 269, 272, 326, 330, 332  
 Situated cognition, 5, 31, 75, 76, 77, 78, 79, 81, 82, 106  
 Situatedness, 5, 42, 75, 76, 81, 82, 97, 244, 258, 260, 261, 273, 326, 327, 328, 330  
 Social conscience, 250, 252  
 Social engineering, 44, 250  
 Social relativism, 78  
 Soft Systems, 18  
 Spiral, 67, 68, 84  
*Stickleley*, 117, 139, 140, 151, 152, 162, 164, 169, 208, 209, 232, 236, 242, 243  
 Structure-fitting, 220  
*Sullivan*, 117, 123, 125, 148, 168, 169, 171, 216, 236, 244, 258, 267, 273, 302, 309  
 Symbolist, 19, 76  
 Synthesis, 4, 21, 74, 110, 127, 226, 258, 263, 287, 331, 336  
 System, 166, 290, 300, 320  
 Technician-architect, 40  
 Trajectory, design, 81, 171, 172, 209, 229, 234, 236, 239, 247, 255, 261, 262, 263, 267, 268, 269, 274, 275  
 Trukese, 32, 270  
 unité jardin verticale, 27, 239  
*Utzon*, 119, 121, 122, 123, 124, 125, 126, 127, 128, 129, 134, 138, 149, 156, 171, 172, 173, 174, 232, 233, 237, 238, 243, 244, 248, 260, 273, 304, 305, 306  
*Van der Rohe*, 117, 121, 128, 129, 135, 137, 146, 148, 150, 162, 211, 236  
 Vendor-architect, 249  
 Vernacularism, 13, 47, 65, 66, 67, 68, 69, 70, 140, 233, 242, 248  
 Viability, 74, 182, 207, 218, 241, 270  
*Voysey*, 117, 128, 134, 145, 250  
 Waterfall, 24, 27, 73, 82, 83, 84, 85, 97, 98, 130  
 Web of Decisions, 234  
 Weltanschauung, 88, 276  
 Whirlpool, 27, 84  
 Whole person in action, 31, 78  
 Wicked problem, 65  
 Workplace reform, 44, 250