A Systematic Approach to Design Domain-Specific Software Architectures

Eduardo S. Almeida, Alexandre Alvaro, Vinicius C. Garcia, Leandro Nascimento, Silvio L. Meira
Federal University of Pernambuco and C.E.S.A.R. - Recife Center for Advanced Studies and Systems, Recife, Brazil
Email: {esa2,aa2,vcg,lmn,srlm}@cin.ufpe.br

Daniel Lucrédio
Institute of Mathematical and Computing Sciences, University of São Paulo, São Carlos, Brazil
Email: lucredio@icmc.usp.br

***Abstract*— Since the first works involving software reuse, domain engineering is considered a key process to develop reusable and flexible software. However, the results have shown that there is still much to do before the vision of domain engineering is completely achieved. Among the reasons for this problem, we may highlight the lack of a process to support the design of domain-specific software architectures. This paper presents such an approach, based on a well defined set of principles, guidelines and metrics. We also present an experimental study that evaluated the viability of applying the approach in domain engineering projects.**

***Index Terms*— software reuse, domain engineering, domain analysis, software architecture, experimental study**

I. Introduction

Software reuse is a key factor for improving quality and productivity [1]. However, this process is more effective when systematically planned and managed in the context of a specific domain, where application families share some functionality. In this scenario, Domain Engineering (DE) [2] has been seen as a facilitator [3]–[6] to obtain the desired benefits.

The DE life cycle encompasses three important sub-processes: Domain Analysis, Domain Design and Domain Implementation. Among these, the Domain Design stands out as a crucial point [7]. The key goal of the domain design is to produce the domain-specific architecture, defining its main elements and their interconnections [7]. The elements determine the static and dynamic decomposition that is valid for all applications in the domain. According to Tracz [8], a Domain-Specific Software Architecture (DSSA) is defined as an assemblage of software components, specialized for a particular type of task (domain), generalized for effective use across that domain, composed in a standardized structure effective for building successful applications.

This paper is based on "Designing Domain-Specific Software Architecture (DSSA): Towards a New Approach," by E.S. Almeida, A. Alvaro, V.C. Garcia, L. Nascimento, D. Lucrédio and S.L. Meira which appeared in the Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA), Mumbai, India, 2007. © 2007 IEEE.

Even being pointed since the late 1980's as an important issue for reusing software [9], the software architecture field, with its design methods, does not address all the particularities of DSSA design. This problem is also identified in software reuse processes [10] which present gaps in important tasks.

In this context, we developed an approach to design domain-specific software architecture, aiming to solve the problems found in the field. In a previous work [11], we presented an overview of the approach, briefly describing its main activities. This paper makes two novel contributions:

- It further describes how each activity is performed, including more details about how to use the approach in real scenarios; and
- It describes the results of an experimental study that evaluated the viability of the approach to design domain-specific software architectures.

The remainder of this paper is structured as follows: section II presents some related works, section III presents the details of the approach, including the activities and guidelines, section IV presents the results of the experimental study that evaluated the approach, and section V presents some concluding remarks.

II. Related Work

Researchers and practitioners have proposed some works related to software architecture design methods, such as ADD [12], CBAM [12], DAGAR [13] and functionality-based design [7]. However, except for the third one, these methods are more appropriated to single systems development, where focus is not explicit on reuse [14]. DAGAR, although defined to work with reuse, has some drawbacks, such as to be focused on Ada, and too generic to be used nowadays. Our approach was inspired mainly by Tracz's work [8], but with more details, guidelines, tasks, and roles in order to systematize the process.

III. A Systematic Approach for DSSA

Our DSSA design approach consists of seven steps, presented in the next sections. The approach is influenced by several works from the literature, such as ADD [12], UML Components [15], and the weak and strong points from reuse processes [10].

A. Decompose Module

The first activity in the approach corresponds to an abstraction and decomposition phase. Initially, the domain architect - an experienced person who conducts the domain design process - based on the assets produced in the domain analysis [16] (business goals, constraints, domain use case model, feature model, and scenarios), chooses the domain architecture modules to decompose. The modules to start are usually the whole applications in the domain, which are further decomposed into subsystems, and sub-modules. According to Parnas [17], the benefits expected of modular decomposition are: **i. managerial** - development time should be shortened because separate groups can work on each module with little need for communication; **ii. flexibility** - it should be possible to make drastic changes to one module without needing to change others; **iii. comprehensibility** - it should be possible to study the system one module at a time.

In this activity, the domain architect interacts with the domain analyst, in order to obtain detailed information about the domain assets (mainly domain use case model, feature model and scenarios), and the project manager, to discuss the business goals and constraints that may have influence on the architectural design.

In our approach, there is not a set of criteria to be used in module decomposition, such as in Parnas' work, nor a set of rule of thumbs. However, based on the state-of-the-art and practical experience, we consider that the following issues should be balanced by the domain architect: *availability, coupling, extensibility, flexibility, functionality, information hiding, maintainability, modifiability, performance, separation of concerns, scalability, security, and usability*.

B. Refine Module

The module refinement is an iterative process that can be divided into three sub-activities, presented next.

1) Choose the Architectural Drivers: According to Bass et al. [12], architectural drivers are the combination of functional and quality requirements that "shape" the architecture or the particular module under consideration. The drivers are found among the top-priority requirements for the module.

The determination of architectural drivers is not always a top-down process. Sometimes detailed investigation is required in order to understand the ramifications of particular requirements. We base the module decomposition on the architectural drivers, because when choosing the drivers, we are reducing the problem to satisfy the most important ones. In our approach, the architectural drivers are the requirements expressed by the feature model, the quality attributes and the scenarios, if applicable.

2) Choose the Architectural Patterns: After identifying and choosing the relevant architectural drivers for each module, the domain architect selects the architectural patterns that can be applied. The patterns satisfy the architectural drivers and are constructed by composing selected tactics. Two factors guide tactic selection. The first is the drivers themselves, and the second is the side effects that a pattern implementing a tactic has on others. In our approach, the vision of a tactic agrees with Bass et al.'s, who define it as a design decision that influences the control of a quality attribute response. A tactic example can be one related to modifiability in order to prevent ripple effects, i.e., changes made in a module affecting other ones. Thus, some possible tactics can be information hiding, restrict communication paths, and the use of facades [18].

3) Allocate Functionality using Views: In the previous sub-activities, the approach showed how the architectural drivers determine the decomposition structure of a module via the use of tactics. In this sub-activity, the goal is to define how the modules can be instantiated. The criteria for allocating functionality is similar to that used in functionality-based design methods, such as most object-oriented design methods, but with a variation to treat features. Thus, two design guidelines (DG) should be considered to allocate functionality:

DG1. *allocating functionality based on use cases*. Applying use cases that pertain to the parent module aids the domain architect to obtain a more detailed understanding of the functionality distribution. Thus, every use case of the parent module must be represented by a sequence of responsibilities within the child modules. Assigning responsibilities to the children during the decomposition leads to the discovery of possible needs for information exchange. This creates a relationship among the modules that needs to be considered. However, it is not important to define how the information is exchanged yet. These questions are dealt with later in the design process.

DG2. *allocating functionality based on features*. During the domain analysis, a set of common features and its variability have been identified and represented through the feature model. This information is very important to design a flexible architecture, adaptable for several applications, and thus the process of allocating functionality should consider it. In this way, in the approach, two features F1 and F2 are related if the following conditions hold:

- if they belong to a same functional category as defined by clients. This functional category may be based on system, module and functional classification;
- if they use common data or information;
- if there is some strong degree of dependency among the features; and
- if they belong to the system layer and they process related system operations or transactions.

These activities should be sufficient to gain confidence that the future applications in the domain will deliver the desired functionality. However, to check if the requirements (derived from domain analysis) can be met, the approach needs more than just to allocate responsibilities. Thus, after allocating the functionality, the domain architect represents the architecture with views. According to Bass et al. [12], a view is a representation of a coherent set of architectural elements, as written by and read by system stakeholders. It consists of a representation of a set of elements and the relations among them.

