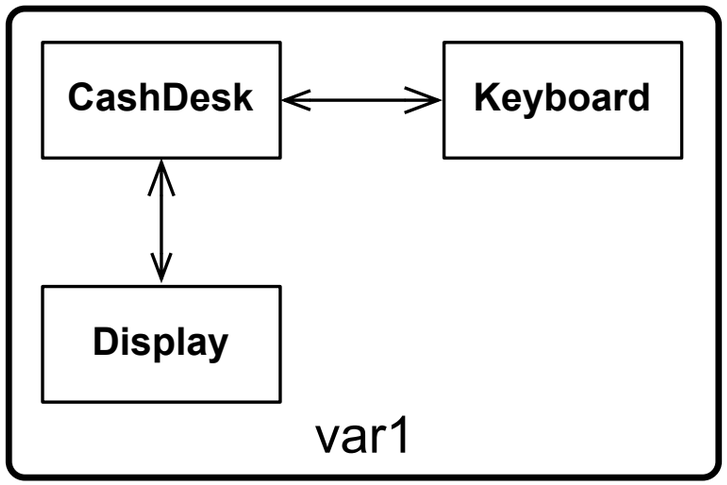
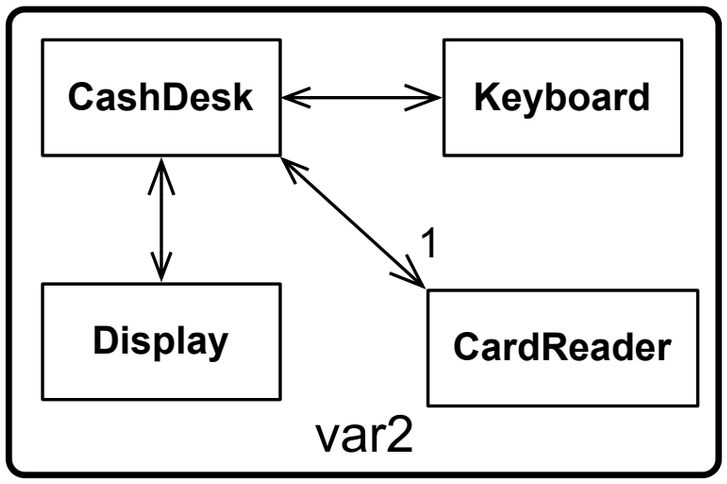
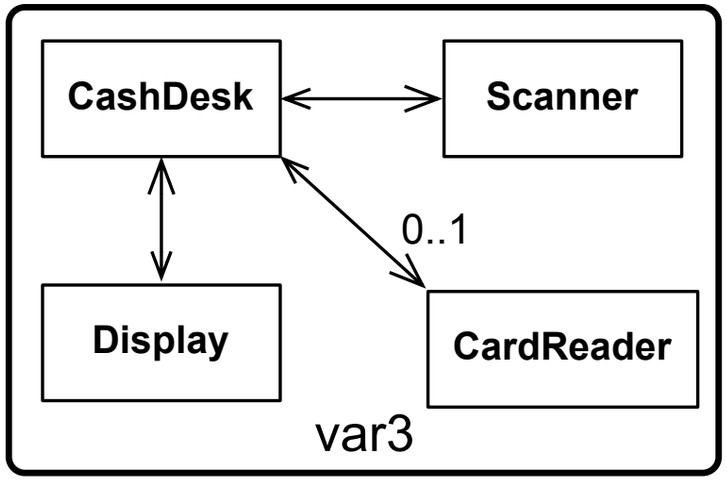
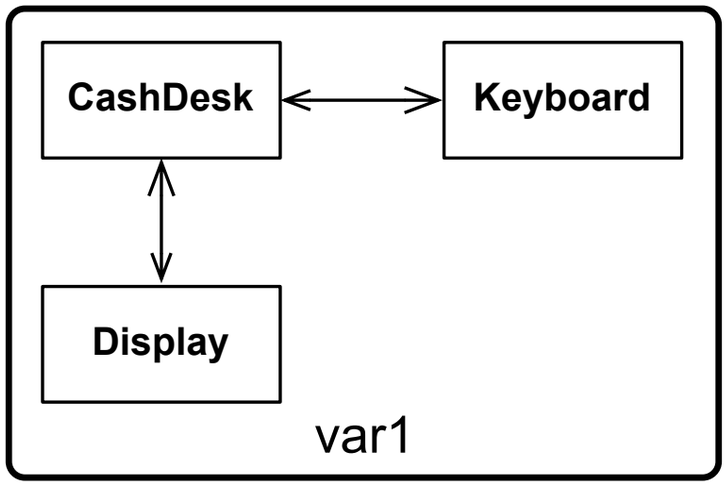
A brief overview of reuse mechanisms

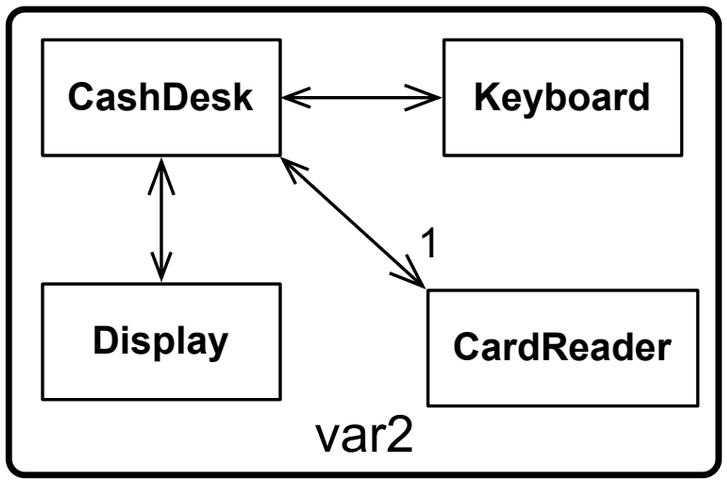

CashDesk
Keyboard
Display
var1

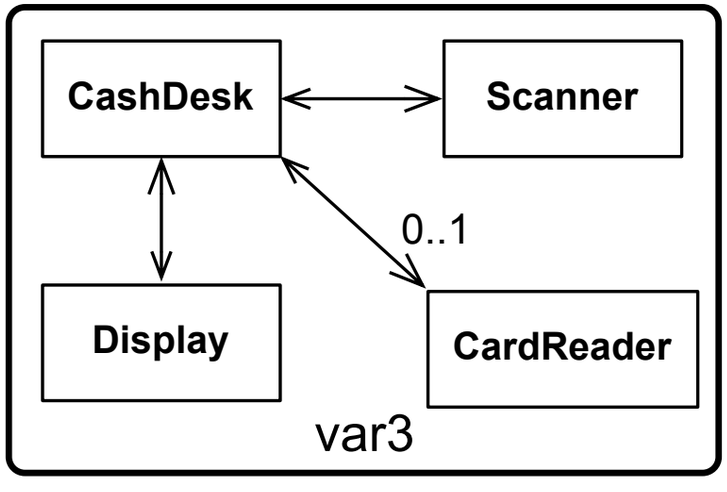

CashDesk
Keyboard
1
Display
CardReader
var2


CashDesk
Scanner
0..1
Display
CardReader
var3

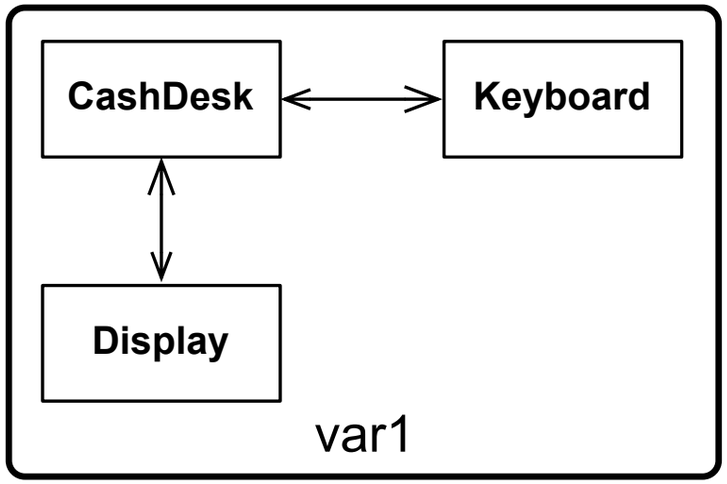
Multiple software artefacts such as class diagrams can sometimes be related and have common parts. In the following, let us refer to the set of artefacts as a **family** and members of this family as **variants**.

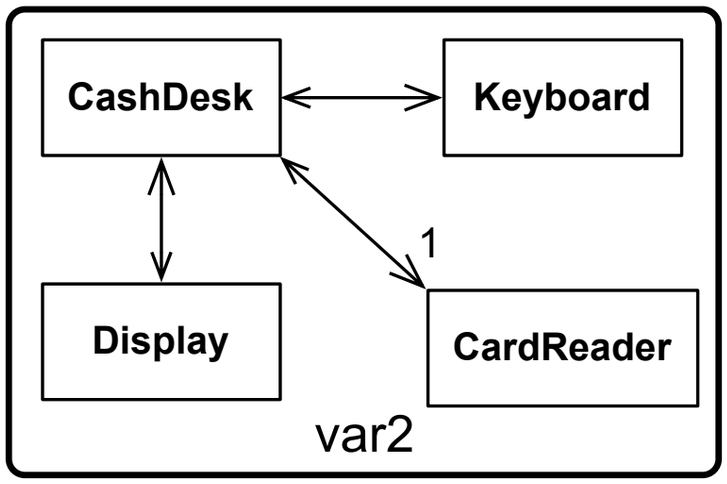


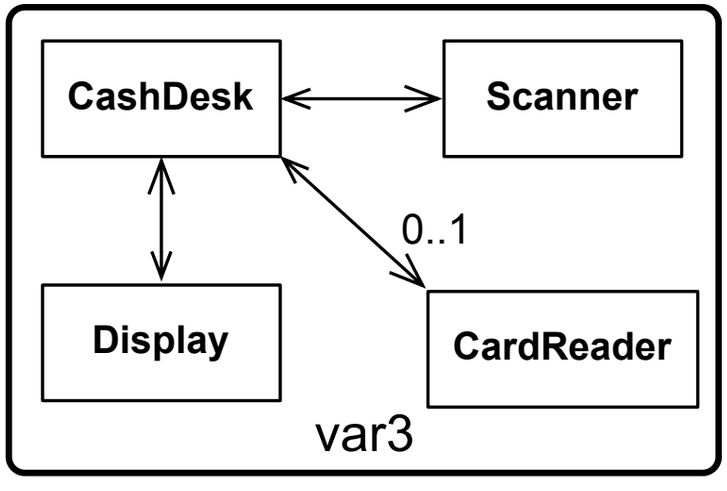




Multiple software artefacts such as class diagrams can sometimes be related and have common parts. In the following, let us refer to the set of artefacts as a **family** and members of this family as **variants**.

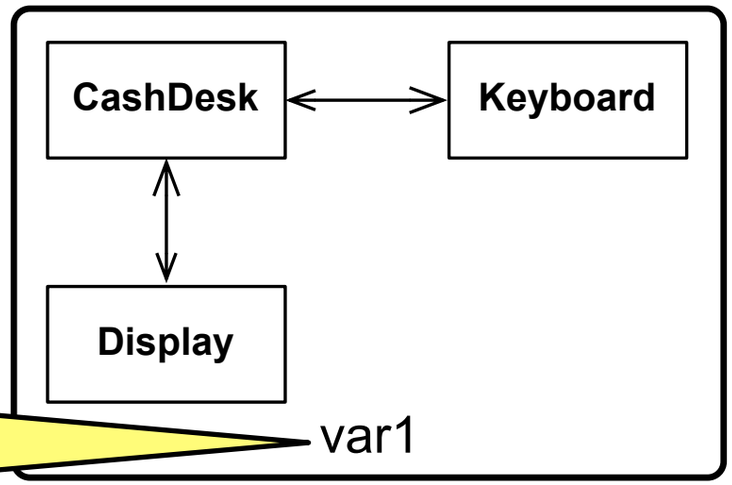




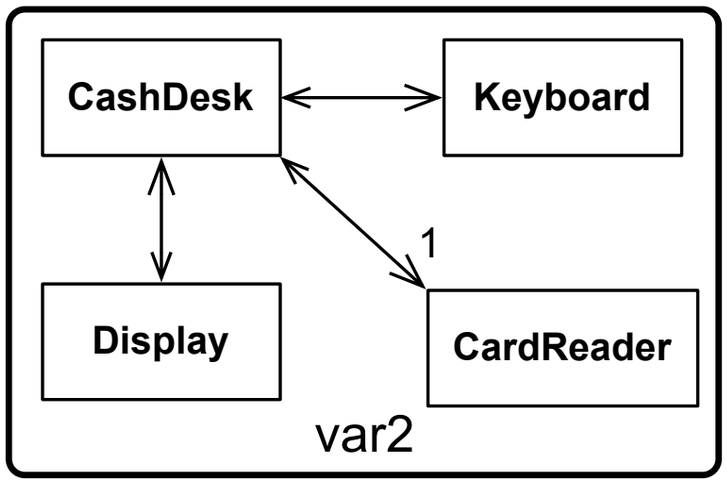


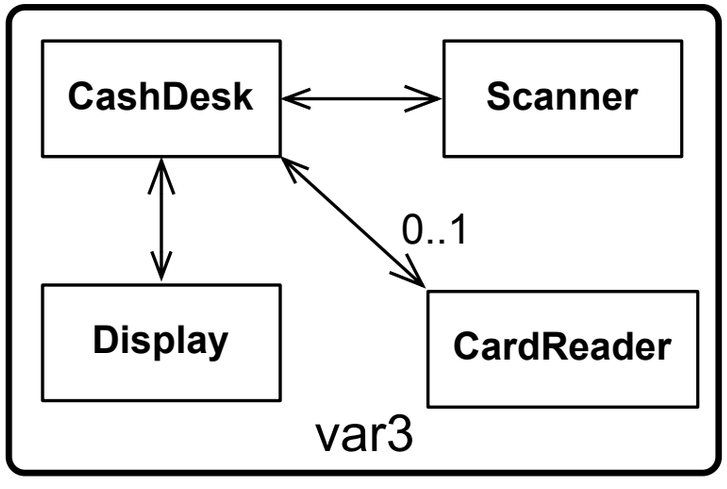
This is a family of cash desk systems. It consisting of three variants with some commonalities.

Multiple software artefacts such as class diagrams can sometimes be related and have common parts. In the following, let us refer to the set of artefacts as a **family** and members of this family as **variants**.



every variant is enclosed in a curved rectangle with a bold stroke and is named varX where X is a number.

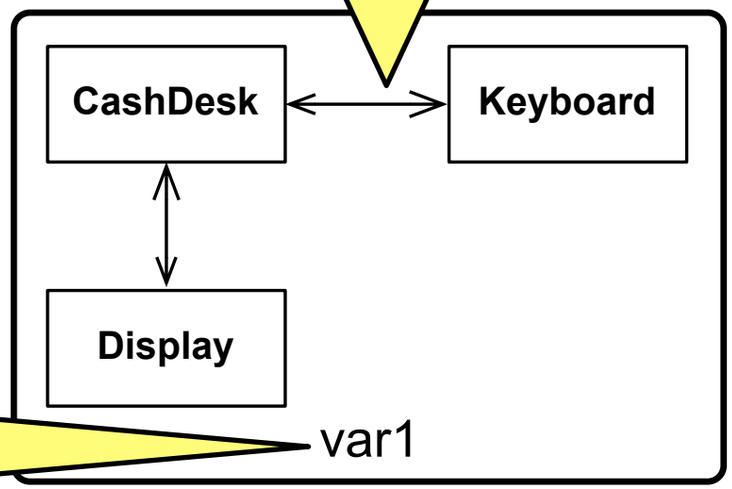




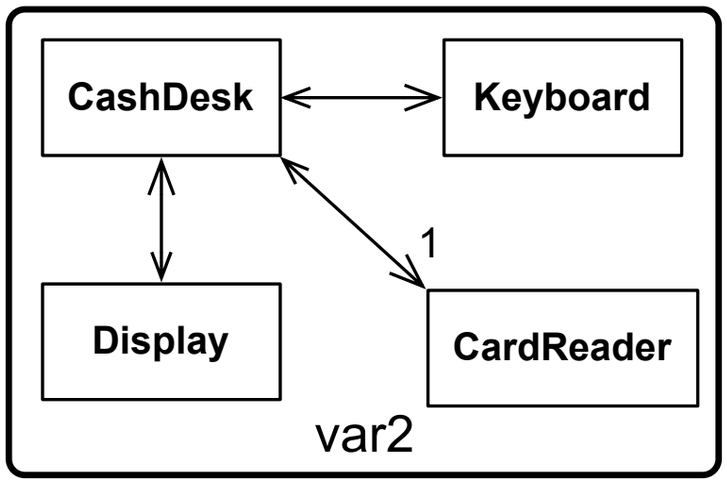
This is a family of cash desk systems. It consisting of three variants with some commonalities.

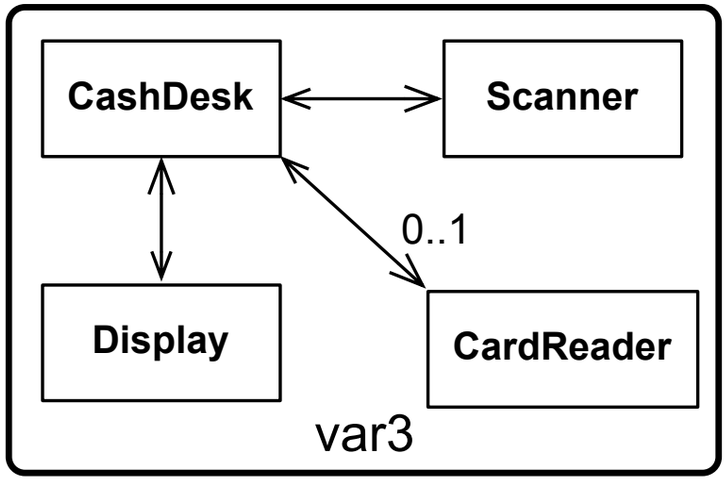
Multiple software artefacts such as class diagrams can sometimes be related and have common parts. In the following, let us refer to the set of artefacts as a **family** and members of this family as **variants**.

we omit all reference labels and multiplicities where they are not absolutely necessary (they are simply “unspecified”)



every variant is enclosed in a curved rectangle with a bold stroke and is named varX where X is a number.

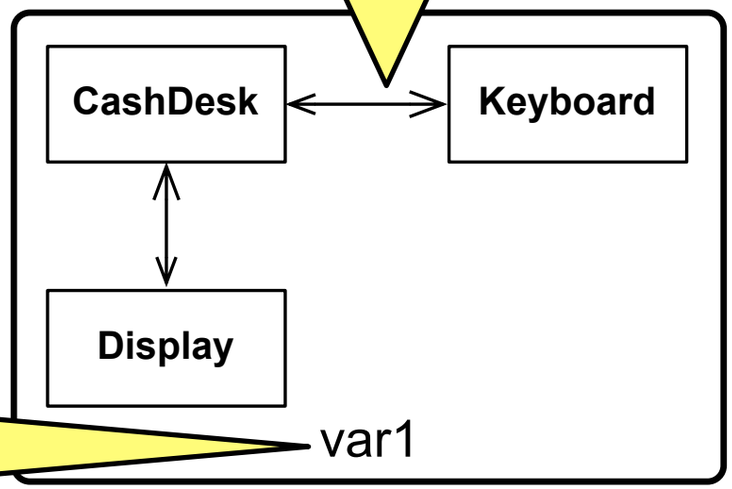




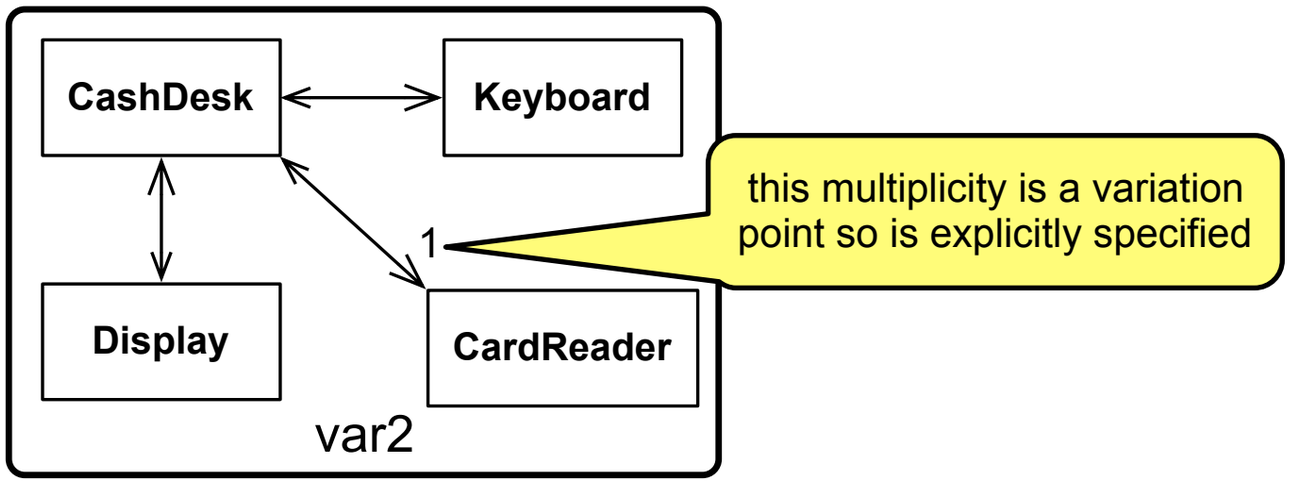
This is a family of cash desk systems. It consisting of three variants with some commonalities.