Thus, one view from each of the three major view groups [12] (module, concurrency, and deployment) can be used. After concluding this sub-activity and obtaining the initial domain architecture, the domain architect starts a more detailed process of designing each module with its variability representation.

C. Represent Variability

According to Svahnberg et al. [19], variability is the ability to change or customize a system. Improving variability in a system implies makes it easier to do certain kinds of changes. Moreover, it is possible to anticipate some types of variability and construct a system in such a way that it is prepared for inserting predetermined changes. In DSSA design, the need for variability is even bigger, since flexibility is a crucial requirement. However, even with about thirteen approaches available in the literature (Aggregation/delegation, Inheritance, Parameterization, Overloading, Delphi properties, Dynamic Class Loading, Static Libraries, Dynamic Link Libraries, Conditional Compilation, Frames, Reflection, Aspect-oriented programming, and Design Patterns) [20], in general, software architects do not have effective ways to do it [21]. In our approach, Design Patterns [18] are used, but together with useful guidelines that determine how and when patterns can be used to represent the different kinds of variability that can exist in a DSSA. Keepence & Mannion [22] and Lee & Kang [23] also use design patterns. However, their approaches just present some patterns that can be used, without discussing useful guidelines of how and why each pattern should be used for each situation.

Thus, in order to design the variability of each module, we consider that it should be traceable from domain analysis assets (features) to the architecture, according to *alternative*, *or* and *optional* features [23].

1) Alternative Features: Alternative features indicate a set of features, from which *only one* must be present in an application. Thus, the following set of patterns can be used [18]:

Abstract Factory and Singleton. The *abstract factory* pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. Thus, it can be used in the following way: one *abstract factory* acts as interface for the remainder of the architecture, while in one application, just one concrete factory will exist for one of the alternative features. The *singleton* pattern ensures that a class has only one instance, and provides a global point of access to it. In this way, this pattern can be used to assure that only one feature can be present in the application.

Factory Method. The *factory* pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. This pattern is similar to the *abstract factory* and can be used also for alternative features.

Prototype and Singleton. The *prototype* pattern specifies the kinds of objects to create using a prototypical instance, and creates new objects by copying this prototype. In this pattern, the prototype specifies how the interaction with the feature should be, by defining a concrete prototype for each feature. Thus, depending on which feature will be in the application, just one concrete prototype will be used.

Strategy. The *strategy* pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Using this pattern, an abstract strategy represents the interaction with the remainder of the architecture, and only one concrete strategy will be present. It is similar to the factory method, but works in the behavior level.

Template Method. The *template method* defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. This pattern is close to *strategy*; however, it forces the implementation to follow some pre-defined steps. It is also useful when the interaction between the alternative feature and the remainder of the architecture needs a more intensive communication.

2) Or Features: Or features represent a set of features, from which *at least one* must be present in an application. For this type of feature, two patterns can be used [18]:

Builder. The *builder* pattern separates the construction of a complex object from its representation, so that the same construction process can create different implementations. This pattern can be used to build composed features. Thus, for the remainder of the architecture, only the *Director* is available, being responsible for deciding which features will be in the application and which will not. For example, the *Director* can be responsible for inserting at least one of the features.

Observer. The *observer* pattern defines a one-to-many dependency between objects so that when one object changes its state, all of its dependents are notified and updated automatically. Using this pattern, features can be added to the application as a plug-in, after the deployment.

3) Optional Features: Optional features are features that *may or may not* be present in an application. For this type of feature, three patterns can be used [18]:

Builder. In optional features, this pattern can decide if a feature will be present in an application or not.

Decorator. The *decorator* pattern can be used for optional features, mainly those that are additional features. Thus, if a feature is present, the *ConcreteDecorator* is responsible for managing and calling the execution. For

example, in a simple search system, after returning the results, nothing is done, but, if a feature to learn with the user is present in the application, after returning the result, one *Decorator* can perform the tasks of storing the search words and the obtained results, making a user profile.

Observer. This pattern can be used in the same way as in *or features*.

4) Guidelines for using Patterns: Seeking a more detailed process, the following additional guidelines are defined for each type of feature:

Alternative features. When a feature can be directly mapped into a single class, we suggest the use of the *Prototype* pattern, because it is simpler and allows to instantiate a specific object, depending on the feature that is used in the application, through simple inheritance. Another suggestion is to use the *Singleton* pattern to keep and manage a unique instance of this class.

However, there could be cases where one feature can only be implemented by more than one class. In this case, it is recommended to use the *Abstract Factory* pattern or *Factory Method*, which allows, through inheritance of the factory, to build more than one class (*ConcreteProduct*) for the same feature. The difference between them is that in the *Abstract Factory* there is a separate class for instantiating the objects, while in the *Factory Method* the instantiation of the objects is incorporated into another class of the architecture. The *Factory Method* is simpler, and should be preferably used. However, it impedes inheritance to be used in the class that encapsulates it. Thus, the *Abstract Factory* should be used if inheritance is to be allowed.

In this case, the *Singleton* pattern can also be used, but together with the *Facade* pattern, in a way that all classes that are related to the feature are united behind a single class. In order to facilitate the construction of the features that require more than one class, together with *Abstract Factory* or *Factory Method*, the *Builder* pattern, which facilitates object construction, is recommended.

To map the behavior, we suggest the utilization of the *Strategy* and *Template Method* patterns, which, through simple inheritance, use only one of the subclasses as the implementation for the behavior.

Alternative features were not common in the starship game domain, however, we believe that the discussed guidelines were defined in a reasonable way and can be useful in other application domains.

Or features. In order to assure that at least one feature is present, and to manage and determine which features are present, we suggest the use of the *Builder* pattern, with the *Director* being responsible for managing these aspects. The *Composite* pattern can also be used to manage the features composition structure, providing the application discovery mechanisms, to find out which features are present and which are not.

The *Adapter* and *Bridge* patterns can also be used to separate the features from the remainder of the architecture. Thus, depending on the number of features that are present, a different adapter or bridge may be used. The difference is that the *Adapter* is more generic, and can be used to unite several classes under a single interface, where the signatures of the method implementations can be different from the specification. The *Bridge* is more appropriate when a single class is needed, and the signatures of the methods implementations are identical to the specification.

The *Decorator* and *Chain of Responsibility* patterns may be used when different features have functionalities that are complementary to each other, so that one feature must execute after the other. The *Decorator* is a structural pattern, more indicated for the cases where the interaction between several features is complex, but well defined, because in these cases, the structure of the additional functionalities must be defined during design time. The *Chain of Responsibility* is a behavioral pattern, i.e., the structure of the interaction is not well defined, and thus it is more indicated for the request-response model. If the interaction between the features is simpler, we suggest the use of *Chain of Responsibility*, preferably together with the *Command* pattern to represent the request.

Optional features. For optional features, the same set of patterns used for the *Or features* can be used, with the difference that in this case it is not necessary to guarantee that at least one feature is present in the application.

After decomposing the modules, refining them, and representing their internal variability, the domain architect starts to define architectural components. In order to do it, three activities are performed: Group Components, Identify Components, and Specify Components. These are described next.

D. Group Components

This activity is composed of four steps: *Measure Functional Dependency*, *Cluster Use Cases*, *Allocate Classes to Components*, and *Select Candidate Components*. There are other works on component identification and grouping [24]–[27]. However, most of them emphasize the principle of functional independence and do not provide systematic steps, metrics and guidelines. In this activity, our work is close to [26] (two first activities), with some improvements that will be described in detail in the next sections; and [27] in the sense of defining steps, metrics, guidelines and roles.

1) Measure Functional Dependency: In this sub-activity, the domain architect, in conjunction with the project manager, assesses the functional dependencies between use cases and cluster closely related use cases into a component. In order to do it, the following Criteria (C) are used to measure the functional dependency between two use cases, UC_i and UC_j:

C_1. *Criterion concerning Sub-Systems*. If both UC_i and UC_j belong to the same sub-system, the measure for this criterion M_{11} gets the value of 1. Otherwise M_{11} gets 0. If the target system is so small that sub-systems are not defined, then M_{11} also gets 1.

C_2. *Criterion concerning Actors*. Use cases initiated or invoked by the same actor are more closely related than

the ones that are not. Thus, measuring this relationship, we should consider that a use case may be initiated by more than one actor. The measure for this criterion, M_{12}, is defined as the following:

$$M_{12} = \frac{(actors\ initiating\ UC_i \cap actors\ initiating\ UC_j)}{(actors\ initiating\ UC_i \cup actors\ initiating\ UC_j)}$$

In the metric, M_{12} measures the proportion of actors that initiate both UC_i and UC_j against the set of all related actors. A higher value of M_{12} indicates a higher degree of use cases sharing the same actor (s). The range of M_{12} is 0..1. If M_{12} is 1, UC_i and UC_j have same actor (s), but if M_{12} is 0, then there is no actor initiating both use cases.

C_3. *Criterion concerning Shared Data*. Use cases manipulating the same set of data are more closely related than other use cases. Thus, the degree of commonality on data manipulated by two use cases M_{13} can be more accurately expressed using a metric rather than simply representing it as a Boolean value or a scale. The measure for this criterion is defined as the following:

$$M_{13} = \frac{(classes\ accessed\ by\ UC_i \cap classes\ accessed\ by\ UC_j)}{(classes\ accessed\ by\ UC_i \cup classes\ accessed\ by\ UC_j)}$$

The range, minimum value and maximum value of this metric are same as those for M_{12}.

C_4. *Criterion concerning Use Case Relationship*. A measure, M_{14}, represents the degree of coupling between two use cases. Three types of relationships among use cases exist: ≪*include*≫, ≪*extend*≫ and generalization. If UC_i and UC_j are specialized from a generalized use case, then M_{14} is 1. If UC_i and UC_j are related with ≪*extend*≫, then M_{14} gets 1. If UC_i includes UC_j with ≪*include*≫, then the relationships between UC_j and use cases other than UC_i should also be considered. For example, UC_j may also be included by UC_k. In this case, if UC_i has a relatively strong relationship with UC_k, then M_{14} gets a value of 1. If UC_i is independent from UC_k, UC_j should be placed in a separate component.

Each criterion may be assigned with a different weight value, and the sum of all weights must be 1. Based on the given criteria, we use a metric [26] to compute the functional dependency between two use cases UC_i and UC_j, FD_{ij}, as follows: $FD_{ij} = \sum(M_k * W_k)$, where M_k is the measure on the k^{th} criterion for two use cases, and W_k is the weight value for the k^{th} criterion. The range of FD_{ij} is between 0 and 1.

2) Cluster Use Cases: After computing the functional dependencies for each pair of use cases, the domain architect defines candidate components by clustering related use cases. In order to register it, the use cases are organized into a worksheet, where the dependencies are established and calculated, such as in Table I.

The clustering algorithm used for this task uses a row and column shifting method, and requires a constant value *t* to identify rows and columns to be shifted. For a given value of *t*, the algorithm performs the best clustering possible. As *t* increases, smaller grained components are obtained. However, the value of *t* is chosen from values of functional dependency computed by the domain expert.

	UC_1	UC_2	UC_3	UC_4	UC_5	UC_6	UC_7
UC_1	1	0.7	0.6	0.2	0.3	0.4	0.5
UC_2	0.7	1	0.7	0.3	0.2	0.5	0.4
UC_3	0.6	0.7	1	0.4	0.2	0.4	0.3
UC_4	0.2	0.3	0.4	1	0.6	0.8	0.9
UC_5	0.3	0.2	0.2	0.6	1	0.7	0.8
UC_6	0.4	0.5	0.4	0.8	0.7	1	0.7
UC_7	0.5	0.4	0.3	0.9	0.8	0.7	1

TABLE I.
FUNCTIONAL DEPENDENCY MATRIX

Different values of *t* can be applied in this step, and different clustering results can be examined by domain experts or developers to select the optimal result.

After calculating the functional dependency and shifting each use case, the next task is to identify the set of adjacent use cases that have FD_{ij} values greater than *t*. Thus, each set of clustered use cases makes up a candidate component. Table I shows an example of this activity, where the domain architect defines a *t* value equals 0.6, obtaining two candidate components (shaded in the table): the first, including use cases UC_1, UC_2, UC_3 and the second composed of UC_4, UC_5, UC_6, UC_7.

A possible problem during clustering occurs when we can not have a clear-cut clustering. A use case can be shifted several times if the use case has more than one functional dependency FD_{ij} that is greater than *t*. This may yield a clustering result with use cases shared by sets (S), and as the result, clear-cut clustering is not possible. This situation is shown in Figure 1 [26].

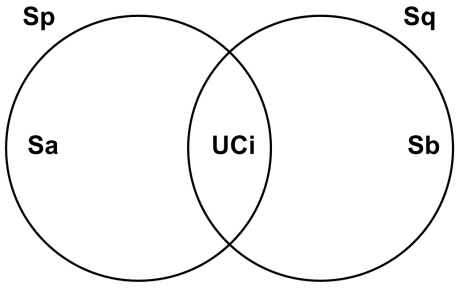

Figure 1. Relationship among Sets of Use Cases

In the Figure, S_p and S_q are clustered sets of use cases. S_a and S_b are sets of use cases included exclusively into S_p and S_q, respectively. A use case UC_i is the intersection of S_a and S_b. That is, S_p is the union of S_a and UC_i, and S_q is also a union of S_b and UC_i.

We may focus on the relationship between S_a, S_b and UC_i to derive a difference as follows; a degree of coupling between a set and a use case is defined as $Coupling(S_a, UC_i)$; If UC_j is an element of S_a, $Coupling(S_a, UC_i)$ is $\sum FD_{ij}$ for all *j*. We calculate the difference between $Coupling(S_a, UC_i)$ and $Coupling(S_b, UC_i)$. If the difference is greater than 0, UC_i should be included in S_p. If the difference is less than 0, UC_i should be included in S_q. If the difference is equal to 0, then the use case may be arbitrarily allocated, or allocated by domain experts.

3) Allocate Classes to Components: After performing the clustering process, the domain architect starts to allocate classes to each component using the dynamic model expressed by sequence diagrams. In our approach, a component includes several use cases. The dynamic behavior of each use case is depicted by a sequence diagram, which specifies a set of participating objects/classes.

Thus, for each component identified in the previous step, the domain architect locates sequence diagrams for use cases included in the component. Next, the participating classes that are shown in these diagrams are assigned to the corresponding component. In order to register it, a worksheet matrix can be used, relating classes and their associated components. However, sometimes the assignment of classes into components may yield a conflict where a class is assigned to more than one component.

For the conflicts, we consider two cases: the *first one* is the dynamic model. In this case, we first compute an entity dependency between a component and a class, and then compare the entity dependency between one conflicting class from one component and one from another component. In order to compute the entity dependency, we used the following criteria:

i. Coupling and cohesion. Initially, the coupling and cohesion can be computed for the classes in order to solve the conflicts.

ii. The number of messages. After analyzing coupling and cohesion, the number of messages being passed can be considered. Through tracing a sequence diagram, we can estimate the number of dynamic messages that are exchanged between classes for each use case during runtime. This criterion, M_{21}, is computed as following:

$M_{21} = \sum msg(C_i, Com_j)$

where $msg(C_i, Com_j)$ is the number of messages passed to class C_i in the sequence diagrams of all use cases associated to the conflicting component Com_j.

iii. Relationship between classes. In general, there are fives types of relationship among two classes: *inheritance*, *association*, *dependency*, *aggregation* and *composition*. Instead of defining constant values for the weights, we define the following comparison on different relationships: 0 < Dependency < Aggregation < Association < Inheritance < Composition < 1. Using this weight values, we can compute the inter-class relationship between a pair of classes, defining a constant value *t*, to determine whether two classes should be put together in a component or not:

$M_{22} = \sum wg(C_i, Com_j)$

where wg measures the weight of the relationship between C_i and other classes in Com_j.