Multiple software artefacts such as class diagrams can sometimes be related and have common parts. In the following, let us refer to the set of artefacts as a **family** and members of this family as **variants**.

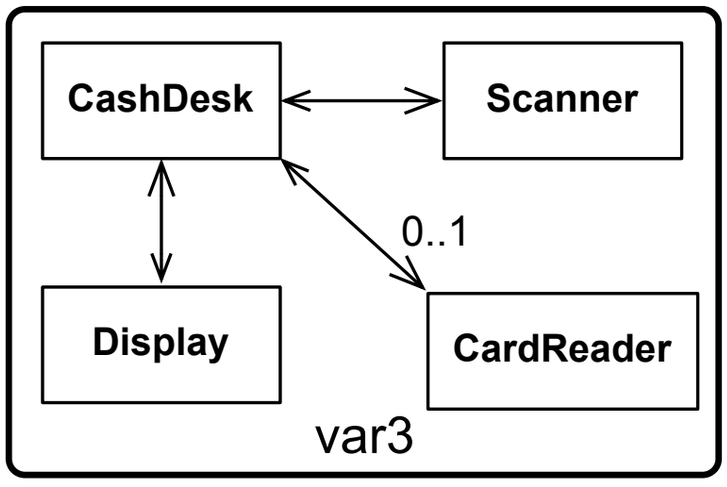
we omit all reference labels and multiplicities where they are not absolutely necessary (they are simply “unspecified”)



every variant is enclosed in a curved rectangle with a bold stroke and is named varX where X is a number.



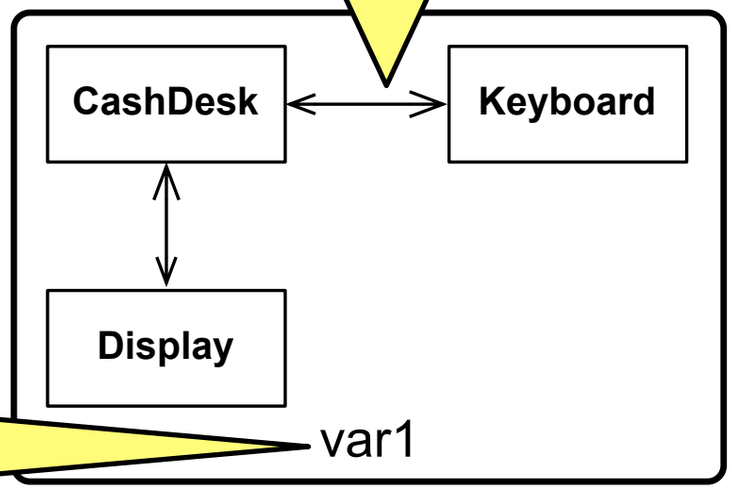
this multiplicity is a variation point so is explicitly specified



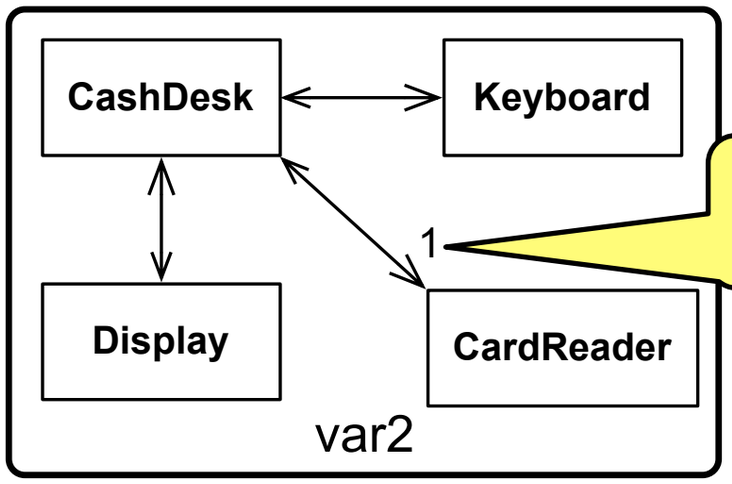
This is a family of cash desk systems. It consisting of three variants with some commonalities.

Multiple software artefacts such as class diagrams can sometimes be related and have common parts. In the following, let us refer to the set of artefacts as a **family** and members of this family as **variants**.

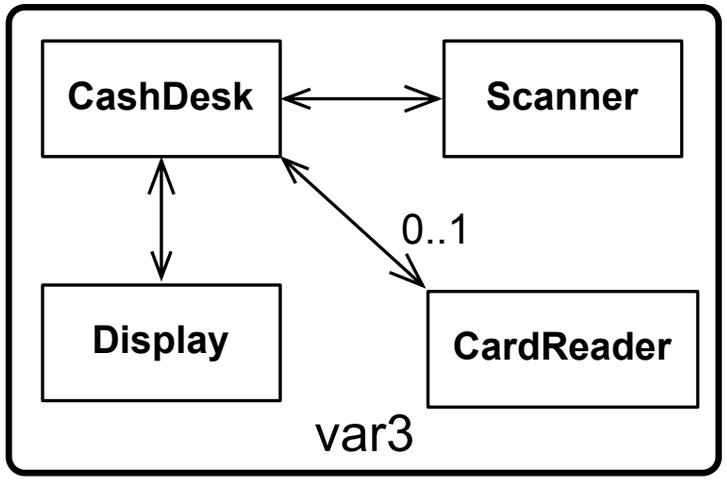
we omit all reference labels and multiplicities where they are not absolutely necessary (they are simply “unspecified”)



every variant is enclosed in a curved rectangle with a bold stroke and is named varX where X is a number.

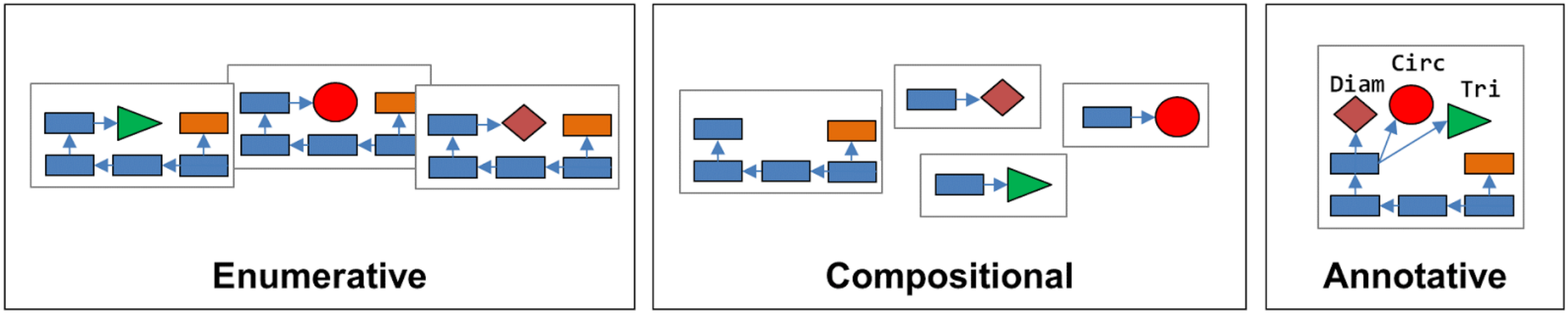


this multiplicity is a variation point so is explicitly specified

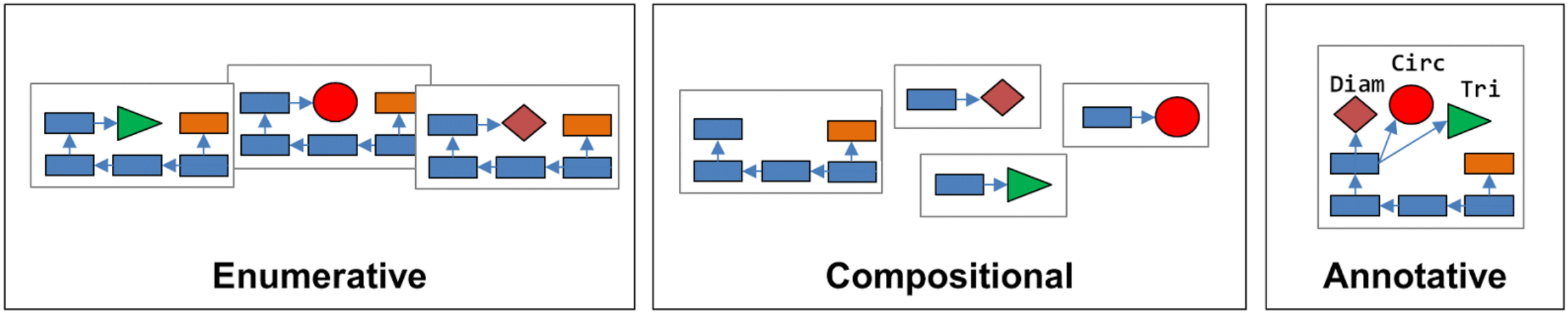


This is a family of cash desk systems. It consisting of three variants with some commonalities.

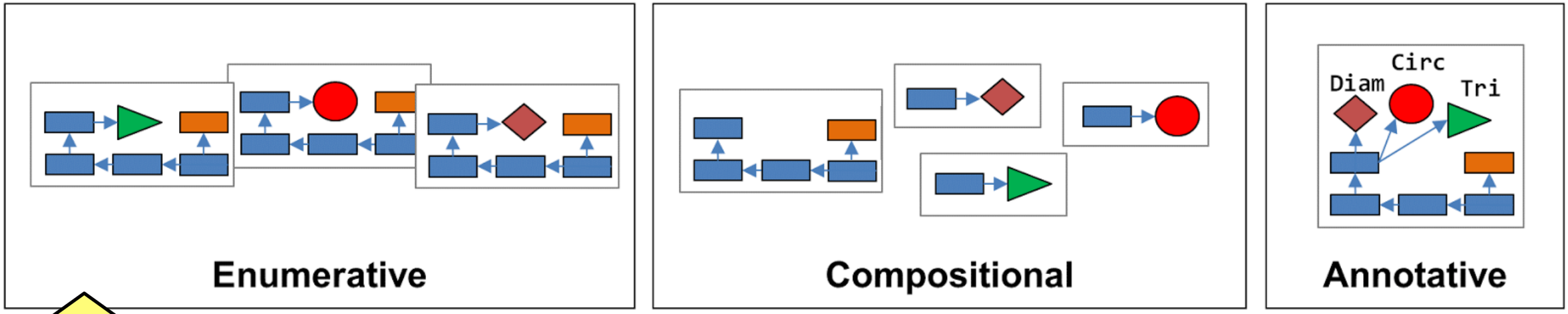
in the following, let us refer to both **classes** and **references** in a class diagram as **elements**


Enumerative
Compositional
Circ
Diam
Tri
Annotative

There exist different **reuse mechanisms** that support working with families of software artefacts.

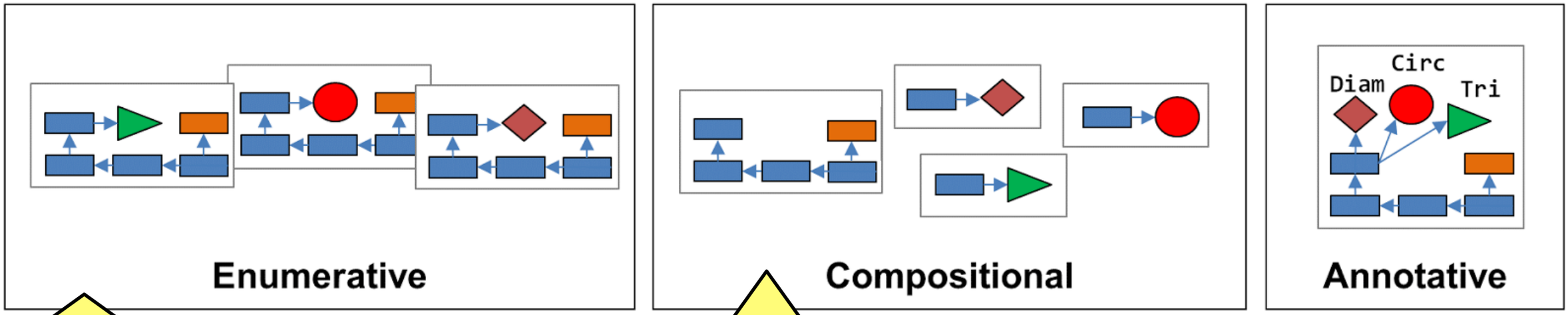


There exist different **reuse mechanisms** that support working with families of software artefacts.



enumerative mechanisms show a complete list of all variants in the family

There exist different **reuse mechanisms** that support working with families of software artefacts.

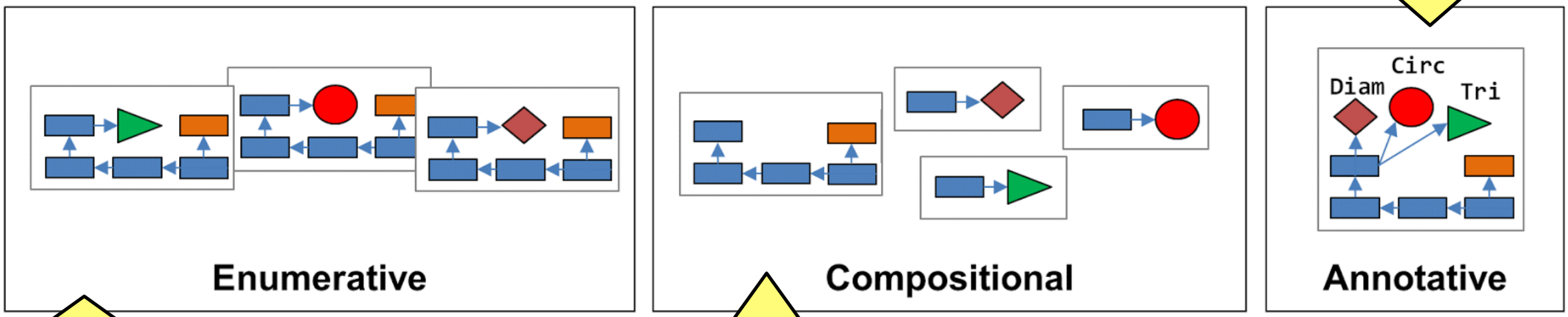


enumerative mechanisms show a complete list of all variants in the family

compositional mechanisms split the artefacts into smaller building blocks that can later be “composed” in some way and reused to produce all variants of a family (instead of enumerating them explicitly)

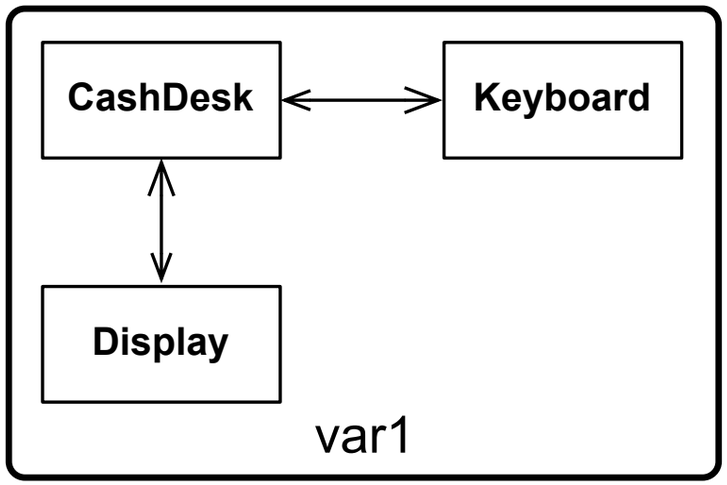
There exist different **reuse mechanisms** that support working with families of software artefacts.

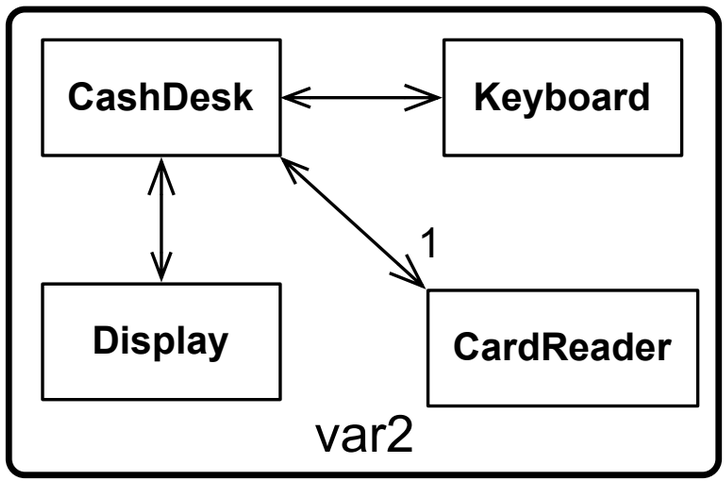
annotative mechanisms combine all artefacts into a single artefact with annotations that indicate which individual products contain which parts

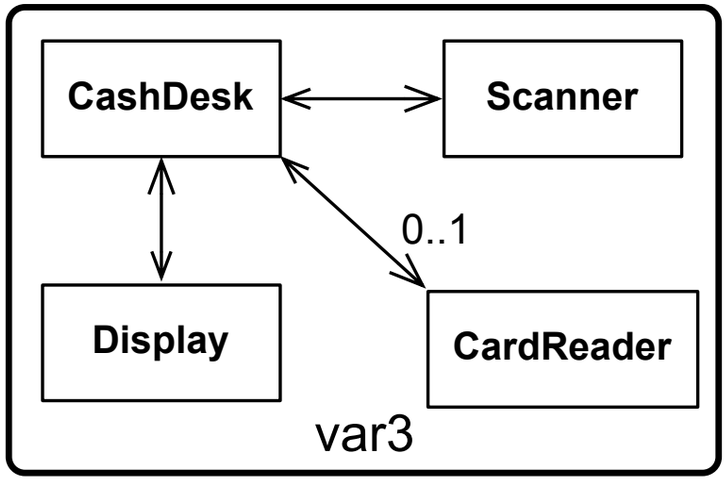


enumerative mechanisms show a complete list of all variants in the family

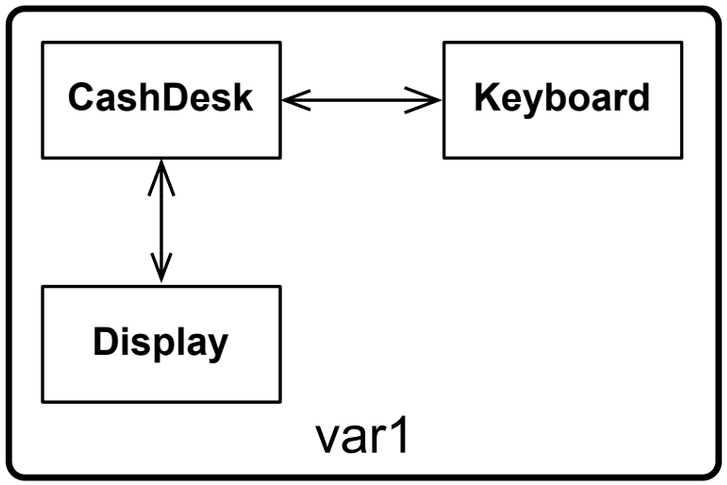
compositional mechanisms split the artefacts into smaller building blocks that can later be “composed” in some way and reused to produce all variants of a family (instead of enumerating them explicitly)

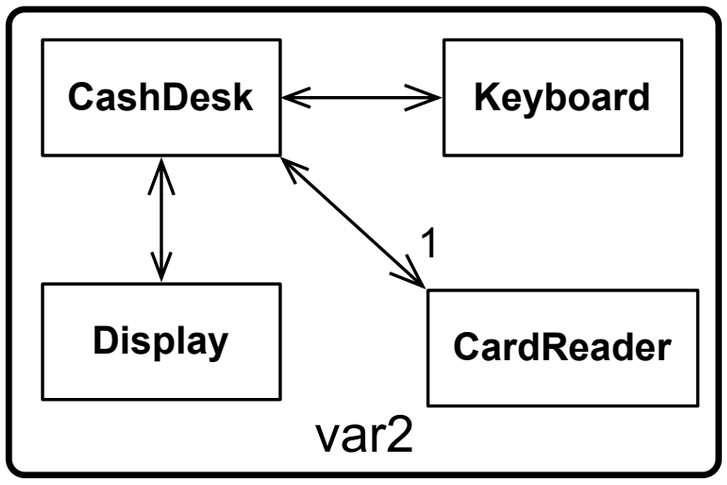

CashDesk
Keyboard
Display
var1

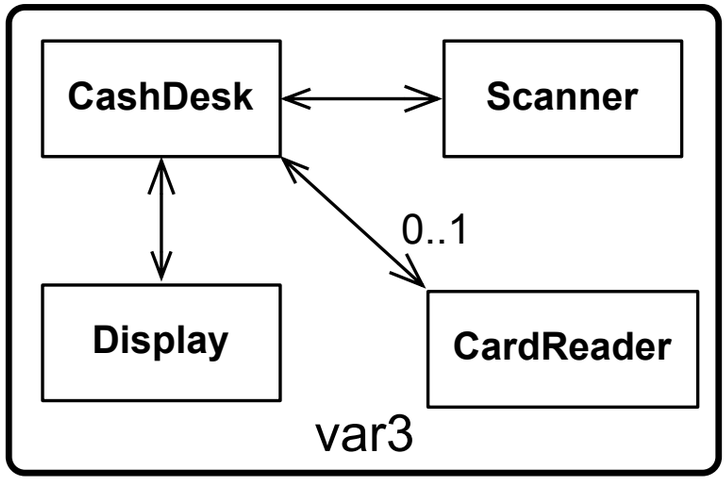

CashDesk
Keyboard
1
Display
CardReader
var2

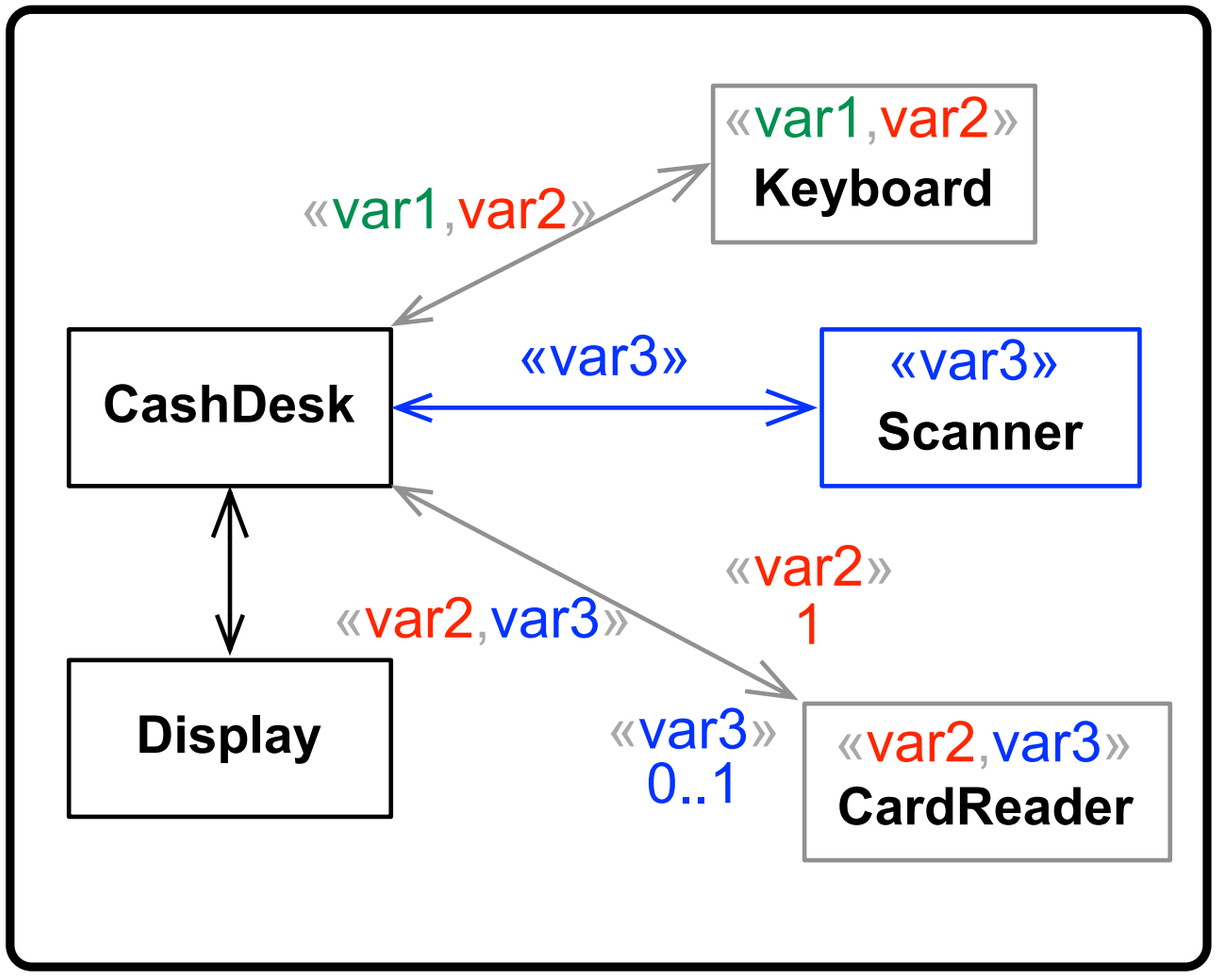

CashDesk
Scanner
0..1
Display
CardReader
var3

An **enumerative** mechanism illustrates the family by simply enumerating all variants as depicted here.

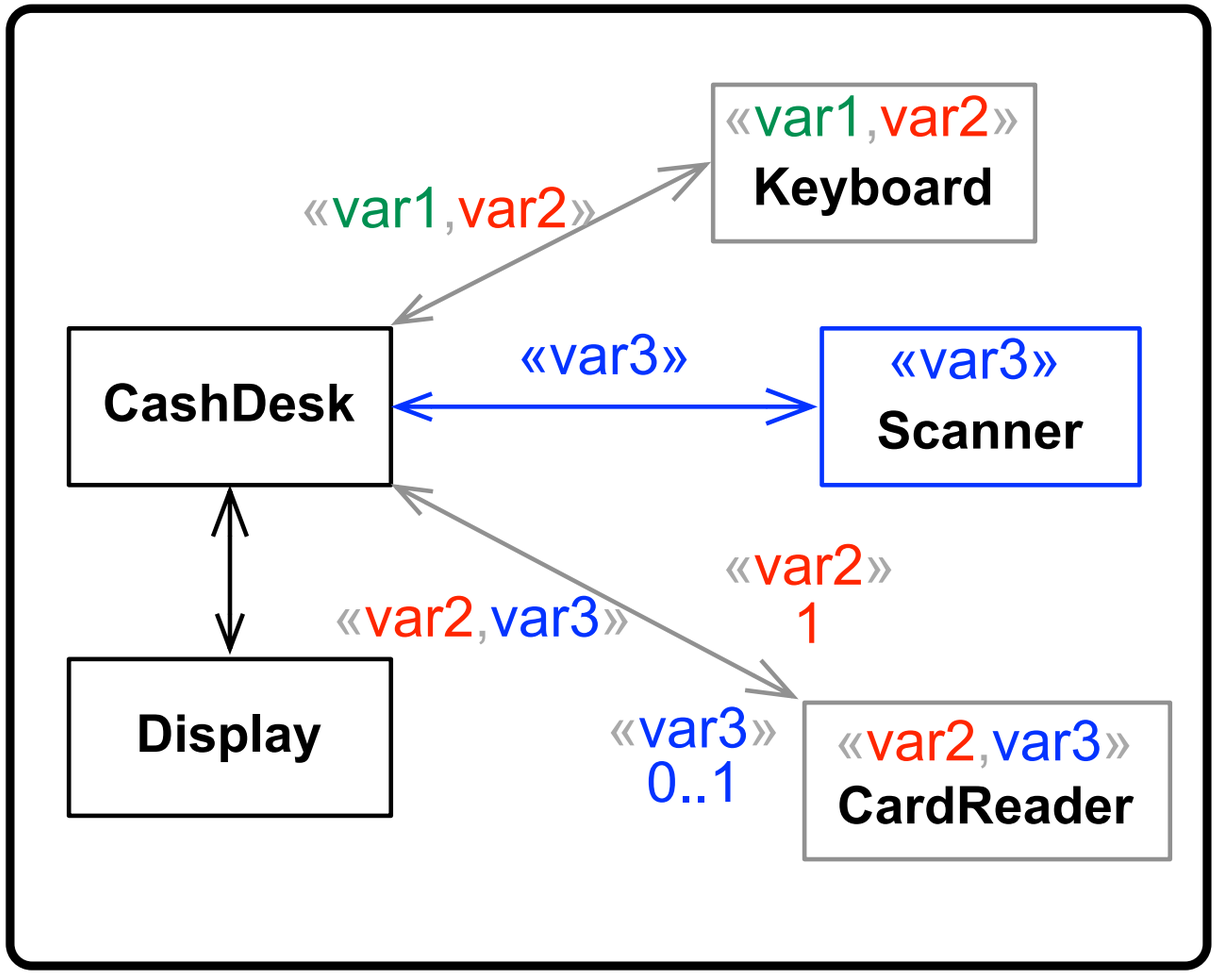




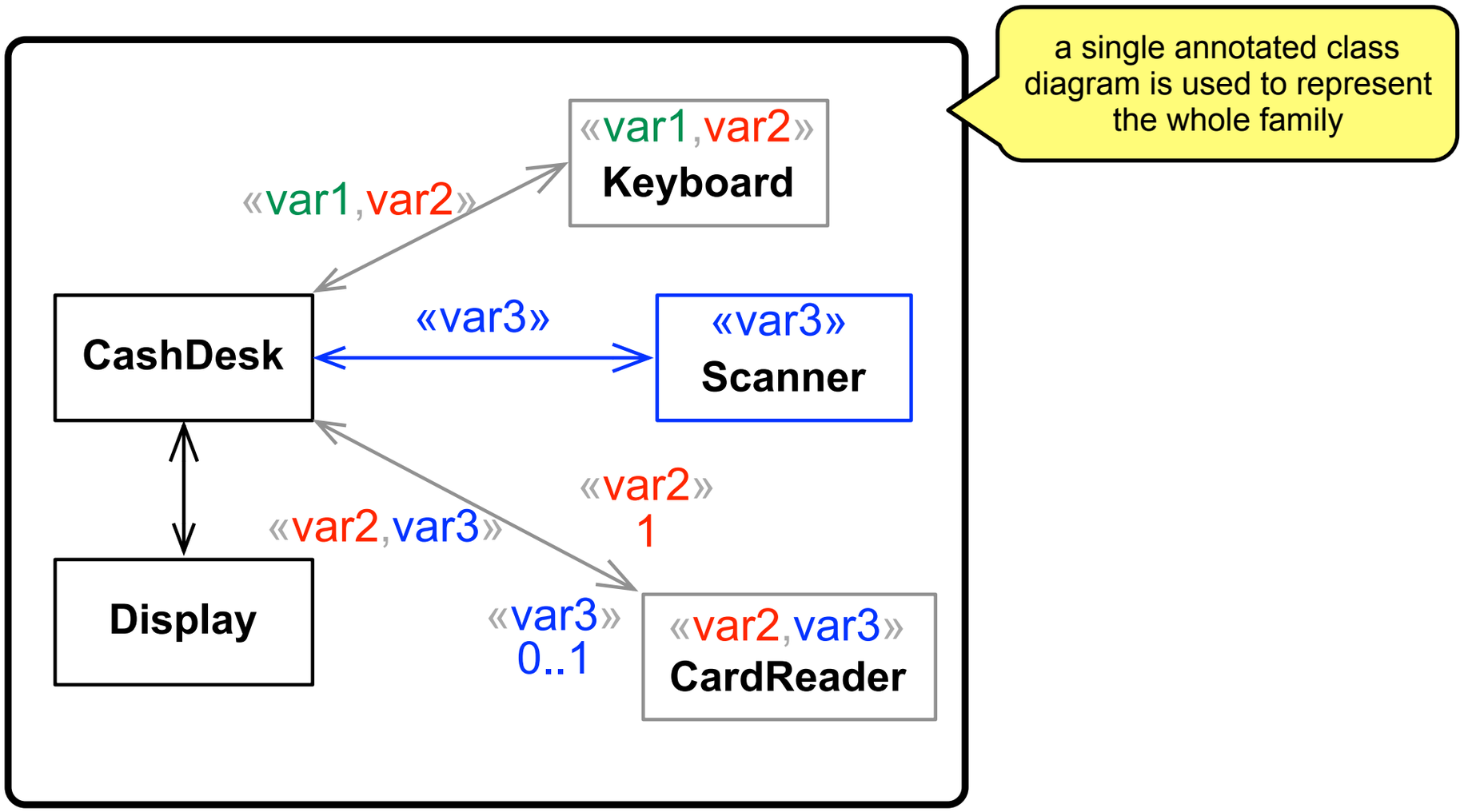



«var1,var2»
Keyboard
«var1,var2»
CashDesk
«var3»
«var3»
Scanner
«var2»
1
«var2,var3»
Display
«var3»
0..1
«var2,var3»
CardReader

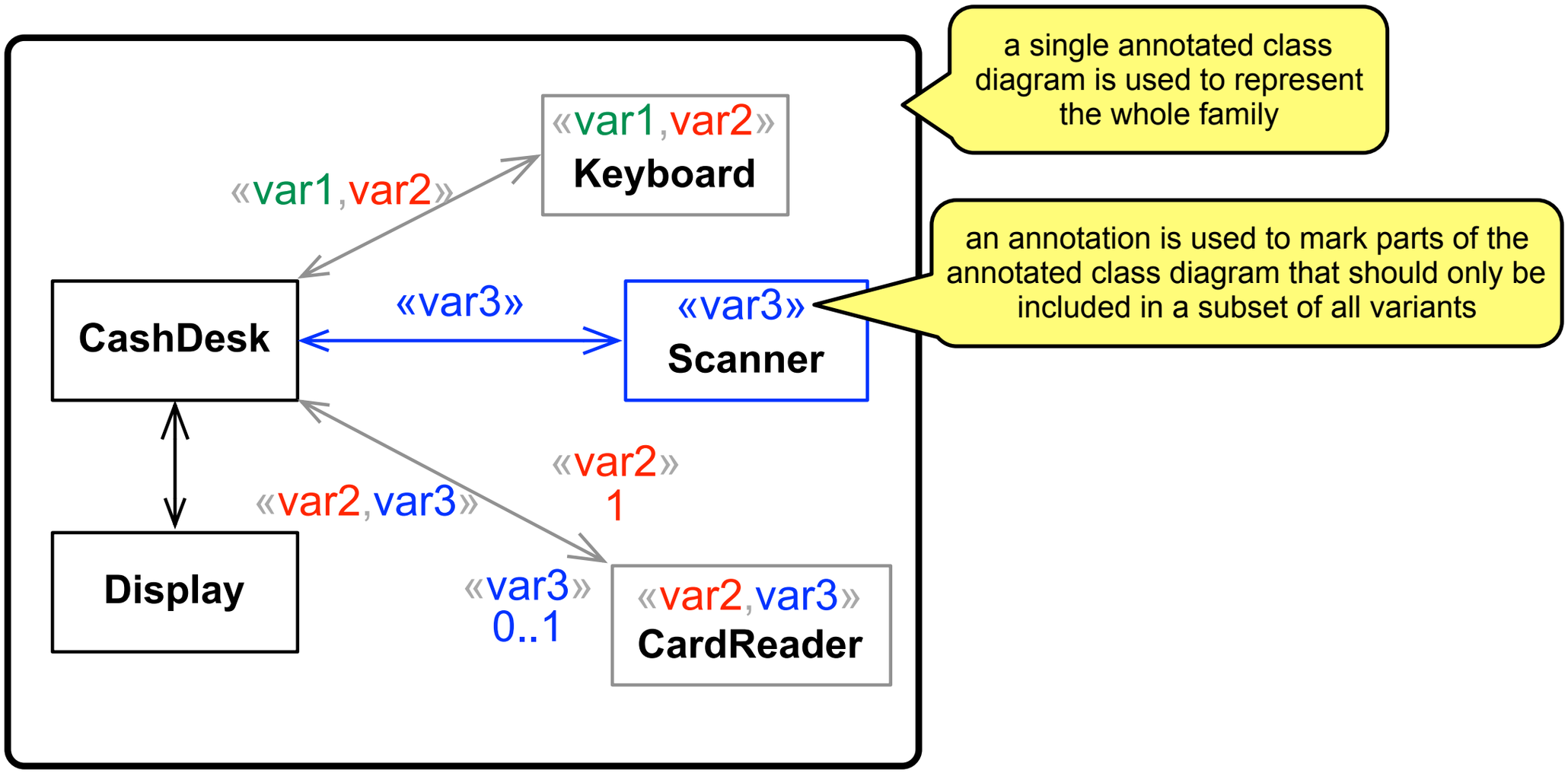
An **annotative** mechanism allows combining all variants into a single representation with suitable annotations.



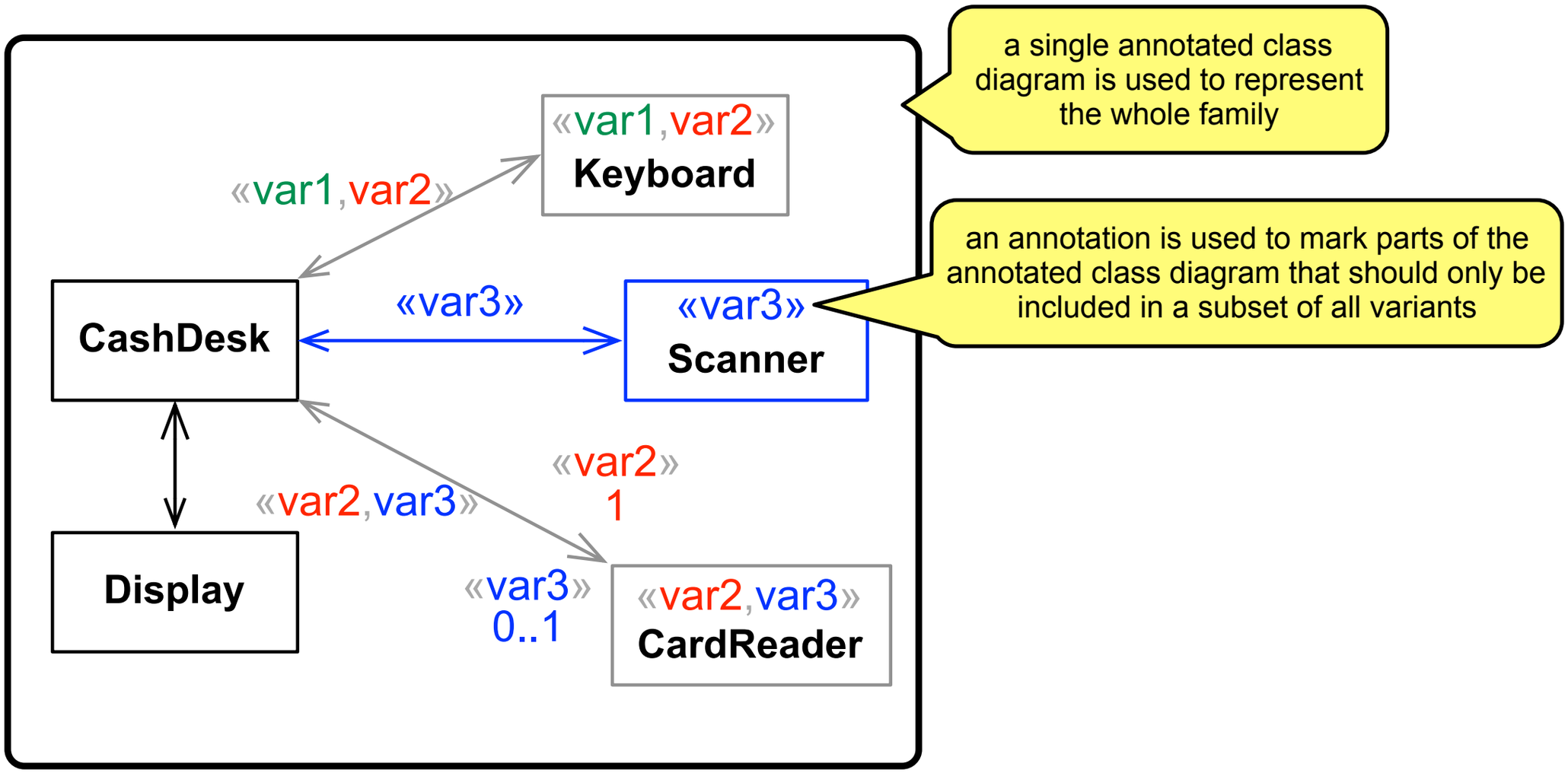
An **annotative** mechanism allows combining all variants into a single representation with suitable annotations.