From the criteria, we use a formula [25] to compute the entity dependency $ED_{ij} = (M_k * W_k)$ between a class C_i and a component Com_j.

The *second case* to treat conflict is based on static relationship. In this case, the strength of the static relationship (S_{ij}) is defined as:

$S_{ij} = W_s * N_{ij}$, where W_s = the static association weight (0 < Dependency < Aggregation < Association < Inheritance < Composition < 1), N_{ij} = the number of associations between classes *i* and *j*. The S_{ij} range is scaled from 0 to 1. Thus, based on static relationship, the domain architect can decide which classes will be allocated to each component.

4) Select Candidate Components: Following the previous steps, the domain architect, in conjunction with the project manager, identifies candidate components. The value of *t* used in the process defines the number of components and their granularity. Thus, it is recommended to apply different values of *t* to generate different clustering results and to let architects and project managers choose an optimal clustering result using the criteria. In this sense, the goal of this step is to choose an optimal configuration from different clustering results generated by applying different *t* values.

As defined previously, the range of *t* is between the minimum and maximum values of all measured functional dependencies, i.e. from 0.1 to 0.9. Thus, a smaller value of *t* produces components with large granularity; on the other hand, the total number of components is decreased. In contrast, a larger value of *t* produces finer grained components and the total number of components is increased.

Additionally, two other criteria can be used to select the candidate components: *costs* - the project manager can decide which components will be specified according to a predefined cost (man/hour) - and *complexity* - as the components are based on use cases, and the use cases have a effort metric estimated, the project manager can select components according to their complexity (low, medium, high).

E. Identify Components

Once the component grouping sub-activity is finished, the domain architect refines the identified components. This refinement consists in analyzing the set of components according to the initial specifications (feature models, domain use case model) in order to assure that the identified components match their specifications.

F. Specify Component

In this activity, the goal is to create an initial set of interfaces and component specifications. This activity is composed of three steps: *Identify Interfaces*, *Identify Core Classes*, and *Refine the Specification*, presented next.

1) Identify Interfaces: In our approach, we consider two types of interfaces: *system* and *business*. An interface is a set of operations, with each operation defining some services or function that the component will perform for the client [28]. The system interfaces and their operations emerge from a consideration of the feature model and mainly of the use case model. This interface is focused on, and derived from, system interactions.

Thus, in order to identify system interfaces for the components, the domain architect uses the following approach: for each use case, he considers whether or not there are system responsibilities that must be modeled.

If so, they are represented as one or more operations of the interfaces (just signatures). This gives an initial set of interfaces and operations.

The business interfaces are abstractions of the information that must be managed by components. Our process for identifying them is the following: *to analyze the feature model to identify classes (for each module and component)*; *to represent the classes based on features with attributes and multiplicity*; and *to refine the business rules using formal language*. Sometimes, depending of the process that was performed by the domain architect in the previous steps, the process to identify business interfaces becomes just a refinement step.

Additionally, for each operation defined in the interfaces, the pre- and post- conditions can be specified using a formal language such as Object Constraint Language (OCL) [29].

2) Identify Core Classes and Refine the Specification: After identifying the interfaces, the domain architect decides which classes from each module are in the core [15]. A core class is a business type that has independent existence within the business, characterized by the following: a *business identifier*, usually independent of other identifiers; and *independent existence*, with no mandatory associations with other types.

The purpose of identifying core classes is to start defining which information is dependent on others, and which information can stand alone. It is useful to allocate information responsibilities to interfaces and to specify precise components. According to Szyperski [28]: *"...a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties"*. The general rule is that we create one business interface for each core class; thus, each business interface manages the information represented by the core class.

At the end, for every component that is specified, the domain architect defines which interfaces its realizations must support (provided and required interfaces), and documents the Component Specification Template (CST). The CST is a template that comprehends the component specification, consisting of: *component name*, *brief description*, *workflow for each use case*, *contained classes*, *use cases and requirements referred*, *provided and required interfaces*, and *commonality and variability represented by features*.

G. Represent Domain Architecture

Once the component specification is performed, the domain architect represents the initial domain architecture based on components. In order to do it, architectural views and component diagrams are used to show the components, their interconnection, and the provided and required interfaces. During this activity, the domain architect can discover and refine other components, using, for example, collaboration diagrams.

IV. Evaluation

We conducted an experimental study in order to evaluate the viability of the approach in domain engineering projects. According to Wohlin et al. [30], the experimental process can be divided into the following main activities: the **definition** is the first step, where the experiment is defined in terms of problem, objective and goals. The **planning** comes next, where the design of the experiment is determined, the instrumentation is considered and the threats to the experiment are evaluated. The **operation** of the experiment follows from the design. In the operational phase, measurements are collected, analyzed and evaluated in the **analysis** and **interpretation**. Finally, the results are presented and packaged in the **presentation** and **package**.

The experimental plan presented here follows the model proposed by Wohlin et al. [30]. Additionally, the experiment defined by Barros [31] was also used as inspiration. The definition and planning activities will be described in future tense, showing the logic sequence between the planning and operation.

A. The Definition

In order to define the experiment, the GQM paradigm [32] was used. The GQM is based upon the assumption that for an organization to measure in a purposeful way it must first specify the goals for itself and its projects, then it must trace those goals to the data that are intended to define those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals.

1) Goal: G_1. To analyze the domain design approach for the purpose of evaluating it with respect to the efficiency and difficulties of its use from the point of view of the researcher in the context of domain engineering projects.

2) Questions: Q_1. Does the domain design approach generate a domain architecture with low instability? Q_2. Does the domain design approach generate a domain architecture with high maintainability? Q_3. Does the domain design approach generate components with low complexity? Q_4. Do the subjects have difficulties to apply the approach?

3) Metrics: M_1. *Module Stability*. According to Martin [33], what it is that makes a design rigid, fragile and difficult to reuse is the interdependency of its modules. A design is rigid if it cannot be easily changed, i.e., a single change begins a cascade of changes of independent modules. Moreover, when the extent of change cannot be predicted by the designer, the impact of the change cannot be estimated. It makes difficult the cost of the changes and in this way, the design becomes rigid. Thus, the *Instability* (I) metric will measure the module stability in order to assess it. This metric is defined as [33]:

$I = (C_e(C_a + C_e))$, where

C_a: *Afferent Coupling*, i.e. the number of classes outside this category that depend upon classes within this category; and

C_e: *Efferent Coupling*, i.e. the number of classes inside this category that depend upon classes outside this category.

The instability metric has range [0, 1], where I = 0 indicates a maximally stable category and I = 1 indicates a maximally instable category.

M_2. *Module Maintainability*. Maintainability is desirable both as an instantaneous measure and as a predictor of maintainability over time. Efforts to measure and track maintainability are intended to aid reducing or reversing a system's tendency toward "code entropy" or degraded integrity, and to indicate when it becomes cheaper and/or less risky to redesign/rewrite than to change it [34]. In domain engineering projects, this issue is very important since a domain is continuously in evolution with new features or products being developed. Thus, the *Maintainability Index* (MI) metric will measure a module's maintainability in order to assess it. This metric is defined as [34]:

$MI = 171 - 5.2 * ln(aveV) - 0.23 * aveV(g') - 16.2 * ln(aveLOC) + 50 * sin(sqrt(2.4 * perCM))$, where:

aveV = average Halstead Volume V per module;

aveV(g') = average extended cyclomatic complexity per module;

aveLOC = average count of lines of code (LOC) per module; and

perCM = average percent of lines of comments per module.

The metric indicates that modules with a MI less than 65 are difficult to maintain, modules between 65 and 85 have reasonable maintainability and those with MI above 85 have good maintainability.