An **annotative** mechanism allows combining all variants into a single representation with suitable annotations.



An **annotative** mechanism allows combining all variants into a single representation with suitable annotations.



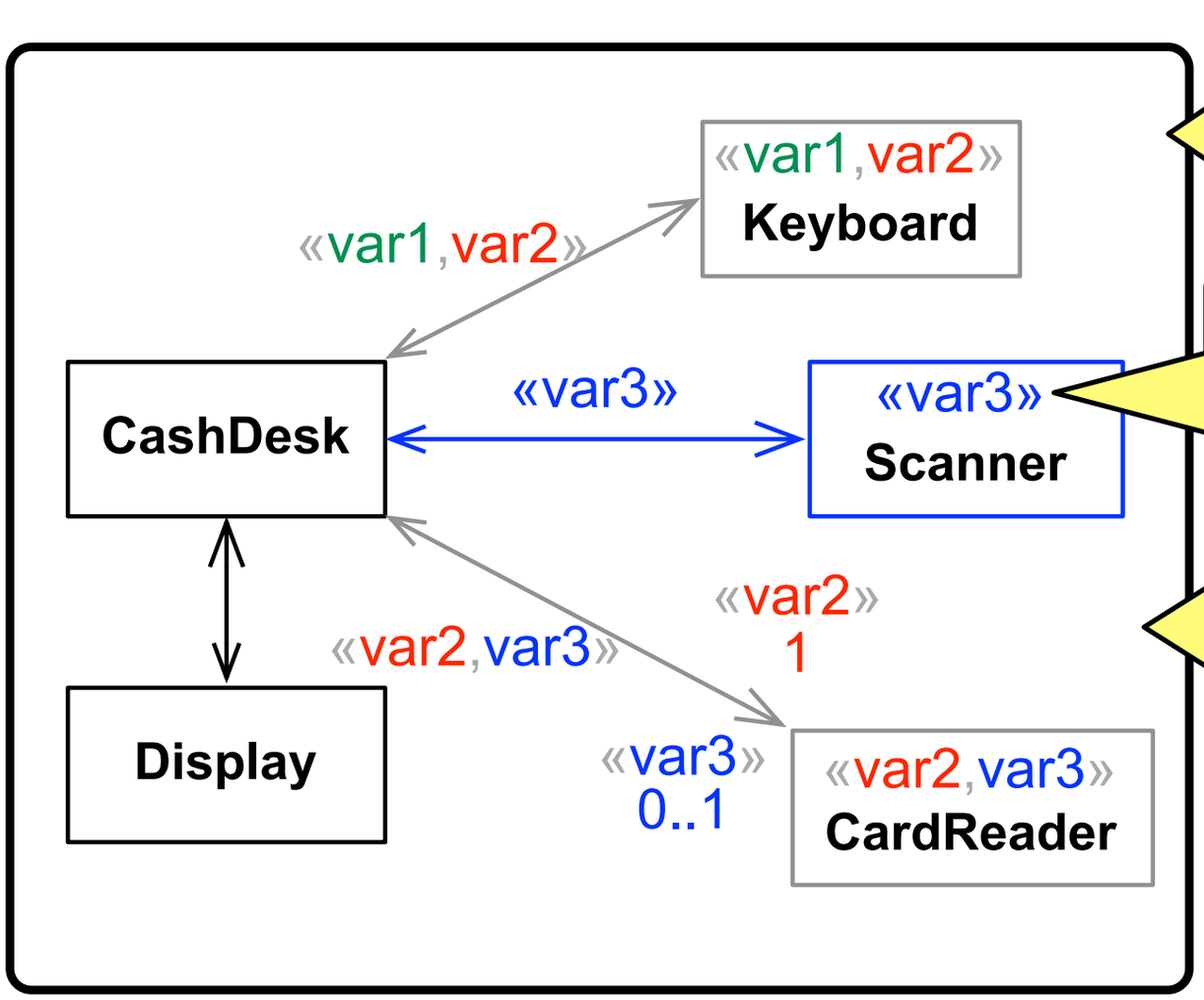
a single annotated class diagram is used to represent the whole family

an annotation is used to mark parts of the annotated class diagram that should only be included in a subset of all variants

colours provide additional visual support:

- elements with a **black outline** belong to all variants
- elements with a gray outline belong to two or more variants, but not to all variants
- elements with a coloured outline belong to precisely one variant

An **annotative** mechanism allows combining all variants into a single representation with suitable annotations.



a single annotated class diagram is used to represent the whole family

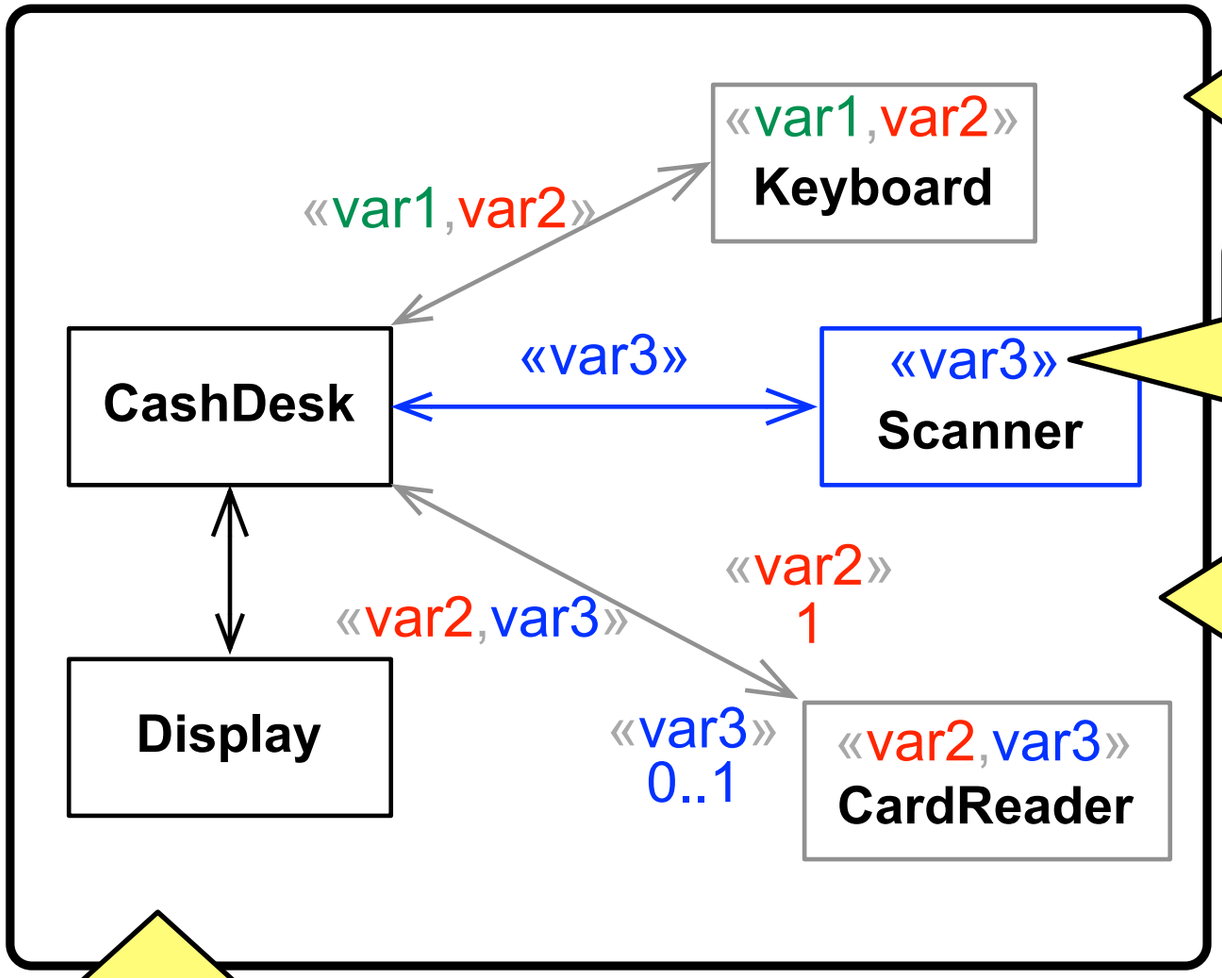
an annotation is used to mark parts of the annotated class diagram that should only be included in a subset of all variants

variants of the family are generated by:
selecting one of the variants (e.g., var1), and removing all parts that are annotated only with the other variants (e.g., if we select var1, we would remove all parts annotated with var2 and/or var3)

colours provide additional visual support:

- elements with a **black outline** belong to all variants
- elements with a gray outline belong to two or more variants, but not to all variants
- elements with a coloured outline belong to precisely one variant

An **annotative** mechanism allows combining all variants into a single representation with suitable annotations.



a single annotated class diagram is used to represent the whole family

an annotation is used to mark parts of the annotated class diagram that should only be included in a subset of all variants

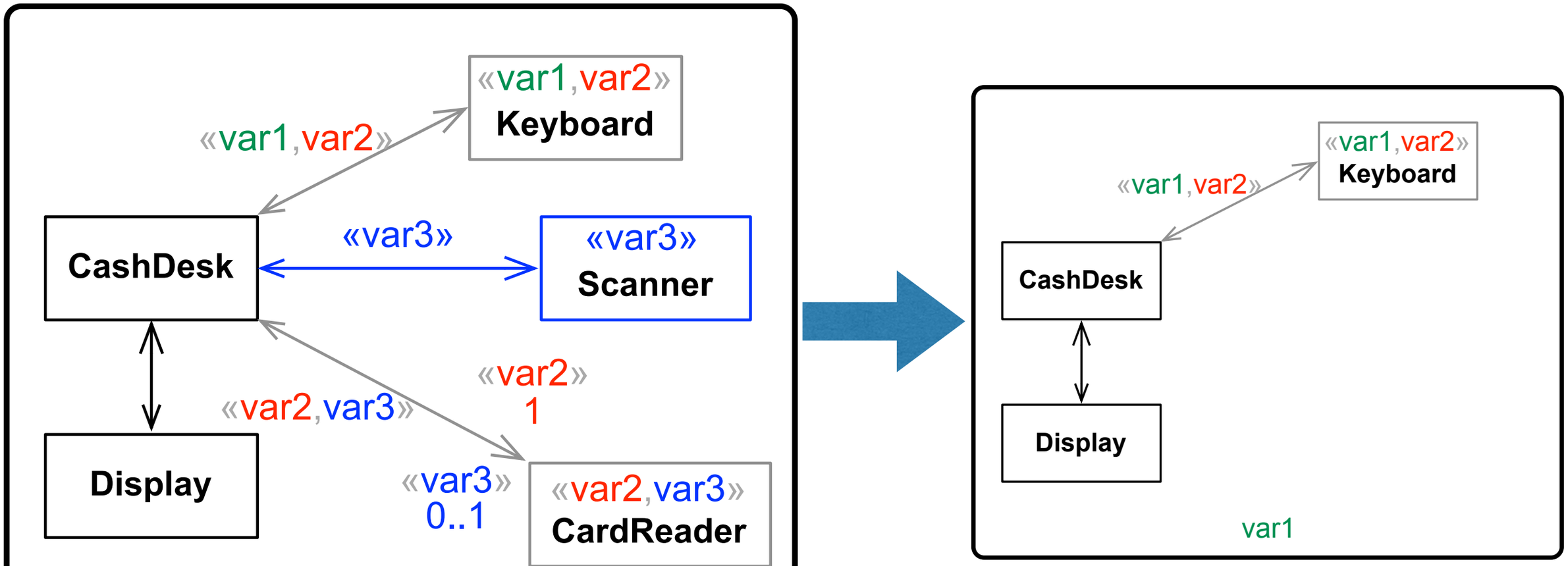
variants of the family are generated by:
selecting one of the variants (e.g., var1), and removing all parts that are annotated only with the other variants (e.g., if we select var1, we would remove all parts annotated with var2 and/or var3)

for example, when we select var1, the classes **Scanner** and **CardReader,** and their incoming associations will be removed, giving us a smaller class diagram (the variant var1).

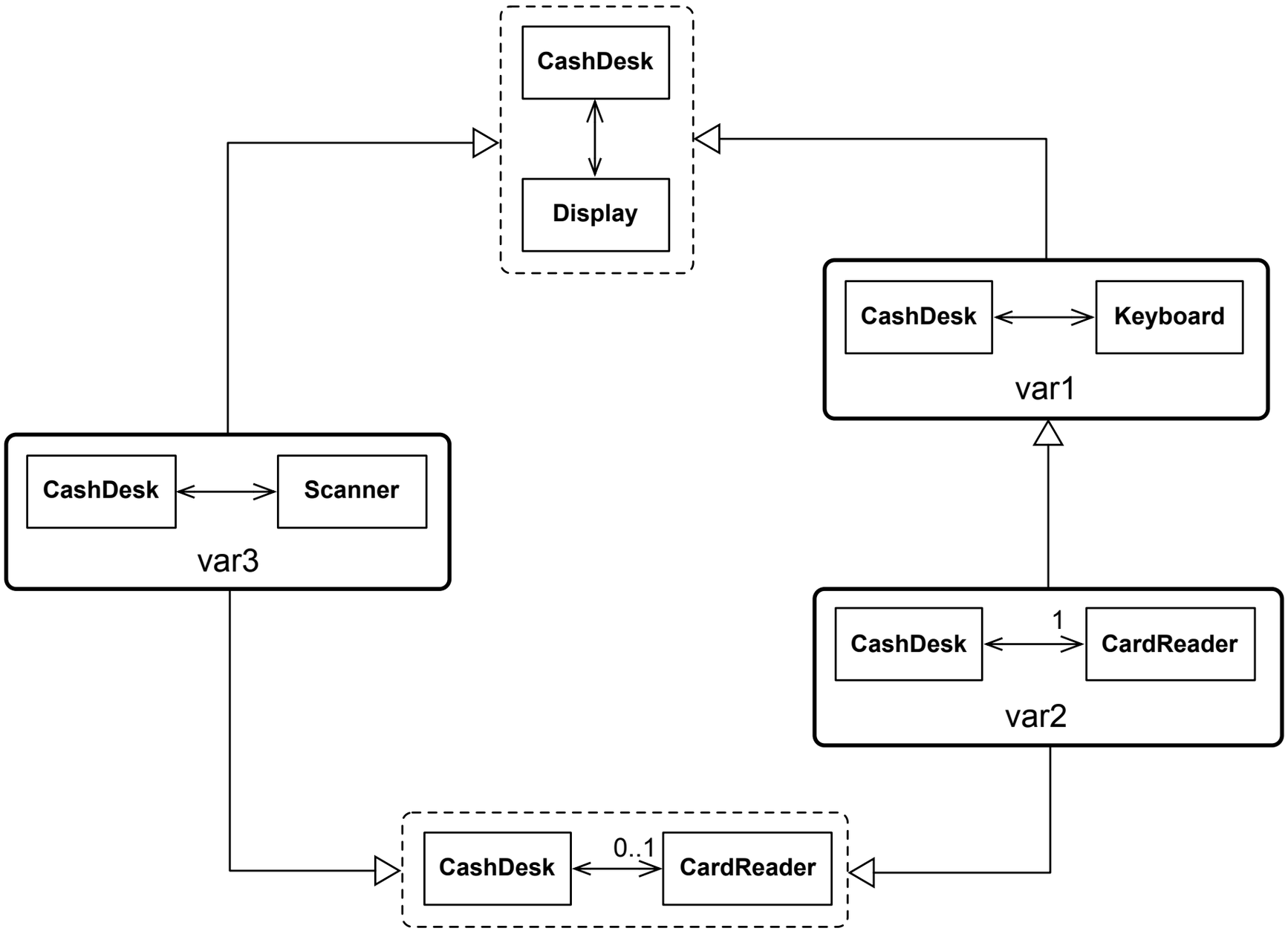
colours provide additional visual support:

- elements with a **black outline** belong to all variants
- elements with a gray outline belong to two or more variants, but not to all variants
- elements with a coloured outline belong to precisely one variant

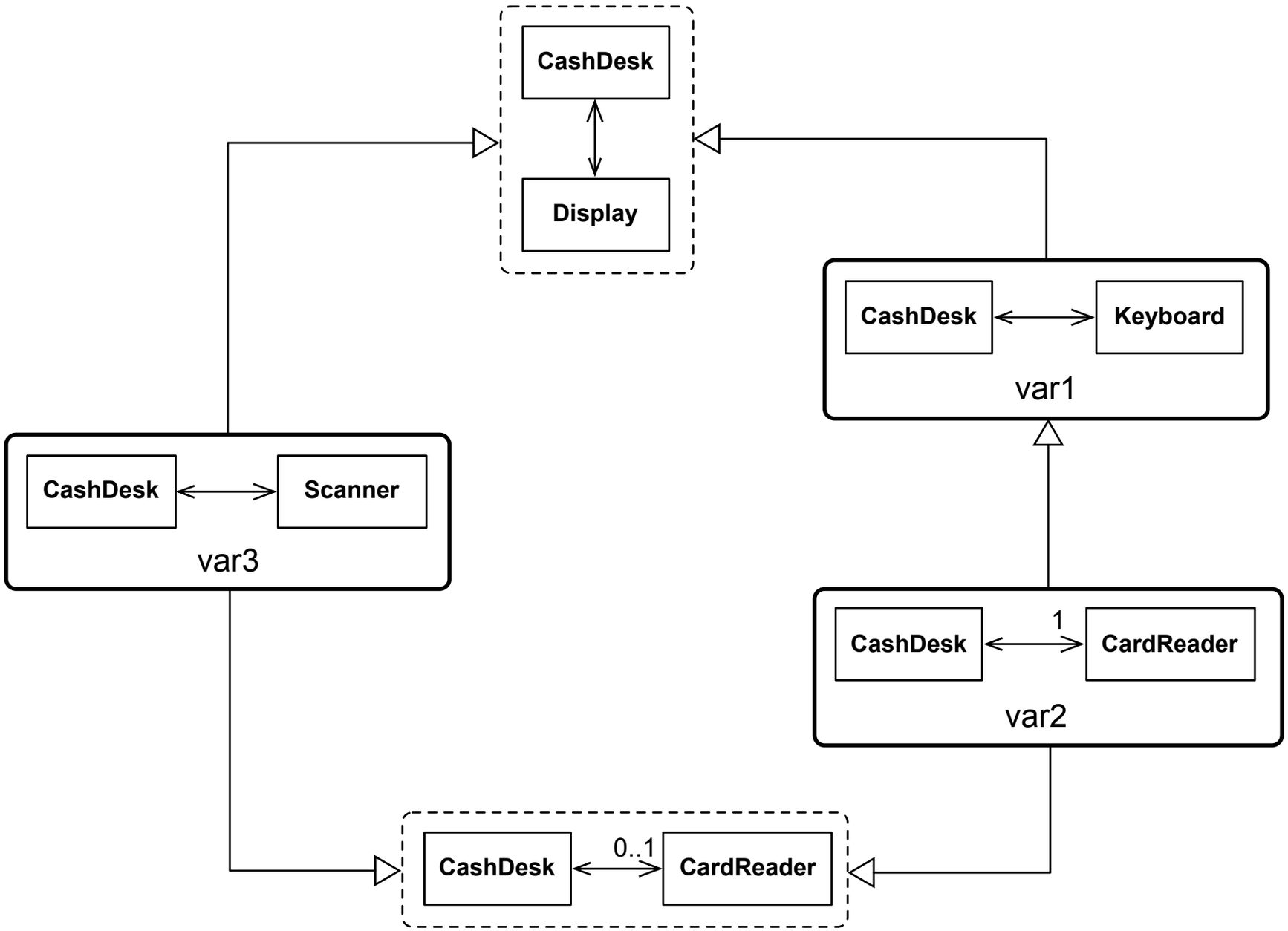
An **annotative** mechanism allows combining all variants into a single representation with suitable annotations.



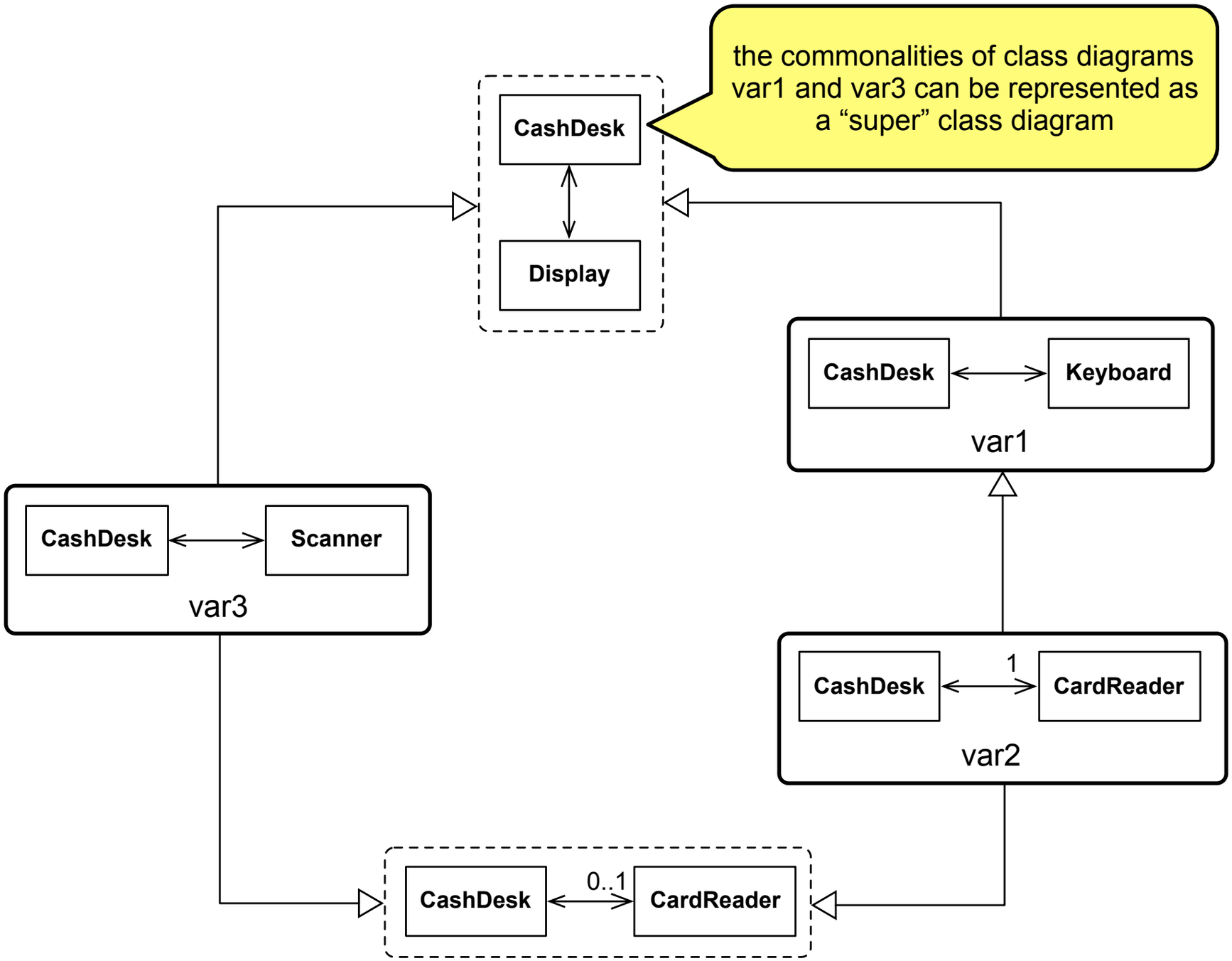
for example, when we select var1, the classes **Scanner** and **CardReader,** and their incoming associations will be removed, giving us a smaller class diagram (the variant var1).


CashDesk
Display
CashDesk
Keyboard
var1
CashDesk
Scanner
var3
CashDesk
1
CardReader
var2
CashDesk
0..1
CardReader

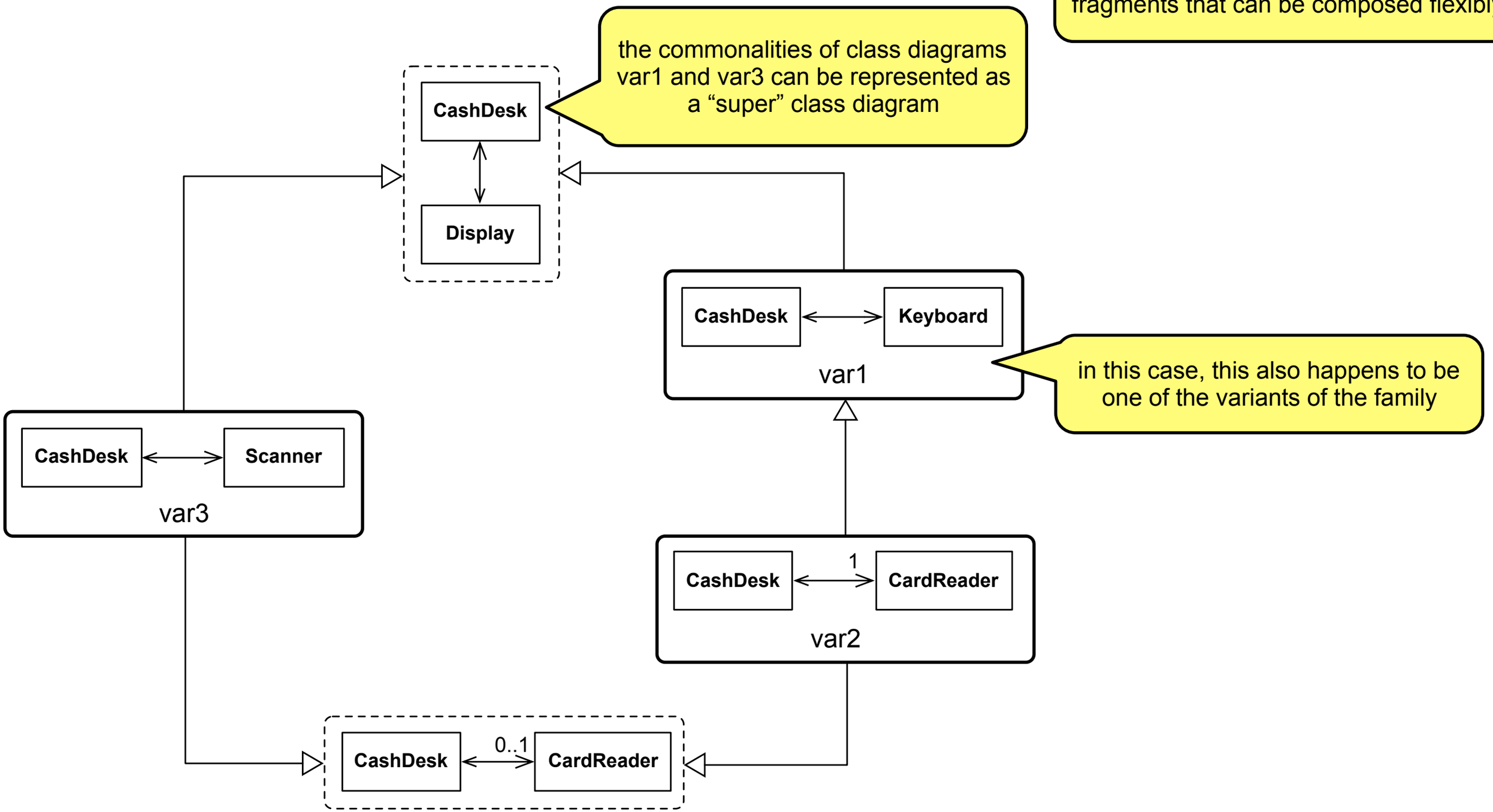
A **compositional** mechanism allows the artefacts to be split into reusable fragments that can be composed flexibly.



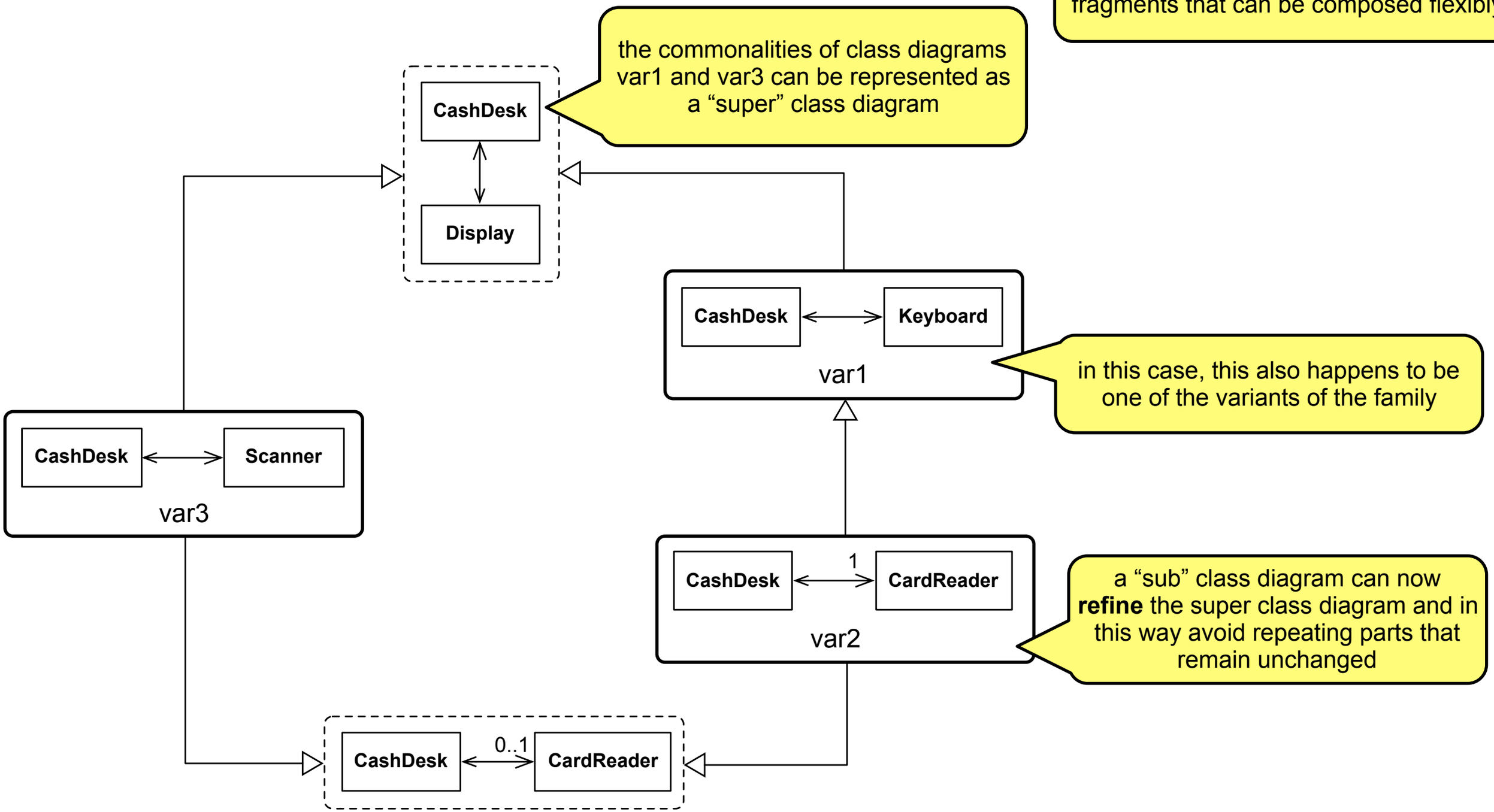
A **compositional** mechanism allows the artefacts to be split into reusable fragments that can be composed flexibly.



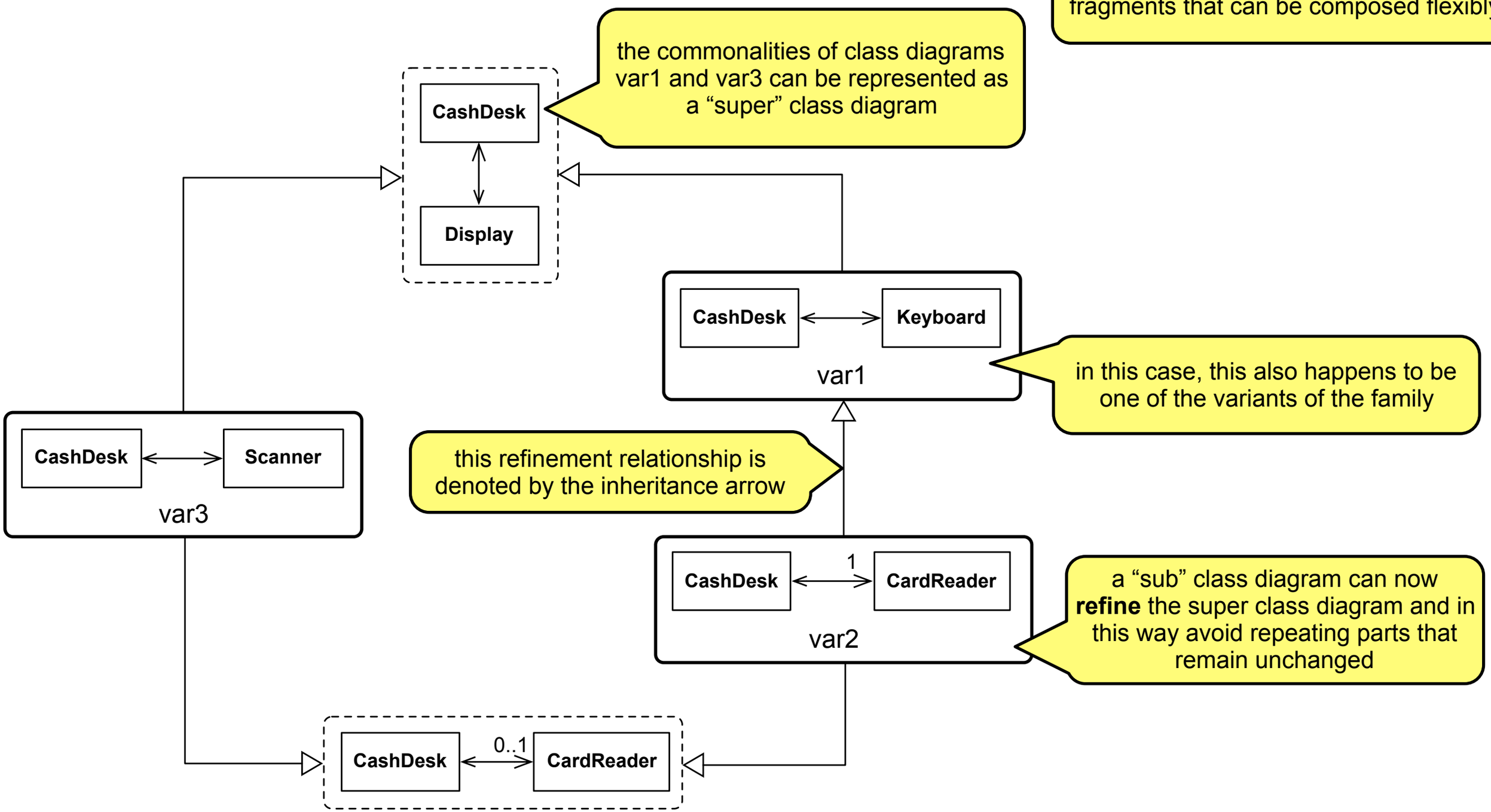
A **compositional** mechanism allows the artefacts to be split into reusable fragments that can be composed flexibly.



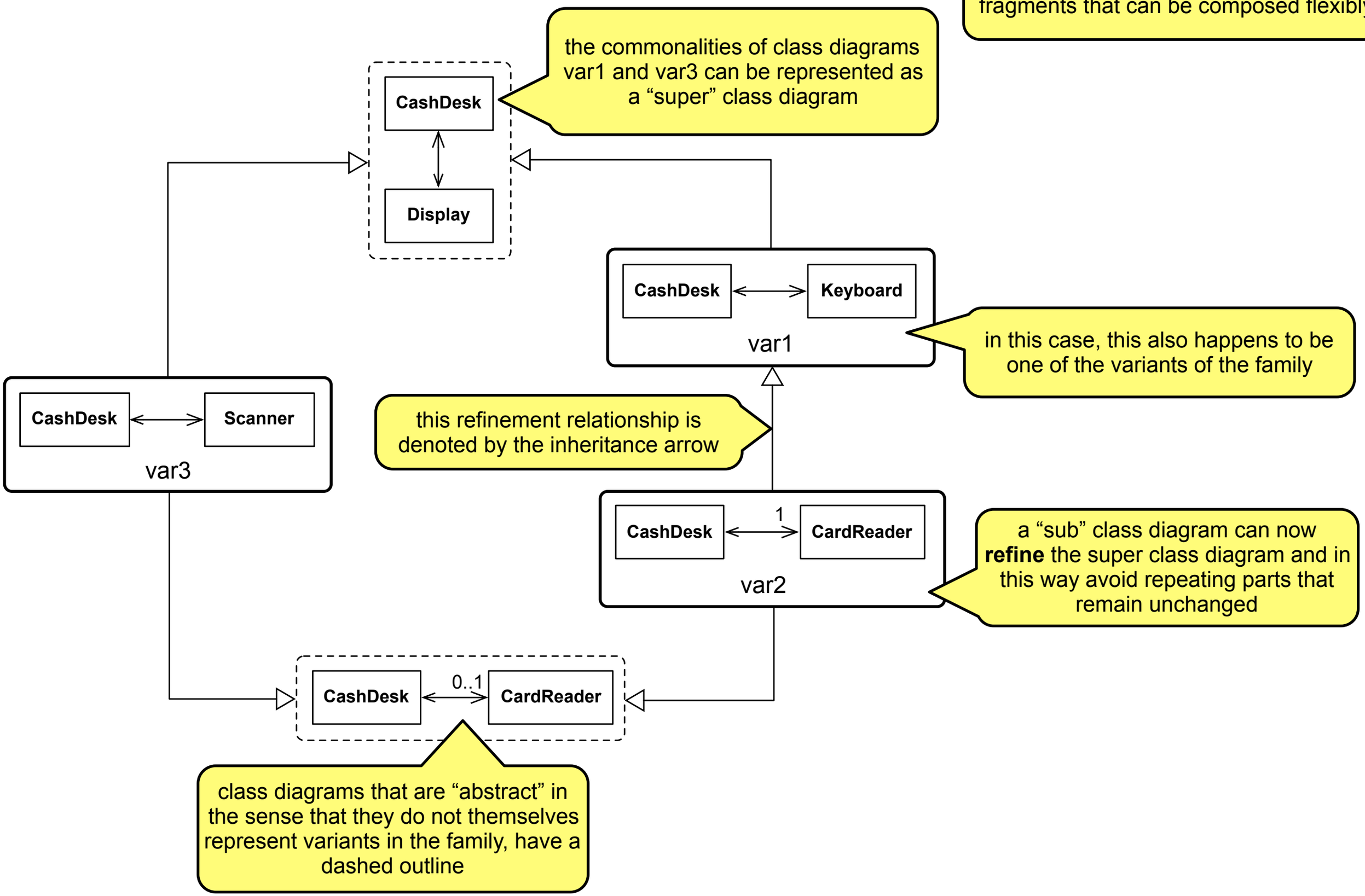
A **compositional** mechanism allows the artefacts to be split into reusable fragments that can be composed flexibly.