M_3. *Component Complexity*. A critical issue in software engineering is related to separation of concerns and how to modularize a software system in order to result in modules and components that are well defined, testable and maintainable. During the 70s, McCabe developed a mathematical technique [35] which provides a quantitative basis for modularization and identifying software modules that will be difficult to test or maintain. In this technique, the complexity measure was presented and used to measure and control the number of paths through a program. This measure is also useful to keep the size of the modules manageable and allow the testing of all the independent paths. Thus, the cyclomatic complexity metric will be used to analyze the complexity of each component developed. The cyclomatic complexity of a module is calculated from a connected graph of the module:

$CC(G) = E - N + p$, where:

E = the number of edges of the graph;

N = the number of nodes of the graph; and

P = the number of connected components.

This metric indicates[1] that a unit with cyclomatic complexity between 1 and 10 is a simple program, without much risk. A value between 11 and 20 represents a more complex program, with moderate risk. Values ranging between 21 and 50 represent a complex program, with high risk. Finally, a complexity greater than 50 is an untestable program and presents a high risk.

We decided to use classic Object Orientation metrics to evaluate the approach because they are more well-established after years of experience with case studies and experiments. Reuse-specific metrics, although more suited to this context, still need more experimentation and maturity [36]. Besides, issues such maintainability, stability and complexity have a large influence on the architecture. For example, if a component has low maintainability, it will be probably harder to reuse. So, by measuring these aspects, we are, to some extent, measuring reuse.

M_4. *Difficulty in understanding the approach*. In order to identify possible weaknesses and misunderstanding in the approach and to define improvements, it is necessary to identify and analyze the difficulties found by users using the approach. Thus, the following metric related to difficulty will measure this issue:

D_D: % of subjects that had difficulties in using the domain design approach and the difficulties distribution

Differently from the other metrics, this metric was never used before, and thus there are no well-known values for it. Thus, arbitrary values were chosen, based on practical experience and common sense. We considered that values above 30% for D_D indicate that the approach is too difficult and should be improved.

B. The Planning

After the definition of the experiment, the planning started. The definition determines the foundations and reasonings for the experiment, while the planning prepares for how the experiment is conducted.

1) Context: The domain design approach will be applied in a university laboratory with the requirements defined by the experimental staff based on real-world projects. The study will be conducted as *single object study* which is characterized as being a study which examines an object on a single team and a single project [37].

2) Training: The training of the subjects using the approach will be conducted in a classroom at the university. The training will be divided in two steps: in the first one, concepts related to software reuse, variability, component-based development, domain engineering, software product lines, asset repository, software reuse metrics, and software reuse processes will be explained during eleven lectures with two hours each. Next, the domain design approach will be discussed during three lectures. During the training, the subjects can interrupt to ask issues related to lectures.

3) Pilot Project: Before performing the study, a pilot project will be conducted with the same structure defined in this planning. The pilot project will be performed by a single subject, who will be trained on how to use the proposed approach. For this pilot, the subject will use the same material described in this planning, and will be

[1]Based on research performed in the Software Engineering Institute (SEI).

observed by the responsible researchers. In this way, the pilot project will be a study based on observation, aiming to detect problems and improve the material.

4) Instrumentation: All the subjects will receive a questionnaire (QT1) about his/her education and experience, a set of papers containing an example of use and the steps of the approach. The material also includes a second questionnaire (QT2) for the evaluation of the subjects' satisfaction using the approach.

5) Criteria: The quality focus of the study demands criteria that evaluate the benefits obtained by the use of the approach and the difficulties of the users. The benefits obtained will be evaluated quantitatively through the domain architecture and components, using the stability, maintainability, complexity and difficulty metrics defined earlier. Moreover, the difficulties of the users will also be evaluated using qualitative data from questionnaire QT2.

6) Null Hypothesis: This is the hypothesis that the experimenter wants to reject with as high significance as possible. In this study, the null hypothesis determines that the use of the approach in domain design does not produce benefits that justify its use and that the subjects have difficulties to apply it. Thus, according to the selected criteria, the following hypothesis can be defined:

H_0: μ the approach generates the architecture with $I >= 0.5$ and $MI < 85$

H_0: μ the approach generates components with $CC >= 21$

H_0: μ $D_D >= 30\%$

7) Alternative Hypothesis: This is the hypothesis in favor of which the null hypothesis is rejected. In this study, the alternative hypothesis determines that the use of the approach produces benefits that justify its use. Thus, the following hypothesis can be defined:

H_1: μ the approach generates the architecture with $I < 0.5$ and $MI >= 85$

H_1: μ the approach generates components with $CC < 21$

H_1: μ $D_D < 30\%$

8) Independent Variables: The independent variables are the education and the experience of the subjects, collected through the questionnaire QT1, the proposed approach and the requirements for the project. This information can be used in the analysis for the formation of blocks.

9) Dependent Variables: The dependent variables are the quality of the architecture and components. The quality of the architecture will be measured through its stability and maintainability. The quality of the components will be measured by its complexity. All the variables will be measured using the ratio scale. According to Wohlin et al. [30], if there is a meaningful zero value and the ratio between two measures is meaningful, a ratio scale can be used.

10) Qualitative Analysis: The qualitative analysis aims to evaluate the difficulty of the application of the proposed approach and the quality of the material used in the study. This analysis will be performed through questionnaire QT2. This questionnaire is very important because it will allow evaluating the difficulties that the subjects have with the approach, evaluating the provided material and the training material, and improving these documents in order to replicate the experiment in the future. Moreover, this evaluation is important because it can be verified if the material is influencing the results of the study.

11) Internal Validity: The internal validity of the study is defined as the capacity of a new study to repeat the behavior of the current study, with the same subjects and objects with which it was executed [30]. The internal validity of the study is dependent of the number of subjects. This study is supposed to have at least between seven and eight subjects to guarantee a good internal validity.

12) External Validity: The external validity of the study measures its capability to be affected by the generalization, i.e., the capability to repeat the same study in other research groups [30]. In this study, a possible problem related to the external validity is the subjects' motivation. Since the study will be conducted in a laboratory, some subjects can perform the study without responsibility or without a real interest in performing the project with a good quality as it could happen in an industrial project. However, the external validity of the study is considered sufficient, since it aims to evaluate the viability of the application of the domain design approach. Since the viability is shown, new studies can be planned in order to refine and improve the approach.

13) Construct Validity: The validation of the construction of the study refers to the relation between the theory that is to be proved and the instruments and subjects of the study [30]. In this study, a relatively well known and easily understandable problem domain was chosen to prevent the experienced users in a certain domain to make use of it. Thus, this choice avoids previous experience of making a wrong interpretation of the impact of the proposed approach.

14) Conclusion Validity: This validity is concerned with the relationship between the treatment and the outcome, and determines the capability of the study to generate conclusions [30]. This conclusion will be drawn by the use of descriptive statistic.

C. The Project Used in the Experimental Study

The project used in the experimental study was to produce the architectural design for the starship game domain, using the proposed approach. In order to do it, three games in this domain were presented to the subjects as shows Figure 2.

The subjects had just the executables[2], without any documentation (requirements and design specification, source code, etc). It is important to highlight that the subjects could analyze other games, consult experts, etc, as part of the process.

1) Selection of Subjects:

[2]The games were available on the internet in conjunction with their emulators and documentation.

Figure 2. Applications in the Starship Game Domain

D. The Instrumentation

For the execution of the study, M.Sc. students in Software Engineering from the Federal University of Pernambuco, Brazil, were selected. They were selected by convenience sampling [30] representing a non-random subset from the universe of students from Software Engineering. In convenience sampling, the nearest and most convenient people are selected as subjects.

2) Data Validation: In this study, descriptive statistics was used to analyze the data set, since it may be used before carrying out hypothesis testing, in order to better understand the nature of the data and to identify abnormal or false data points [30].

3) Instrumentation: Before the experiment can be executed, all experiment instruments must be ready. This includes the experiment objects, guidelines, forms and tools. In this study, questionnaires QT1 and QT2, in conjunction with the papers about the approach were used. The questionnaires presented the subjects' names in order to check additional information or misunderstanding. However, the subjects were notified for the information confidentially. Additionally, the subjects could use any tools and environments during the execution of the tasks.

E. The Operation

1) Experimental Environment: The experimental study was conducted during part of a M.Sc. and Ph.D. Course in Software Reuse, from April to September 2006, at Federal University of Pernambuco. The experiment was composed of seven subjects and the project was developed in 355 hours, 23 minutes and 57 seconds. This project, which involved 44 features, produced 79 classes and 5 components.

2) Training: The subjects who used the proposed approach were trained before the study began. The training took 28 hours, divided into 14 lectures with two hours each, during the course.

3) Subjects: The subjects were 7 MS.c. students from the Federal University of Pernambuco. All the subjects had industrial experience in software development (more than one year). Three subjects had participated in industrial projects involving some kind of reuse activity, for instance, component development, framework development, or web services development. All the subjects had previous knowledge about at least one domain analysis process (FODA [38]); three subjects had training in conferences on some issues related to software reuse, such as design patterns and component-based development; and finally, two subjects had co-authored papers involving some aspects of software reuse.

F. The Analysis and Interpretation

1) Instability: After collecting the information about the modules instability, the step of data set reduction started. It is important because errors in the data set can occur either as systematic errors or as outliers, which means that the data point is much larger or much smaller than one could expect looking at the other data points [30]. Figure 3 shows the instability data graphically.

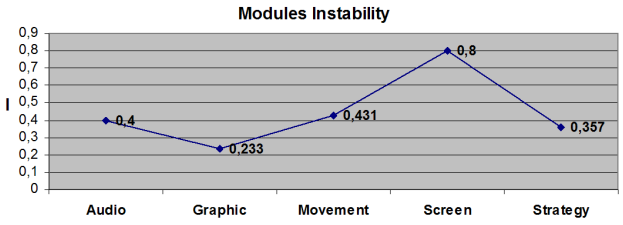

Figure 3. Modules Instability Graphic

As it can be seen in Figure 3, the Graphic and Screen modules presented low (0.233) and high (0.8) values, respectively, when compared with the other data points. Thus, these values could be considered as outliers. In order to analyze this aspect, a box plot graphic can be useful [39], since it is recommended to visualize the dispersion and skewedness of samples. Box plots can be made in different ways [30]. In this thesis, the approach defined by Fenton & Pfleeger [39] was chosen. The main difference among the approaches is how to handle the whiskers. Fenton & Pfleeger proposed to use a value, which is the length of the box, multiplied by 1.5 and added or subtracted from the upper and lower quartiles respectively.

The middle bar in the box is the median. The lower quartile q1, is the 25% percentile (the median of the values that are less than median), and the upper quartile q3 is the 75% percentile (the median of the values that are greater than median). The length of the box is $d = q3 - q1$. The tails of the box represent the theoretical bound within all data points are likely to be found if the distribution is normal. The upper tail is $q3 + 1.5d$ and the lower tail is $q1 - 1.5d$. Figure 4 shows the instability box plot graphic with its information.

Values outside the upper and lower tails are the outliers. Figure 5 shows the outliers analysis for the instability.

As it can be seen in the Figure, Screen and Graphic represent suspect outliers. When outliers are identified, it is necessary to decide what to do with them: exclude the data or include it in the analysis. In this case, both data were considered, since in the game domain, screen management is the most intensive task that is performed, and thus this module had to be highly coupled with the others, resulting in high instability. On the other hand, Graphic is a simple module responsible to represent

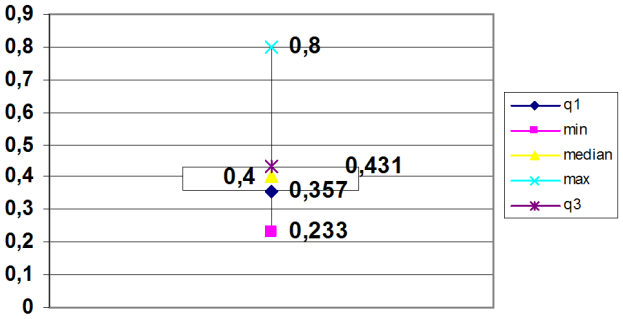

Figure 4. Instability Box Plot Graphic

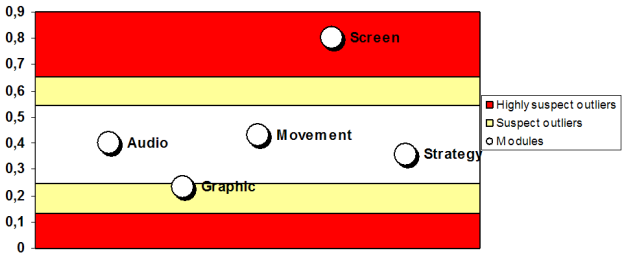

Figure 5. Outliers Analysis for the Instability

visual elements for the player such as number of lives, current score and high score, and thus it results in a low instability.

Table II shows the descriptive statistics with the data collected during the experimental study. The statistics present some relevant information for analysis. The instability mean (0.4442) **rejects** the null hypothesis. It indicates that the approach aids in producing the domain architecture with a good stability. Moreover, all the modules, except by *Screen*, present the instability value below the null hypothesis, reinforcing the premises that the approach allows to design stable modules.

Measure	Instability
Mean	0.4442
Maximum	0.8
Minimum	0.233
Standard Deviation	0.212679806
Null Hypothesis	$I \geq 0.5$

TABLE II.
RESULTS FOR THE INSTABILITY ANALYSIS

2) Maintainability: The second analysis consisted in analyzing the maintainability index for the architecture modules. This aspect is very important in a domain, since it is in continuous evolution. Thus, it is important to try to design the domain architecture with a high maintainability index. Figure 6 shows the maintainability index data graphically.

As it can be seen in the Figure, in general, the data set does not present a large discrepancy. The outliers analysis showed that there are no outliers in this case, even with *Movement* and *Strategy* having values in a different range compared with the other ones.

Table III shows the descriptive statistics with the data

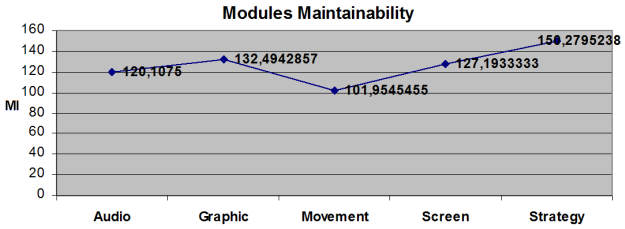

Figure 6. Modules Maintainability Graphic

collected during the experimental study. The instability mean (126.40583766) **rejects** the null hypothesis. It indicates that the process aids in producing the domain architecture with a high maintainability index. Moreover, it is important to highlight that all the modules had MI above 85, what indicates that all of them have a good maintainability and are not difficult to maintain.

Measure	Maintainability
Mean	126.40583766
Maximum	150.2795238
Minimum	101.9545455
Standard Deviation	17.6435327
Null Hypothesis	$M_I < 85$

TABLE III.
RESULTS FOR THE MAINTAINABILITY ANALYSIS

3) Complexity: The third analysis consisted in analyzing the components complexity. This aspect is important because it can identify components that will be difficult to test or maintain. Thus, it is important that the approach is able to design components with low complexity. Figure 7 shows the complexity data graphically.

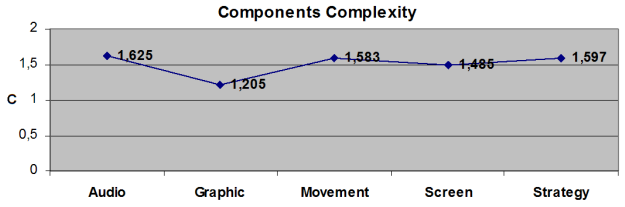

Figure 7. Component Complexity Graphic

According to the Figure, in general, the data present a similar distribution. According to the outliers analysis, only *Graphic* presents the complexity in a different range. However, as this component is a simple component, this data was not considered an outlier and it was included in the analysis.

Table IV shows the descriptive statistics with the data collected during the experimental study. The complexity mean (1.499) **rejects** the null hypothesis. It indicates that the process aids in producing the components with low complexity. Moreover, it is important to highlight that all the components had CC bellow 21, what indicates that none of them have much risk and are simple.