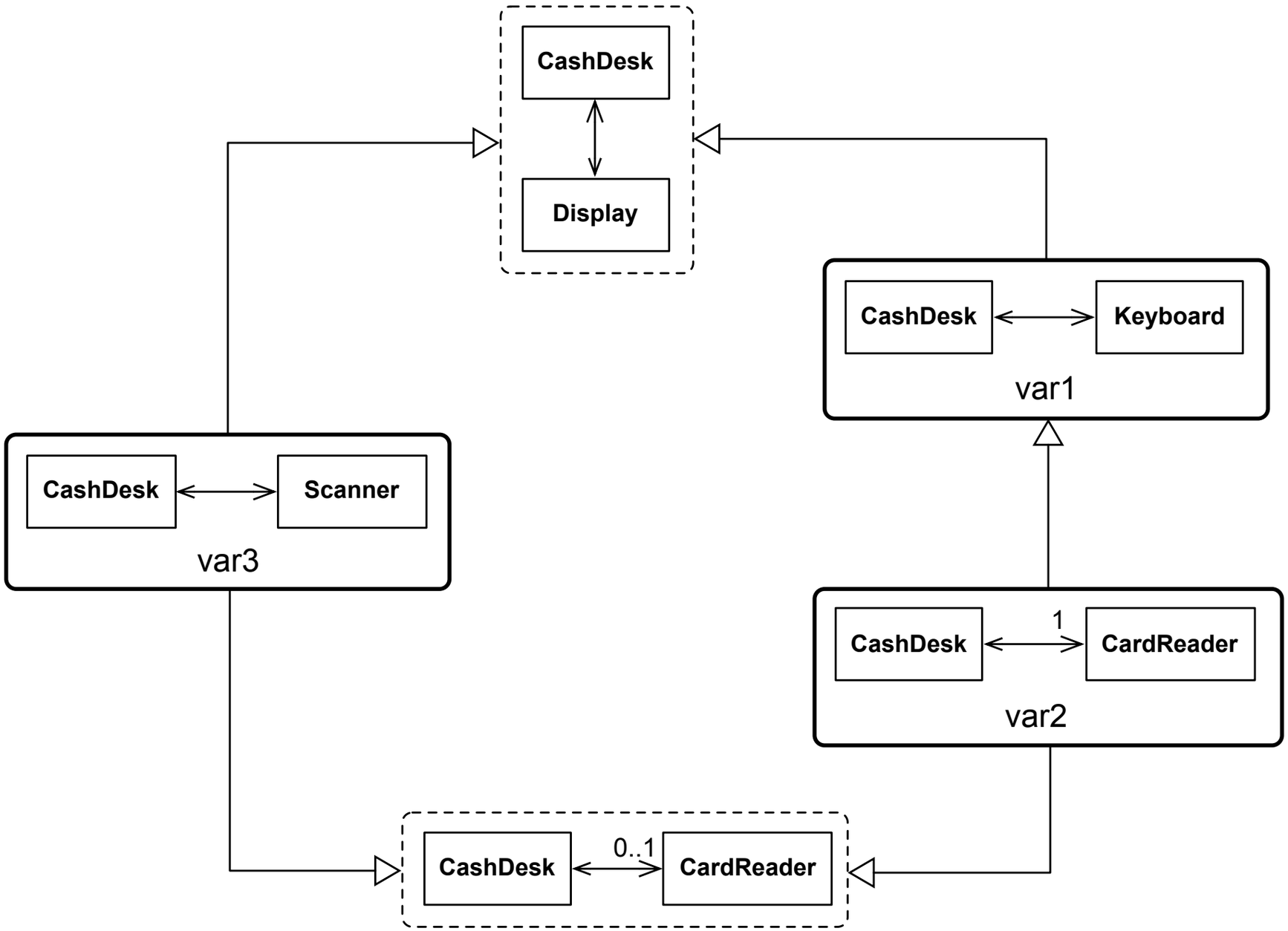


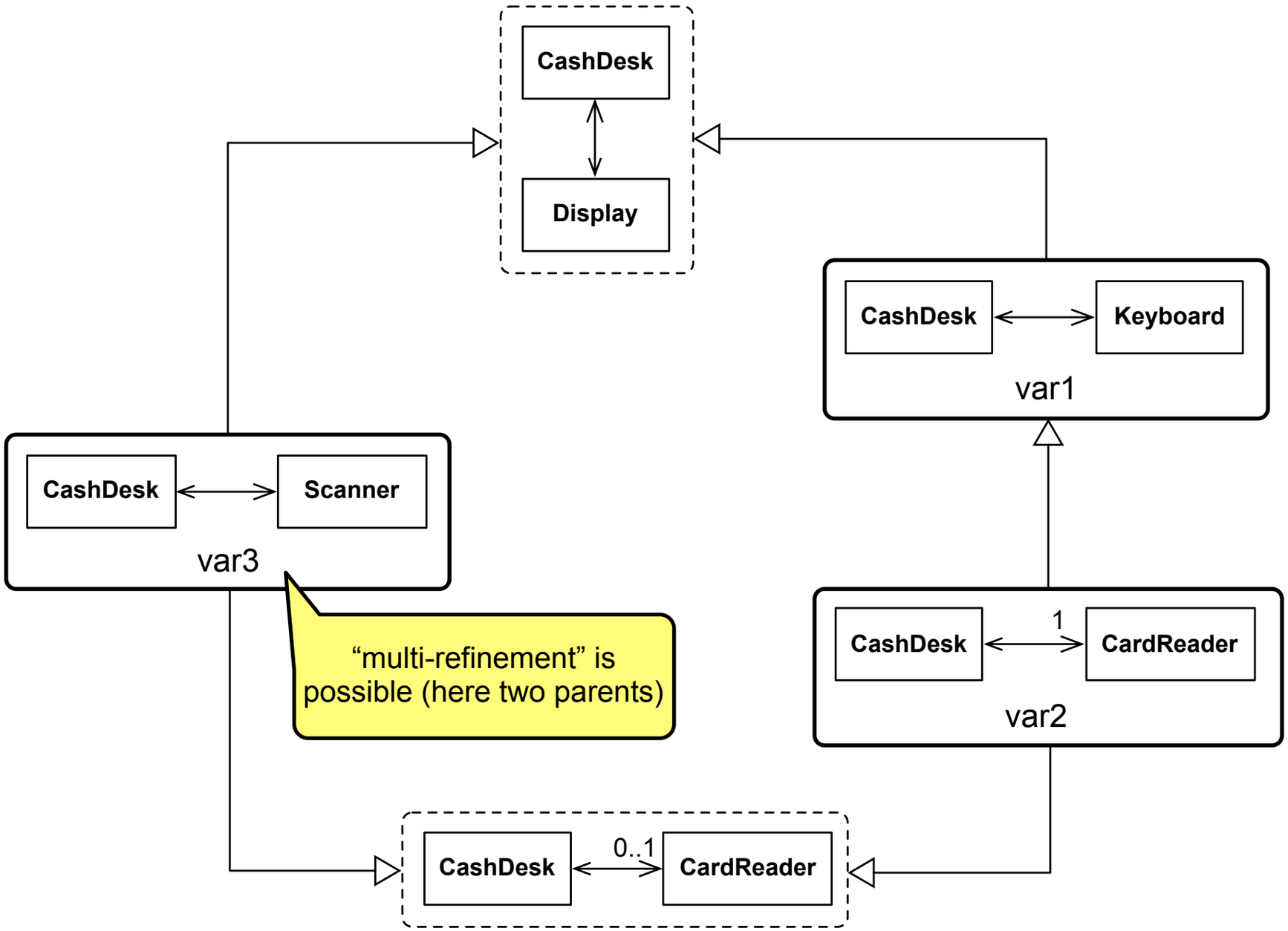
A **compositional** mechanism allows the artefacts to be split into reusable fragments that can be composed flexibly.

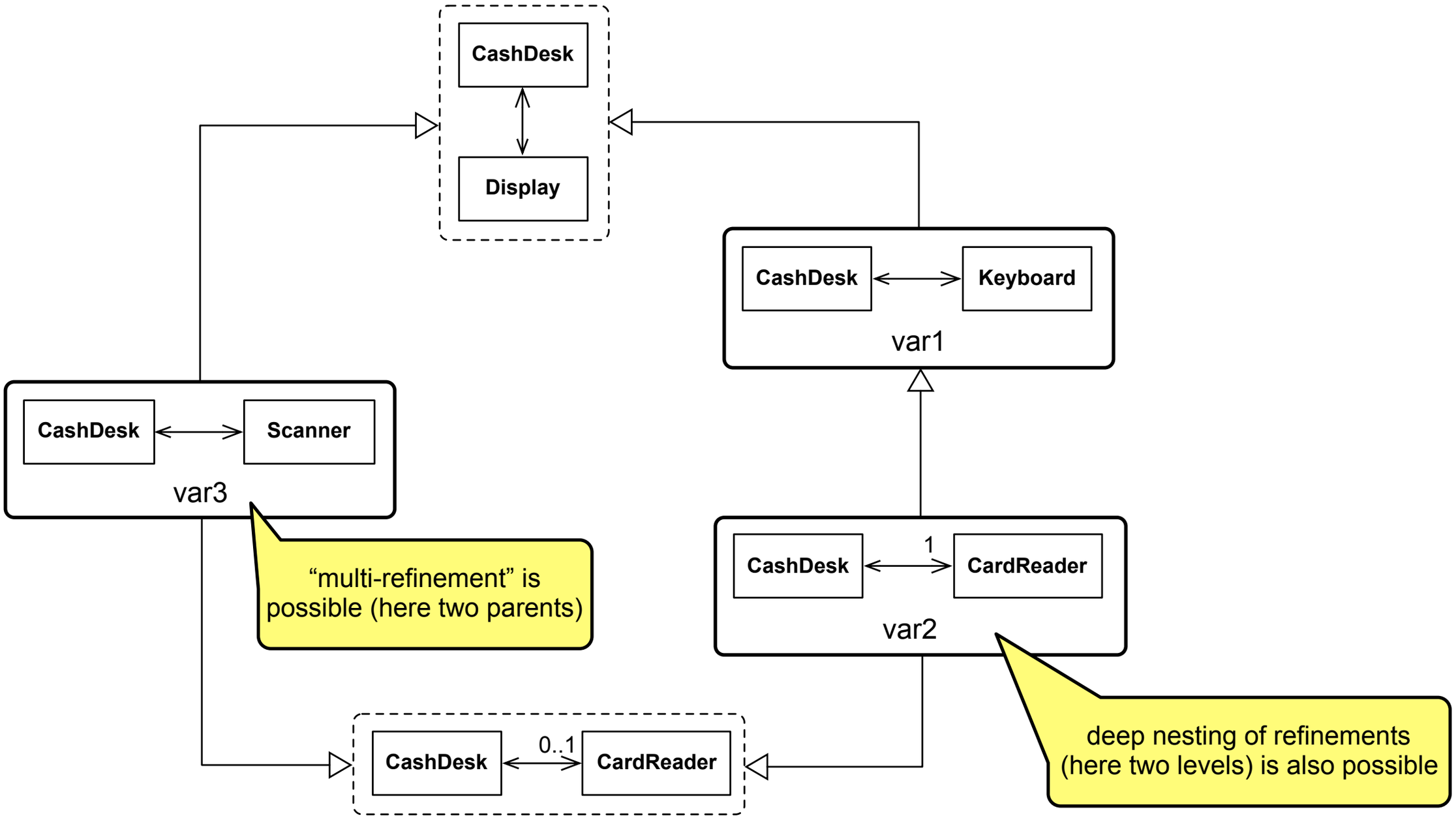


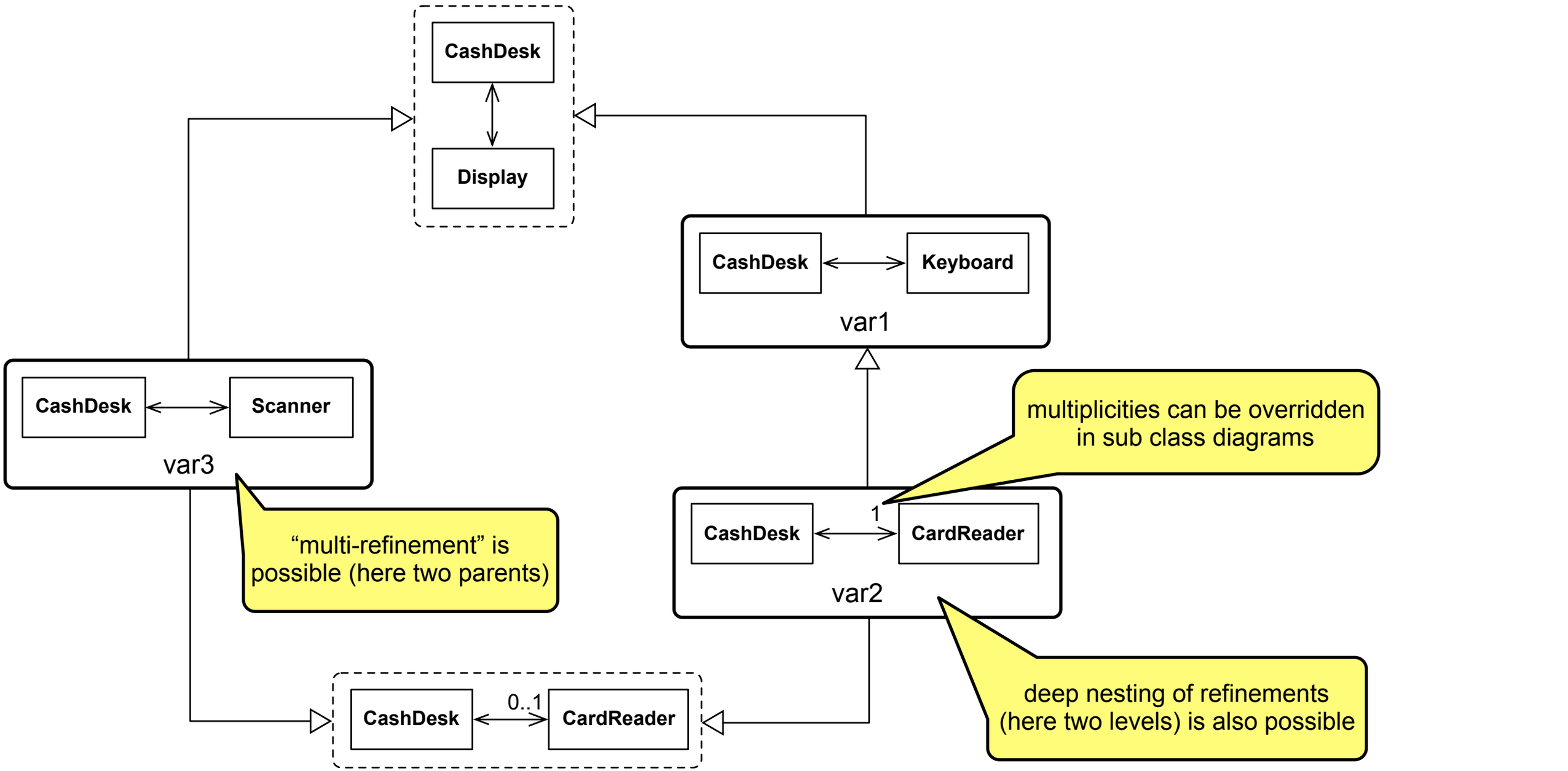
A **compositional** mechanism allows the artefacts to be split into reusable fragments that can be composed flexibly.

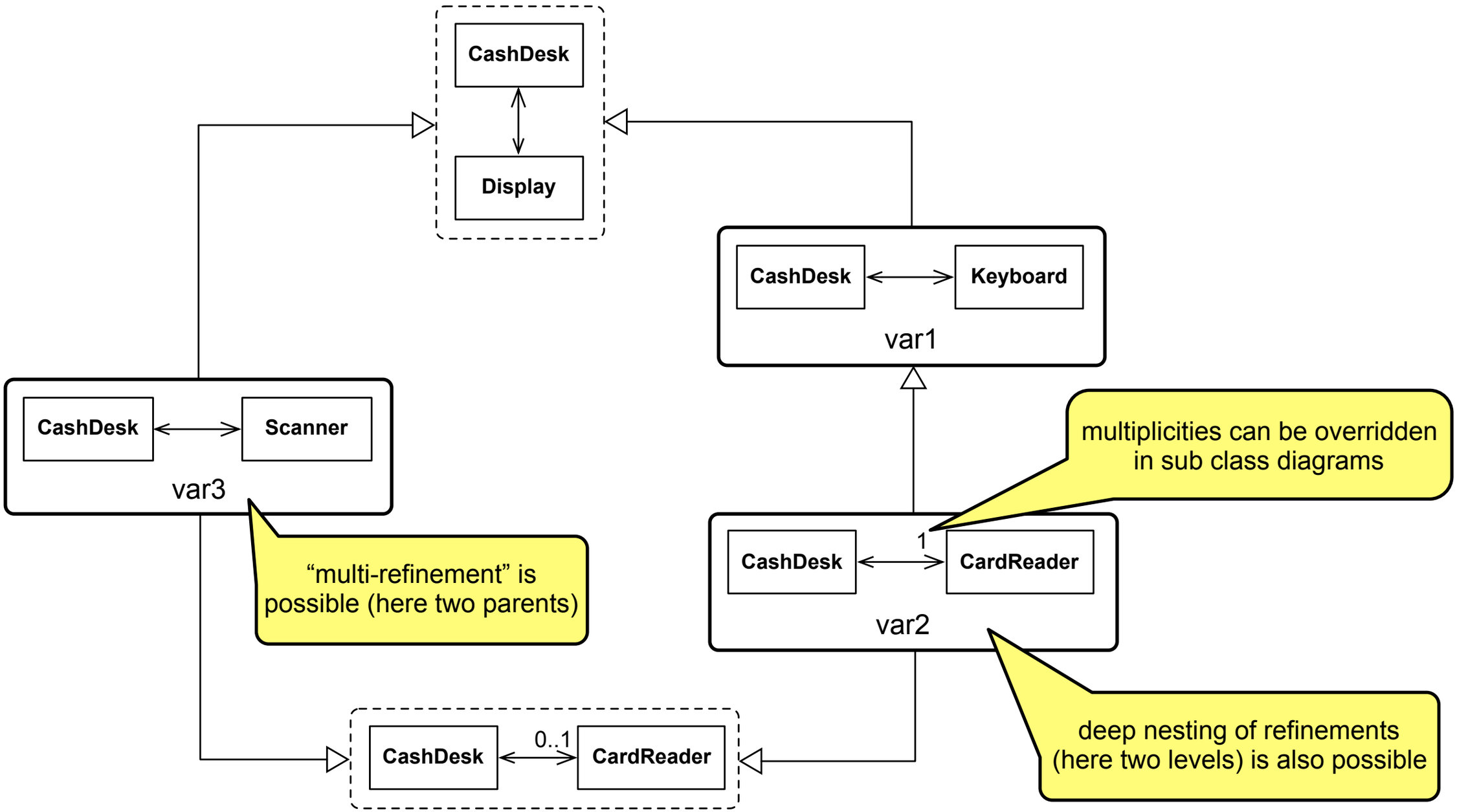



CashDesk
Display
CashDesk
Keyboard
var1
CashDesk
Scanner
var3
CashDesk
1
CardReader
var2
CashDesk
0..1
CardReader

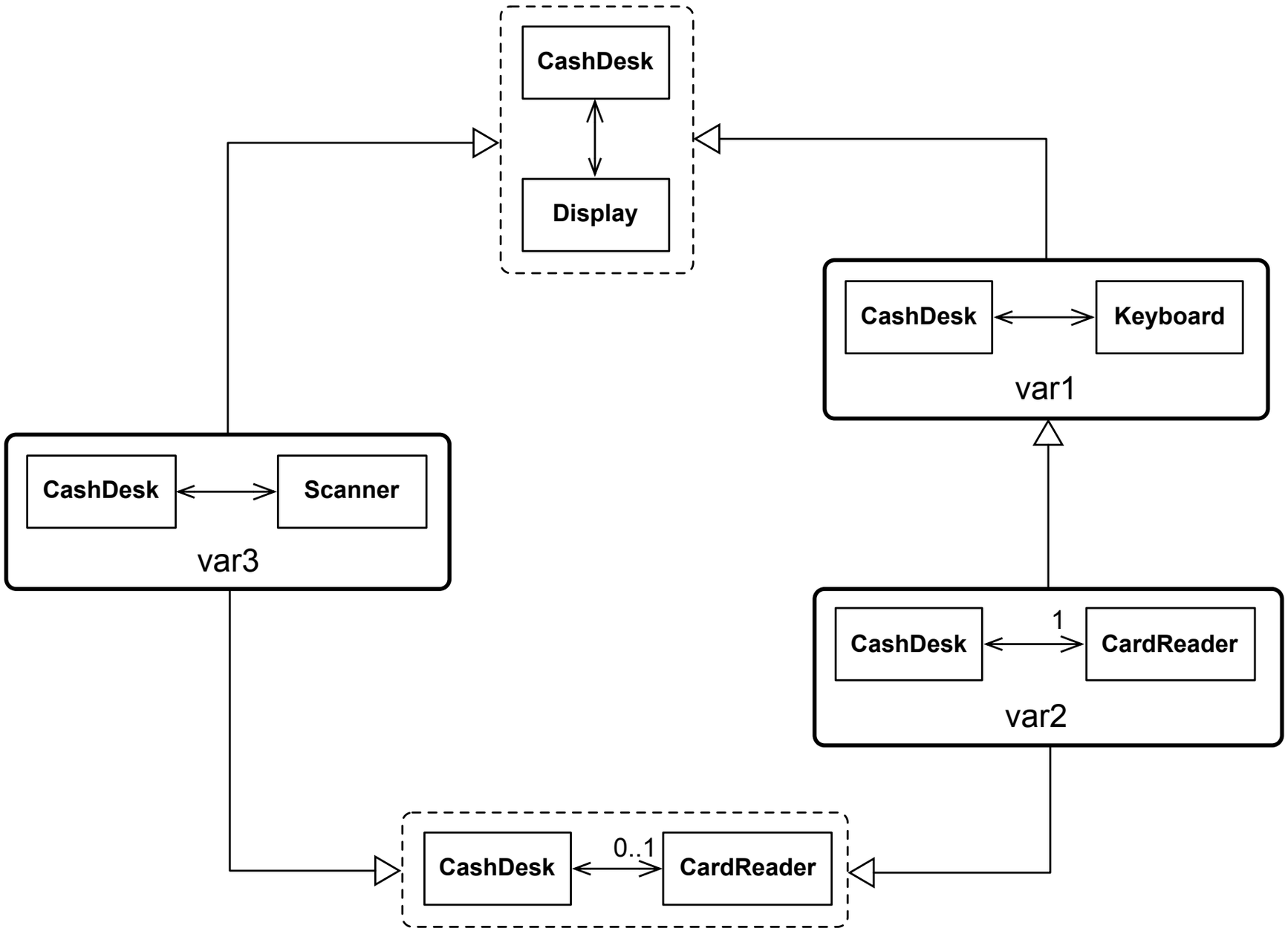

CashDesk
Display
CashDesk
Keyboard
var1
CashDesk
Scanner
var3
“multi-refinement” is possible (here two parents)
CashDesk
1
CardReader
var2
CashDesk
0..1
CardReader

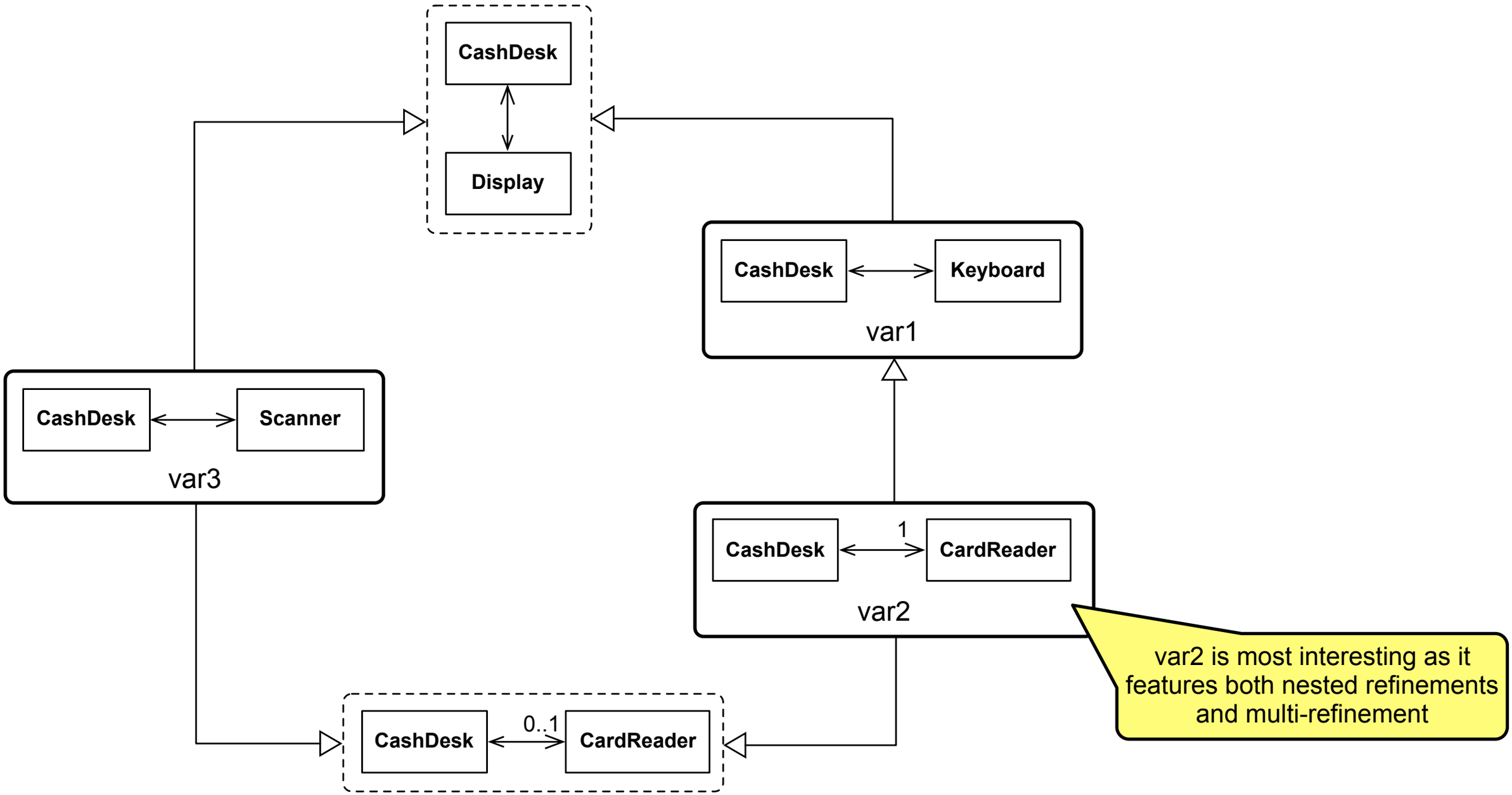

CashDesk
Display
CashDesk
Keyboard
var1
CashDesk
Scanner
var3
“multi-refinement” is possible (here two parents)
CashDesk
1
CardReader
var2
CashDesk
0..1
CardReader
deep nesting of refinements (here two levels) is also possible

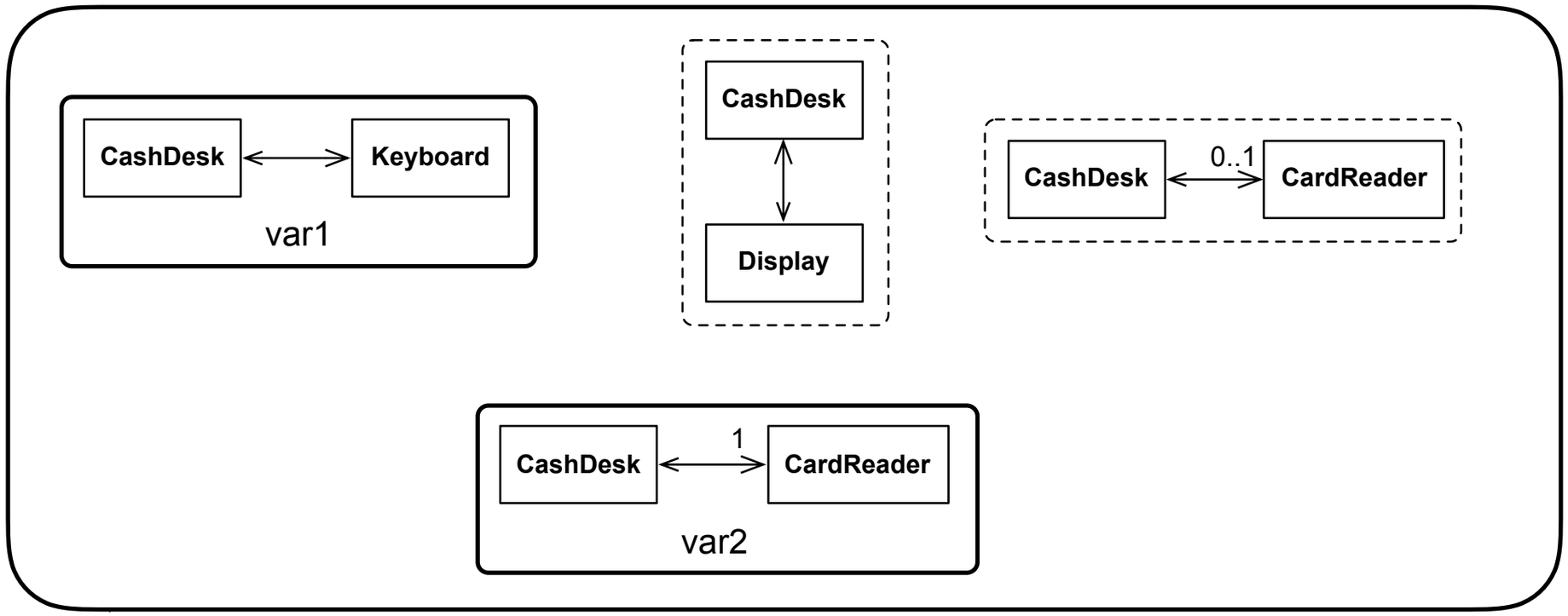

CashDesk
Display
CashDesk
Keyboard
var1
CashDesk
Scanner
var3
multiplicities can be overridden
in sub class diagrams
“multi-refinement” is
possible (here two parents)
CashDesk
1
CardReader
var2
deep nesting of refinements
(here two levels) is also possible
CashDesk
0..1
CardReader



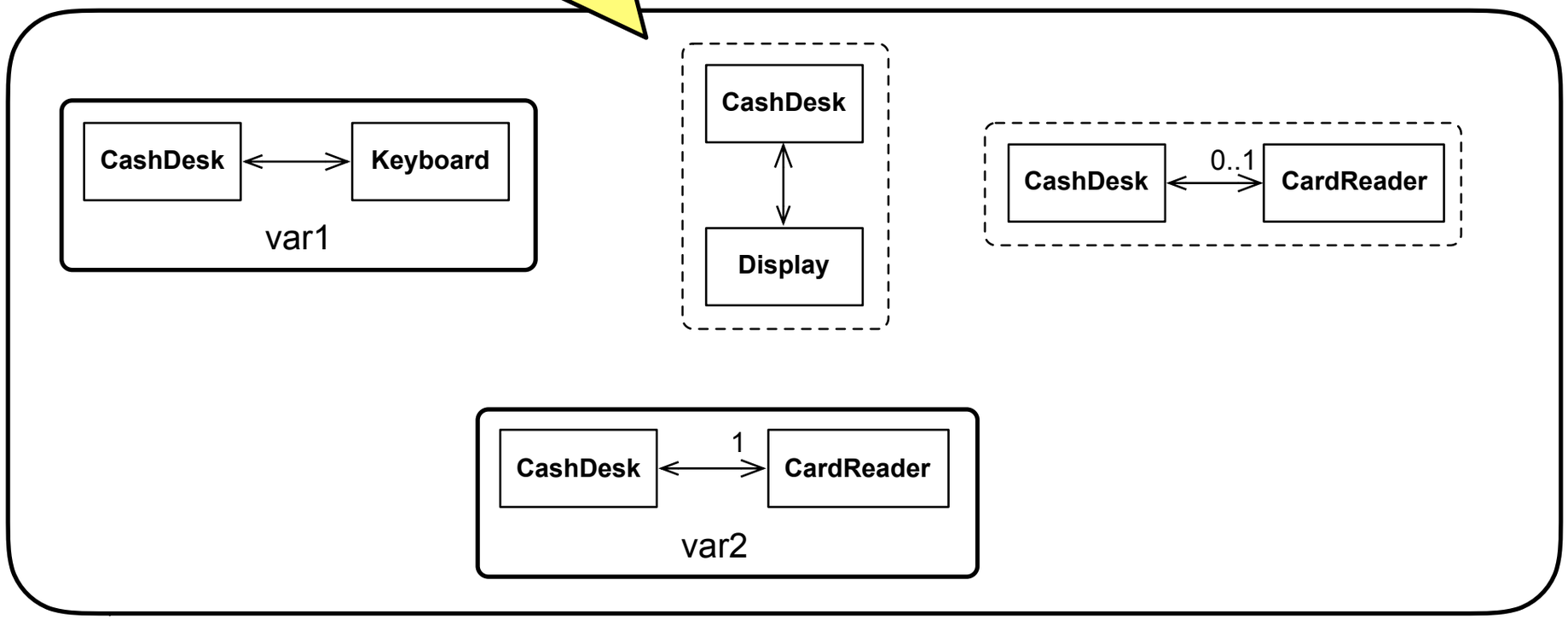
Let’s now see how to derive “flat” variants from such a refinement hierarchy.


CashDesk
Display
CashDesk
Keyboard
var1
CashDesk
Scanner
var3
CashDesk
1
CardReader
var2
CashDesk
0..1
CardReader

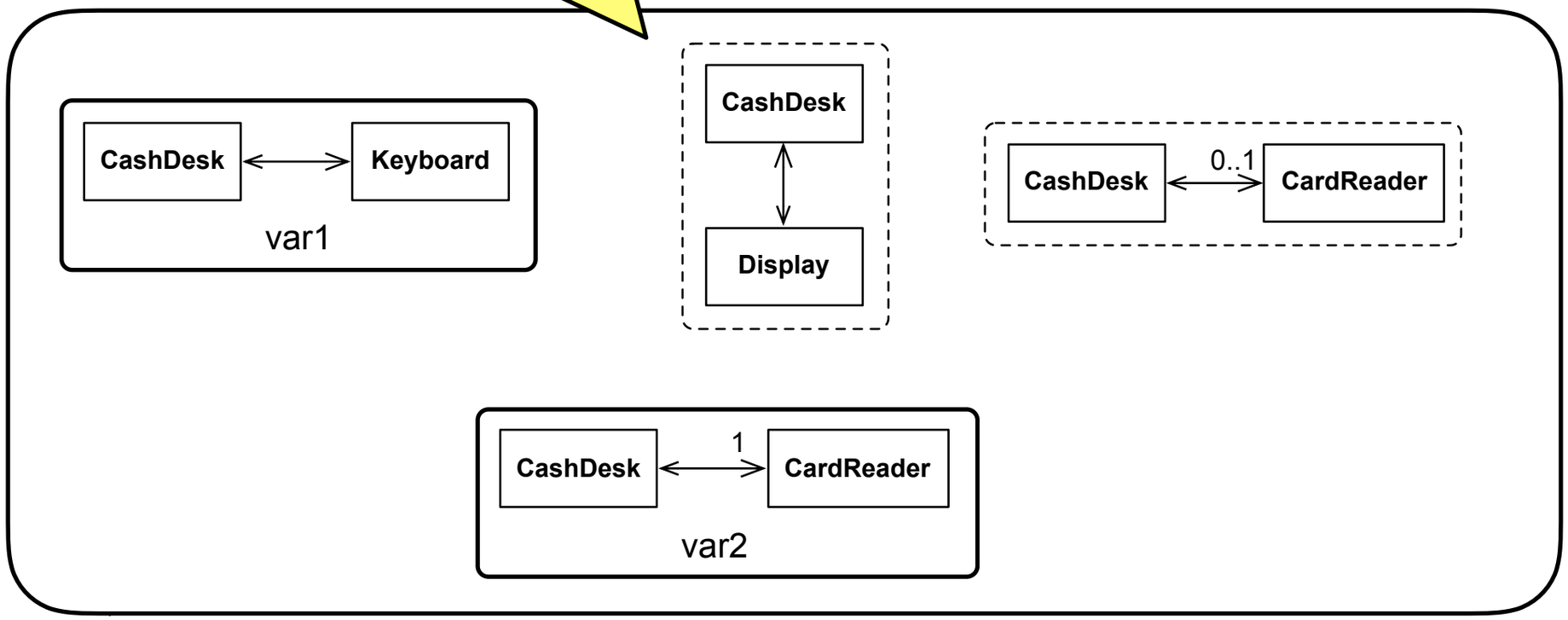

CashDesk
Display
CashDesk
Keyboard
var1
CashDesk
Scanner
var3
CashDesk
1
CardReader
var2
var2 is most interesting as it features both nested refinements and multi-refinement
CashDesk
0..1
CardReader


CashDesk
Keyboard
var1
CashDesk
Display
CashDesk
0..1
CardReader
CashDesk
1
CardReader
var2

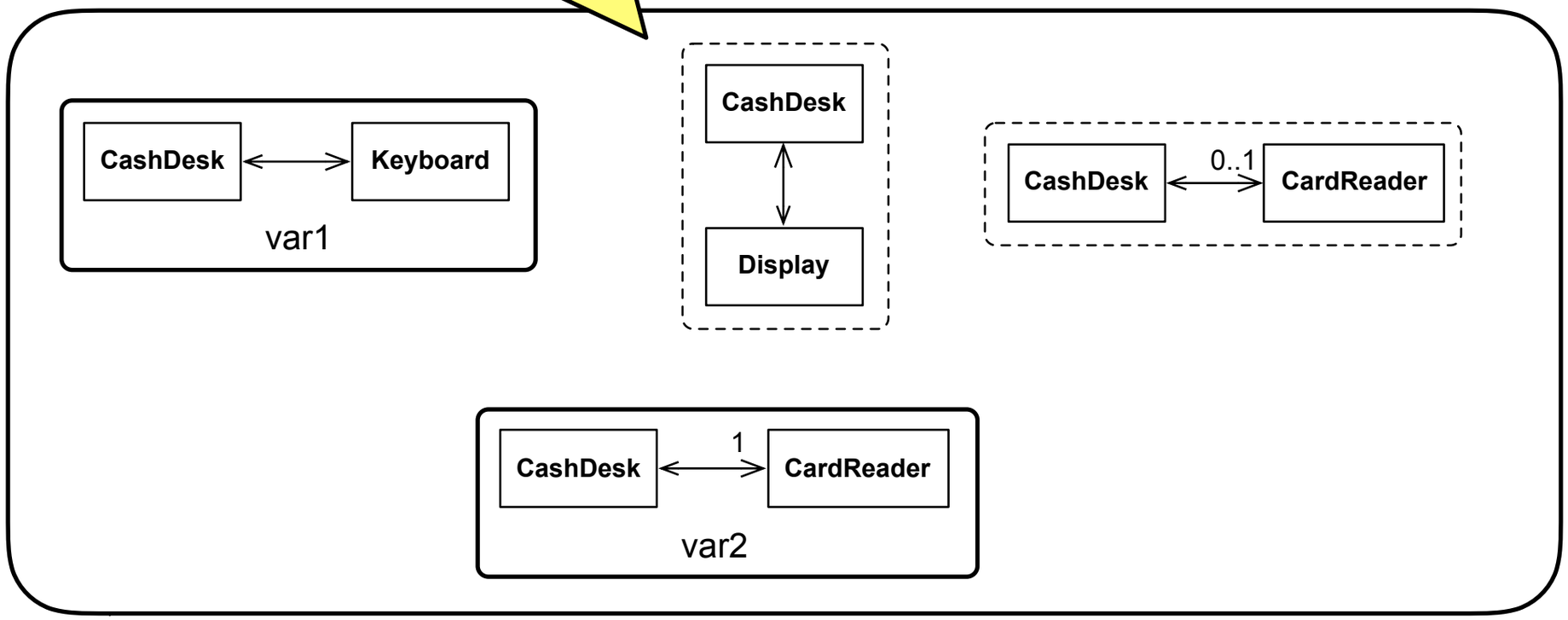
Step 1 (Union): Compute all transitive parents of the variant to be flattened and build a union of all these class diagrams together with the variant itself (here var2)

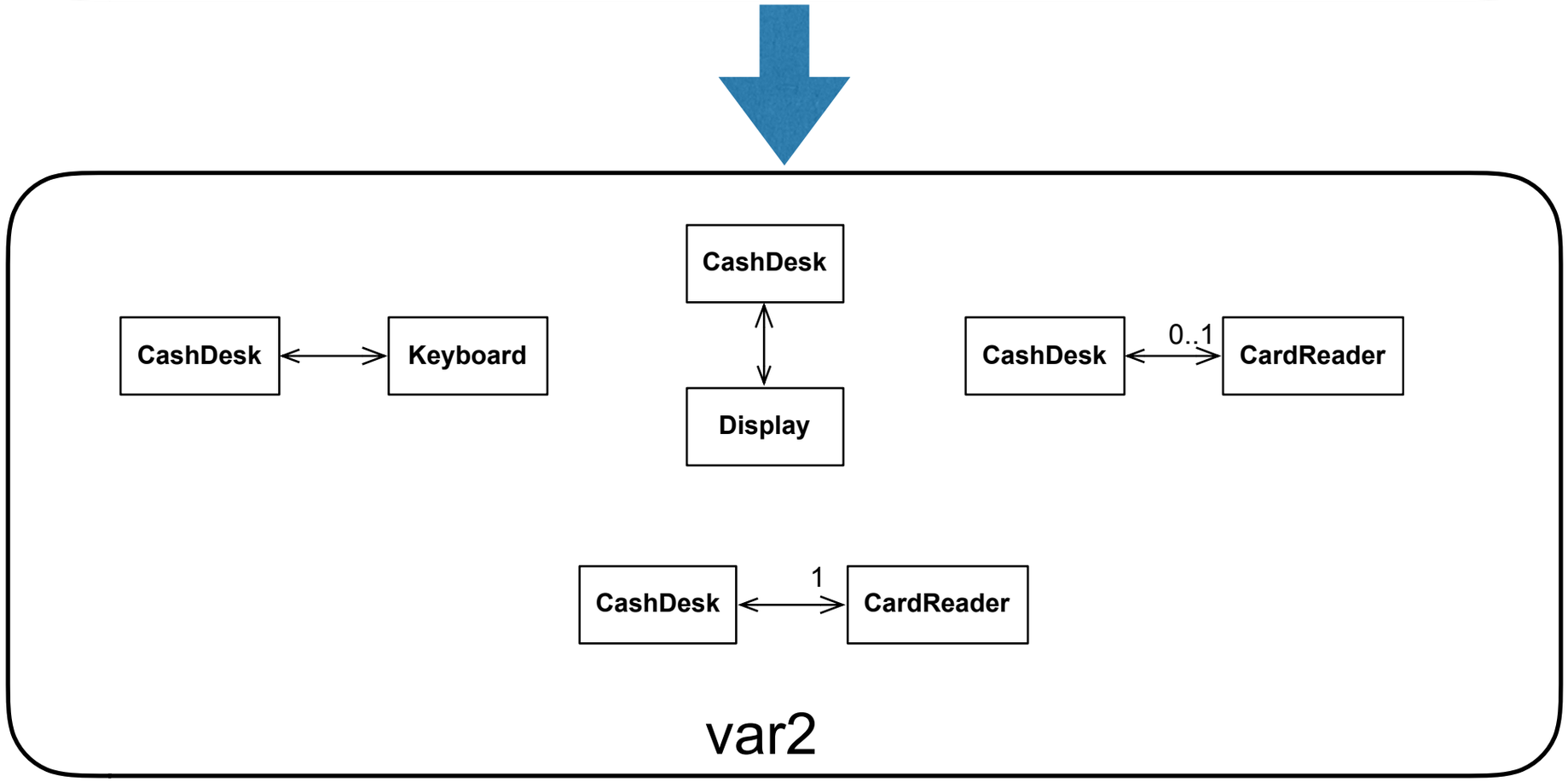


Step 1 (Union): Compute all transitive parents of the variant to be flattened and build a union of all these class diagrams together with the variant itself (here var2)