4) Difficulty in Understanding the Approach: After analyzing the questionnaire, and the subjects' answers about the difficulties in using the approach to design

Measure	Complexity
Mean	1.499
Maximum	1.625
Minimum	1.205
Standard Deviation	0.172603592
Null Hypothesis	$CC \geq 21$

TABLE IV.
RESULTS FOR THE COMPLEXITY ANALYSIS

domain-specific software architectures, we identified that only one subject (ID 1) did not have difficulties. One subject (ID 7) did not answer this question and this data point was excluded from the analysis. One subject (ID 6) had difficulty to understand the clustering process and recommended more examples showing how to do it. For two subjects (ID 2, 6), the decompose module activity needs to be better explained because its relationship with the refine module activity was obscure. Moreover, one subject (ID 6) had difficulty to identify the architectural drivers. The same subject reported that the documentation has a lack of details and needs to be reviewed in order to improve its understanding. Four subjects (ID 3, 4, 5, 6) had difficulty to create the functional dependency matrix, and one of them (ID 3) reported that examples can be useful to build it. Another commented aspect was the pattern applicability. For one subject (ID 2), more examples are needed, to show how to represent the variability using the design patterns. Another subject (ID 3) highlighted that it is necessary to define the right moment to use the patterns and recommended examples using the GRASP patterns. At the end, other subject (ID 6) commented that guidelines should be defined in the refine module activity for using the architectural patterns. The same subject had difficulty to identify the architectural drivers for the modules, recommending examples showing how to identify it. Finally, the last issue was related to variability. Three subjects (ID 3, 4, 5) had difficulties to represent the variability in design, especially, to map the variability present in features into classes.

In summary, a total of five subjects (ID 2, 3, 4, 5, 6) had problems to understand the approach, representing approximately 83,3%. This value **does not reject** the null hypothesis (DD $\geq$ 30%). However, it is necessary to highlight that this value for the null hypothesis was defined without any previous data, since this was the first time this aspect was analyzed, differently from the other metrics, which had pre-established values. Nevertheless, the next time that the experiment is performed this value can be refined based on this experience, resulting in a more calibrated metric.

G. Conclusions

Even with the analysis not being conclusive, the experimental study indicates that the approach allows designing the domain architecture with a good stability and maintainability. Additionally, the components designed using the approach present a low complexity. On the other hand, the aspects related to understanding the approach need to be reviewed and improved. With the results identified in the experiment, the values can be calibrated in a more accurate way. Nevertheless, most problems identified by the subjects in terms of difficulties are more related to the provided training than with the approach itself.

H. Lessons Learned

After concluding the experimental study, we could identify some aspects that need further consideration in order to repeat the experiment.

1) Domain: Two subjects (ID 1, 5) reported that the game domain should be re-discussed since it is a non-trivial domain and often a domain expert is necessary to aid in the process.

2) Domain: Besides the improvements related to the lectures, three subjects (ID 4, 5, 6) highlighted that the training should include a complete and detailed example covering all the activities of the approach.

3) Questionnaires: The questionnaires should be reviewed in order to collect more precise data related to feedback and the approach. Moreover, a possible improvement can be to collect it after the iterations during the project avoiding losing useful information by the subjects.

4) Pilot Project: The pilot project was performed just by one subject. Perhaps, this aspect should be re-analyzed in order to include all the subjects that will execute the experimental study.

5) Subjects Skills: The approach does not define the skills necessary to each role. Moreover, in this experiment, the roles were defined in an informal way, often allocating the subjects for the roles defined in their jobs. However, this issue should be reviewed in order to be more systematic and reduce risks.

6) Motivation: As the project was relatively long, it was difficult to keep the subjects' motivation during all the execution. Thus, this aspect should be analyzed in order to try to control it. A possible solution can be to define some checkpoints during the project.

V. CONCLUSION AND FUTURE WORK

Domain design is a critical aspect in a domain engineering process since its main goal is to develop the domain-specific software architecture for a set of applications in a domain. However, the existing approaches do not present a systematic way of how to do it, since they are not completely close to domain design. Some are related to software architecture in general and other to component-based development. Moreover, the software reuse processes have different emphasis on steps of domain analysis, design, and implementation, not covering all their activities in details.

In this context, this paper discussed a systematic approach to design domain-specific software architectures, based on a set of guidelines, metrics, inputs, outputs, activities, sub-activities, and roles. We also presented the definition, planning, operation, analysis and interpretation of the experimental study that evaluated the viability of the domain design approach. The study analyzed the

possibility of subjects using the approach to design the domain architecture with good stability and maintainability, and components with low complexity. The study also analyzed the difficulties of using the approach. Even with the reduced number of subjects (7), the analysis has shown that the domain design approach can be viable. It also identified some directions for improvements.

We are currently improving the approach with the suggestions made by the subjects, as well as the experience gathered after its usage in real industrial projects.

REFERENCES

[1] V. R. Basili, L. Briand, and W. Melo, "How Reuse Influences Productivity in Object-Oriented Systems," *Communications of the ACM*, vol. 39, no. 10, pp. 104–116, 1996.

[2] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[3] M. Griss, "Making Software Reuse Work at Hewlett-Packard," *IEEE Software*, vol. 12, no. 01, pp. 105–107, 1995.

[4] W. B. Frakes and S. Isoda, "Success Factors of Systematic Software Reuse," *IEEE Software*, vol. 11, no. 01, pp. 14–19, 1994.

[5] M. Morisio, M. Ezran, and C. Tully, "Success and Failure Factors in Software Reuse," *IEEE Transactions on Software Engineering*, vol. 28, no. 04, pp. 340–357, 2002.

[6] W. B. Frakes and K. Kang, "Software Reuse Research: Status and Future," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 529–536, 2005.

[7] J. Bosh, *Design and Use of Software Architectures*. Addison Wesley, 2000.

[8] W. Tracz, "Domain-Specific Software Architecture (DSSA) Pedagogical Example," *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 3, pp. 49–62, 1995.

[9] D. Perry and A. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[10] E. S. d. Almeida, A. Alvaro, D. Lucrédio, V. C. Garcia, and S. R. d. L. Meira, "A Survey on Software Reuse Processes," in *IEEE International Conference on Information Reuse and Integration (IRI 2005)*. Las Vegas, USA: IEEE/CS Press, 2005.

[11] E. S. d. Almeida, A. Alvaro, V. C. Garcia, L. Nascimento, D. Lucrédio, and S. R. d. L. Meira, "Designing Domain-Specific Software Architecture (DSSA): Towards a New Approach," in *6th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, Mumbai, India, 2007.

[12] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice, 2nd Edition*. Addison-Wesley, 2003.

[13] STARS, "DAGAR: A Process for Domain Architecure Definition and Asset Implementation, Technical Report," 1996.

[14] M. Matinlassi, "Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QUADA," in *26th International Conference on Software Engineering (ICSE)*, Scotland, 2004.

[15] J. Cheesman and J. Daniels, *UML Components - A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.

[16] E. S. d. Almeida, J. C. C. P. Mascena, A. P. C. Cavalcanti, A. Alvaro, V. C. Garcia, D. Lucrédio, and S. R. d. L. Meira, "The Domain Analysis Concept Revisited: A Practical Approach," in *9th International Conference on Software Reuse (ICSR)*. Torino, Italy: Lecture Notes in Computer Science, Springer-Verlag, 2006.

[17] D. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[19] M. Svahnberg, J. van Gurp, and J. Bosch, "On the Notion of Variabilities in Software Product Lines," in *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, Amsterdam, Netherlands, 2001, pp. 45–54.

[20] M. Anastasopoulos and C. Gacek, "Implementing Product Line Variabilities," in *Symposium on Software Reusability (SSR)*, Toronto, Canada, 2001, pp. 109–117.

[21] J. Coplin, D. Hoffman, and D. Weiss, "Commonality and Variability in Software Engineering," *IEEE Software*, vol. 15, no. 6, pp. 37–45, 1998.

[22] B. Keepence and M. Mannion, "Using Patterns to Model Variability in Product Families," *IEEE Software*, vol. 16, no. 4, pp. 102–108, 1999.

[23] K. Lee and K. C. Kang, "Feature Dependency Analysis for Product Line Component Design," in *8th International Conference on Software Reuse (ICSR)*, Madrid, Spain, 2004, pp. 69–85.

[24] V. Sugumaran, M. Tanniru, and V. C. Storey, "Identifying Software Components from Process Requirements using Domain Model and Objects Libraries," in *20th ACM International Conference on Information System*, Charlotte, North Carolina, USA, 1999, pp. 65–81.

[25] H. Jain and N. Chalimeda, "Business Component Identification - A Formal Approach," in *5th International Enterprise Distributed Object Computing Conference (EDOC)*, Seattle, USA, 2001, pp. 183–187.

[26] S. D. Kim and S. H. Chang, "A Systematic Method to Identify Software Components," in *11th Asia-Pacific Software Engineering Conference (APSEC)*, Seoul, South Korea, 2004, pp. 92–98.

[27] A. P. Blois, C. M. L. Werner, and K. Becker, "Towards a Components Grouping Technique within a Domain Engineering Process," in *31st IEEE EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA), Component-Based Software Engineering Track*, Porto, Portugal, 2005, pp. 18–27.

[28] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming - Second Edition*. Addison Wesley / ACM Press, 2002.

[29] J. Warmer and A. Kleppe, *The Object Constraint Language - Precise Modeling with UML*. Addison-Wesley, 1999.

[30] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

[31] M. O. Barros, "Project Management based on Scenarios: A Dinamic Modeling and Simulation Approach (in portuguese)," Ph.D. Thesis, Federal University of Rio de Janeiro, Rio de Janeiro, 2001.

[32] V. R. Basili, G. Caldiera, and H. D. Rombach, "The Goal Question Metric Approach," in *Encyclopedia of Software Engineering, Vol. II*, 1994, vol. 2, pp. 528–532.

[33] R. Martin, "OO Design Quality Metrics: An Analysis of Dependencies," 1994. [Online]. Available: http://www.objectmentor.com/resources/articles/ oodmetrc.pdf

[34] E. VanDoren, "Maintainability Index Technique for Measuring Program Maintainability," 1997. [Online]. Available: http://www.sei.cmu.edu/str/descriptions/ mitmpm.html

[35] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[36] J. C. C. P. Mascena, E. S. d. Almeida, and S. R. d. L. Meira, "A Comparative Study on Software Reuse Metrics and Economic Models from a Traceability Perspective," in *IEEE International Conference on Information Reuse and Integration (IRI)*. Las Vegas, Nevada, USA: IEEE/CS Press, 2005.

[37] V. R. Basili, R. W. Selby, and D. H. Hutchins, "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering*, vol. 12, no. 7, pp. 733–743, 1986.

[38] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Carnegie Mellon University," Technical Report CMU/SEI-90-TR-21, 1990, software Engineering Institute, Carnegie Mellon University.

[39] N. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. Course Technology, 1998.

Eduardo Santana de Almeida is a reuse consultant at Recife Center for Advanced Studies and Systems (C.E.S.A.R), where he has conducted more than five industrial projects focused on the various reuse aspects, such as methods, processes and tools. Nowadays, he is a member of Consulters Program, a group composed of seventeen people out 650 employees of C.E.S.A.R, whose his responsibility is the software reuse area. He is also the author of about 65 papers published and presented in conferences on these and other software reuse topics in leading venues worldwide and has served as referee in important journal papers with focus on reuse such as IEEE Transaction on Software Engineering and Journal of System and Software. He is also head of the Reuse in Software Engineering (RiSE) group where one of his activities is to start new cooperation projects around the word involving software reuse and Member of the Brazilian Committee composed of 50 specialists in the software component area, created by Brazilian Government. Dr. Almeida was Co-Chair of the 6th Brazilian Workshop on Component-Based Development (2006), creator of the WIRE - Workshop for Reuse Introduction in Companies where he has served as Co-Chair and Member of the Steering Committee, and Guest Editor of the Journal of the Brazilian Computer Society, Special Issue on Software Reuse (2008). He has gained two awards of the Brazilian Symposium on Software Engineering - Tools Session (2002, 2004) and is also Member of the IEEE Computer Society and the Brazilian Computer Society. In addition he was leader of the conception on the first open source book on software reuse around the world and has worked as independent consultant involved in training, mentoring, consultant and solution development focused on software reuse. Dr. Almeida is also IEEE Computer Society Certified Software Development Professional (CSDP).

Alexandre Alvaro is a senior member of the RiSE (Reuse in Software Engineering) group, at the Federal University of Pernambuco. He received his M.Sc. in computer science from Federal University of Pernambuco in 2005 and he is a Ph.D. candidate in Computer Science at the Federal University of Pernambuco. Nowadays, he is a Systems Engineer at the Recife Center for Advanced Studies and Systems (C.E.S.A.R.). Author of several papers in important vehicles, such as ECOOP and Euromicro conferences, in software reuse, component-development and software component certification areas, he has presented research works at conferences in Brazil and several other countries, including Portugal , United States and Hong Kong . He is involved in four industrial projects that focus on different aspects of reuse, such as processes, environments and tools.

Vinicius Cardoso Garcia is a member of the Brazilian Computer Society, and a senior member of the RiSE - Reuse in Software Engineering Group, at the Federal University of Pernambuco. He received his BSc. in computer science from the Salvador University (UNIFACS) in 2001, and the M.Sc. in computer science from Federal University of So Carlos in 2005. He is currently a Ph.D. candidate at the Federal University of Pernambuco since 2005, and also a Systems Engineer at the Recife Center for Advanced Studies and Systems (C.E.S.A.R.) since 2005. Vinicius Garcia is (co-)author of over 30 referenced publications presented at conferences such as WCRE, IRI, ECOOP, CBSE, ICSR and EUROMICRO, amongst others. He is currently involved in four research projects in the computer Science area, more specifically in Software Reuse Maturity Models and Software Reuse Adoption in Software Development Process.

Leandro Nascimento is a member of the RiSE - Reuse in Software Engineering Group, and is a Bachelor of Computer Science at Federal University of Pernambuco. He is a M.Sc. Candidate in Computer Science at Federal University of Pernambuco. As the result of his graduation work, he presented a Framework for Graphical User Interface development in J2ME. Nowadays, he is a System Engineer at the Recife Center for Advanced Studies and Systems (C.E.S.A.R) and has been working on J2ME projects, developing research into the area related to Component-Based Software.

Daniel Lucrédio graduated and got his M.Sc. degree in Computer Science at the Federal University of São Carlos, and is currently a Ph.D. candidate at the University of São Paulo São Carlos. He is the author of MVCASE, a free tool for UML modeling and component-based development, three times awarded at the Brazilian Symposium on Software Engineering, in 2001, 2002 and 2004. Author of several papers in important vehicles, such as IEEE and Euromicro conferences, in the software reuse and component-development areas, he presented research works and speeches at conferences in Brazil and several other countries, including Canada, United States, France and Hong Kong. He is also co-author of the first open source book on software reuse. He currently works as a software development projects coordinator, using most recent Java Technologies, and performs research on applying Model-Driven Engineering to improve software reusability.

Silvio Lemos Meira is the chair of Software Engineering at the Center for Informatics at the Federal University of Pernambuco in Recife , Brazil . Prof. Meira holds a B.Sc. in Electronic Engineering from ITA, S. J. Campos, Brazil, an M.Sc. in Computer Science from the Federal University of Pernambuco and a Ph.D. in Computing, under David Turner, from the University of Kent at Canterbury , UK . Among many other responsibilities, Prof. Meira is the leader of the Reuse in Software Engineering (RiSE) group and the Chief Scientist at C.E.S.A.R. (the Recife Center for Advanced Studies and Systems). Silvio Lemos Meira is (co-) author of well over 150 referenced publications in software engineering conferences and journals.