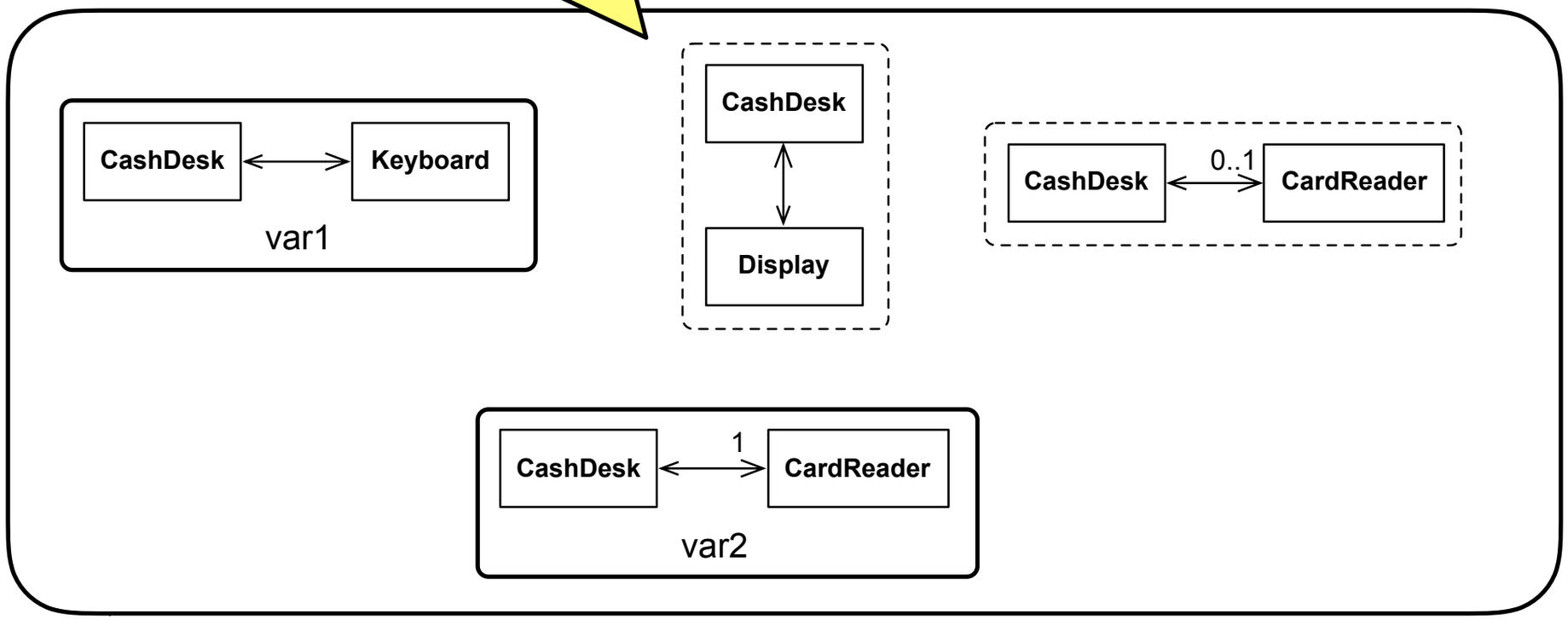


Step 1 (Union): Compute all transitive parents of the variant to be flattened and build a union of all these class diagrams together with the variant itself (here var2)

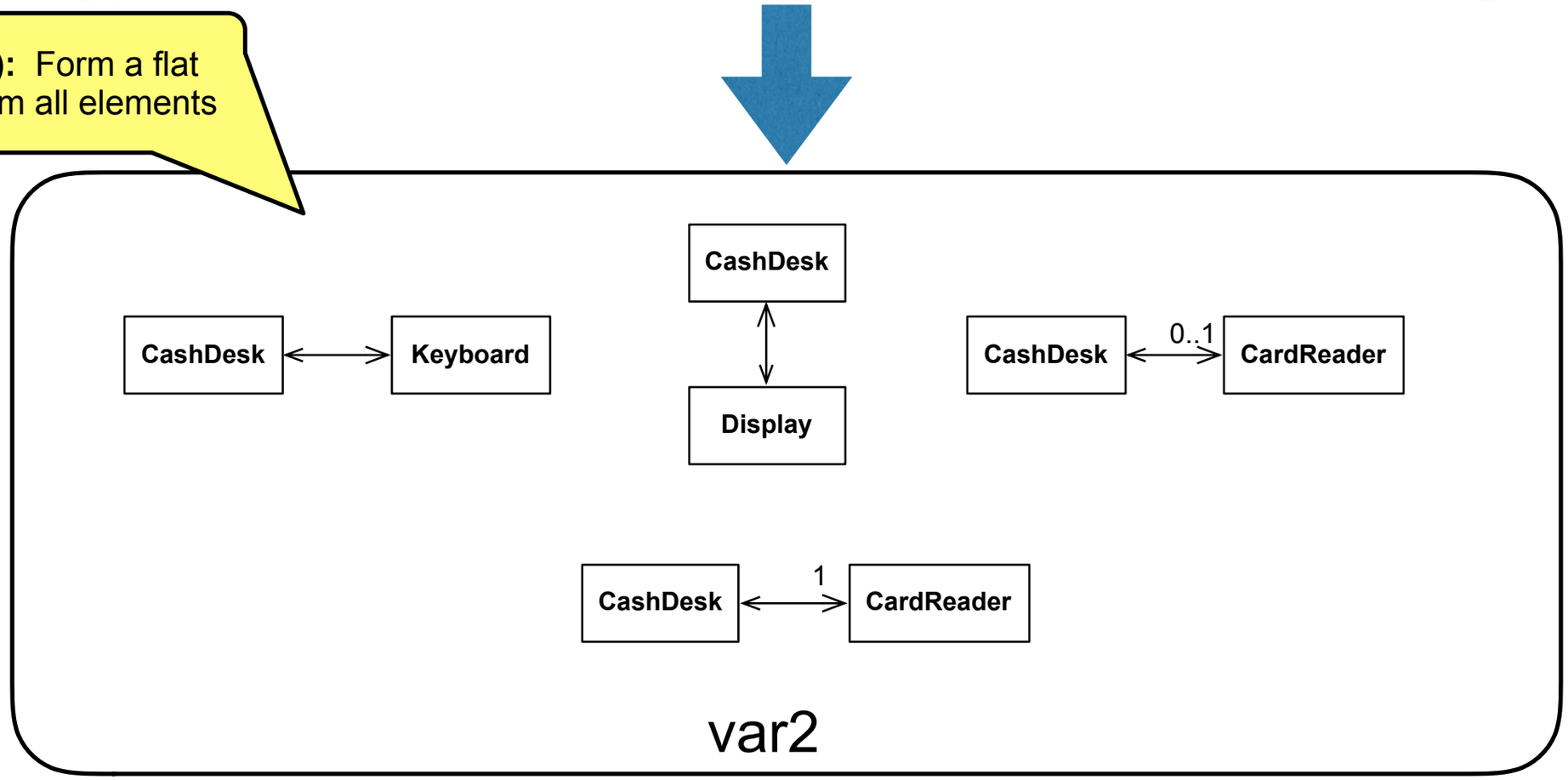




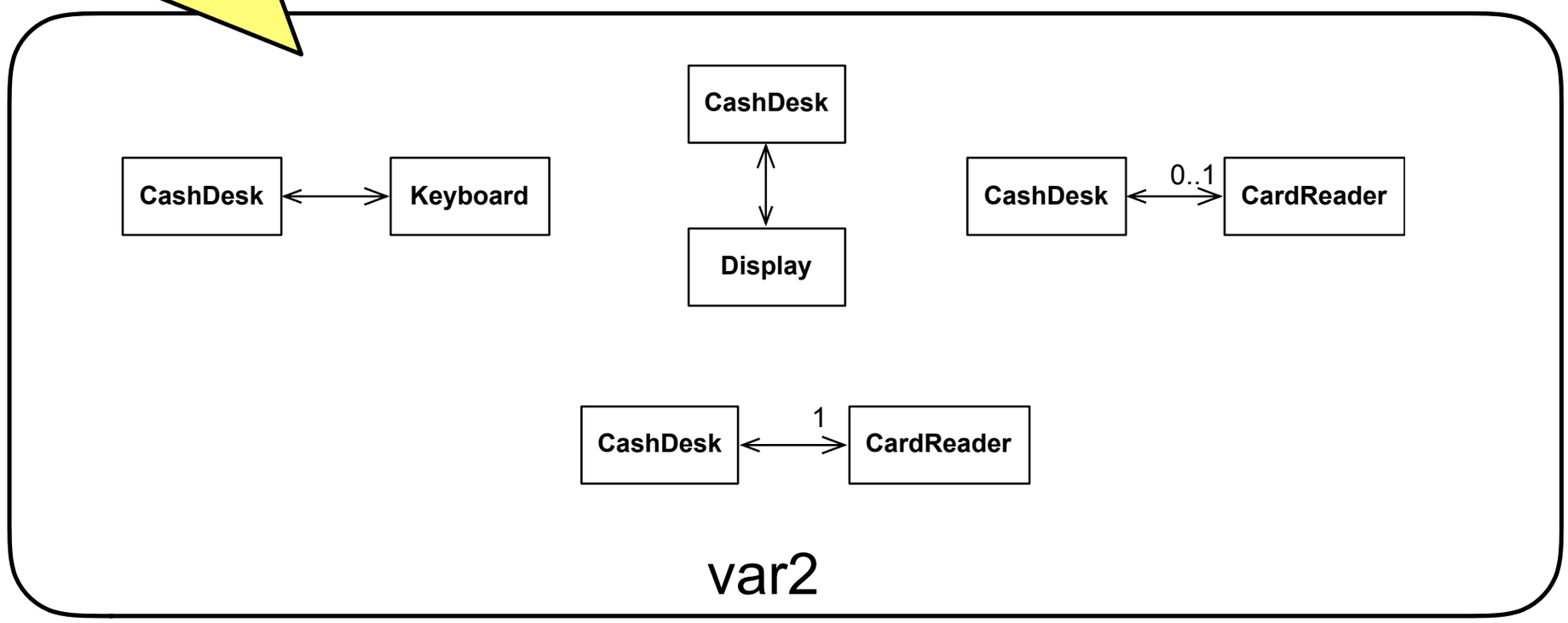
Step 1 (Union): Compute all transitive parents of the variant to be flattened and build a union of all these class diagrams together with the variant itself (here var2)



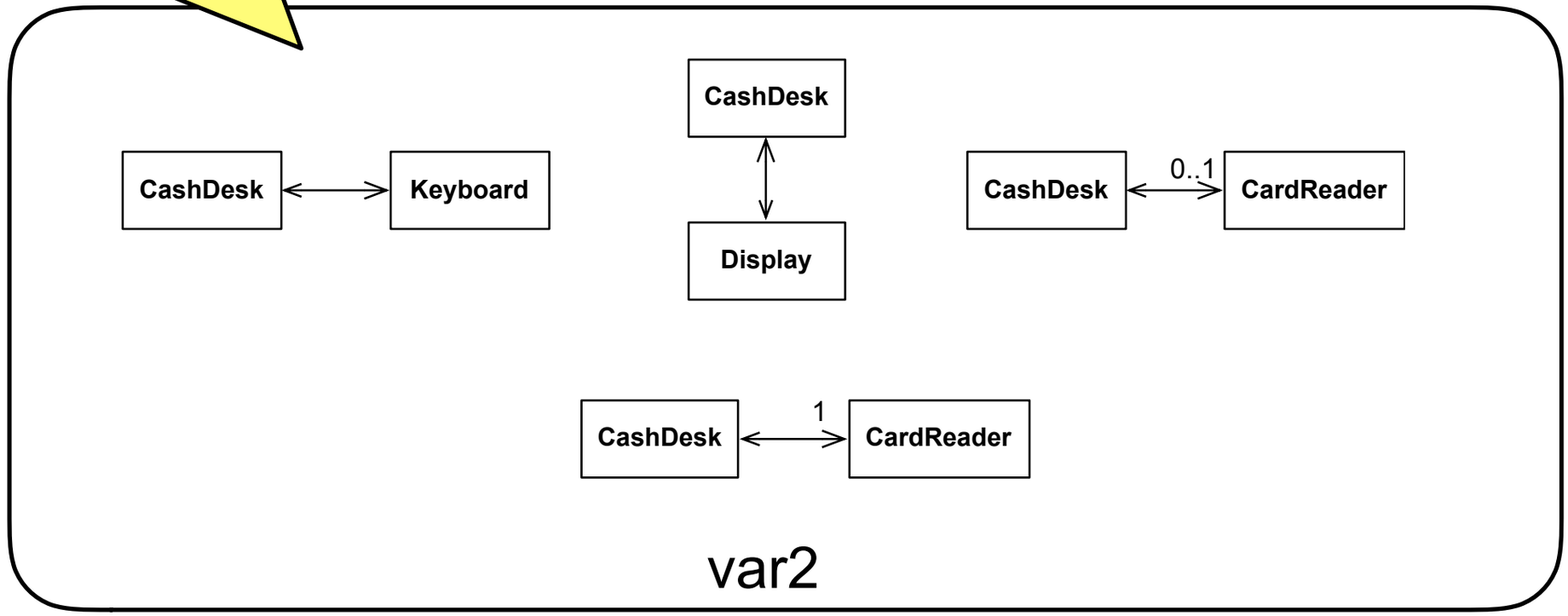
Step 2 (Flatten): Form a flat class diagram from all elements



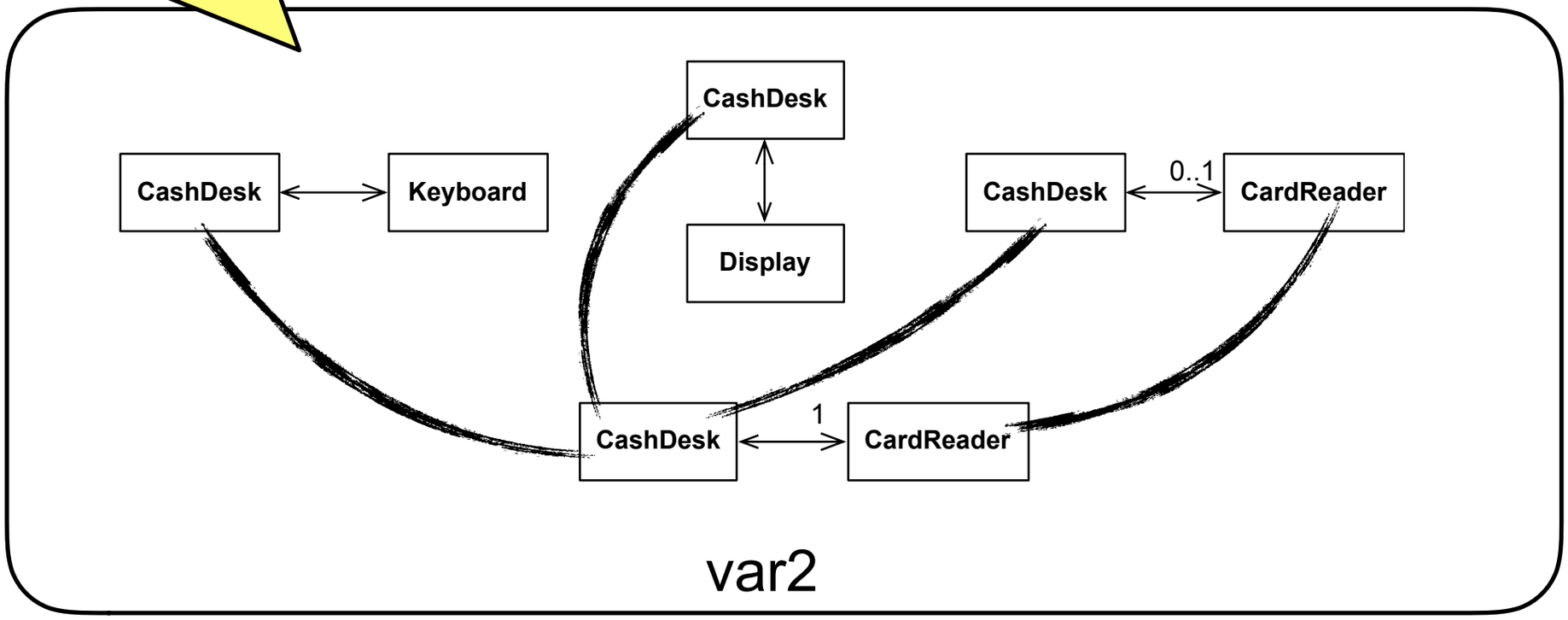
Step 2 (Flatten): Form a flat class diagram from all elements

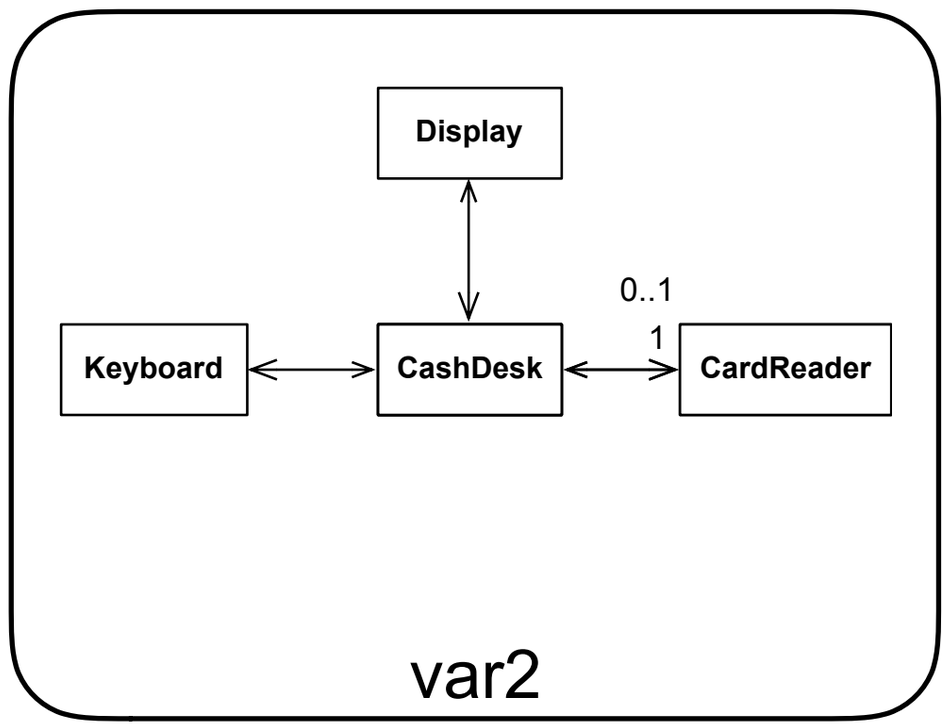


Step 2 (Flatten): Form a flat class diagram from all elements

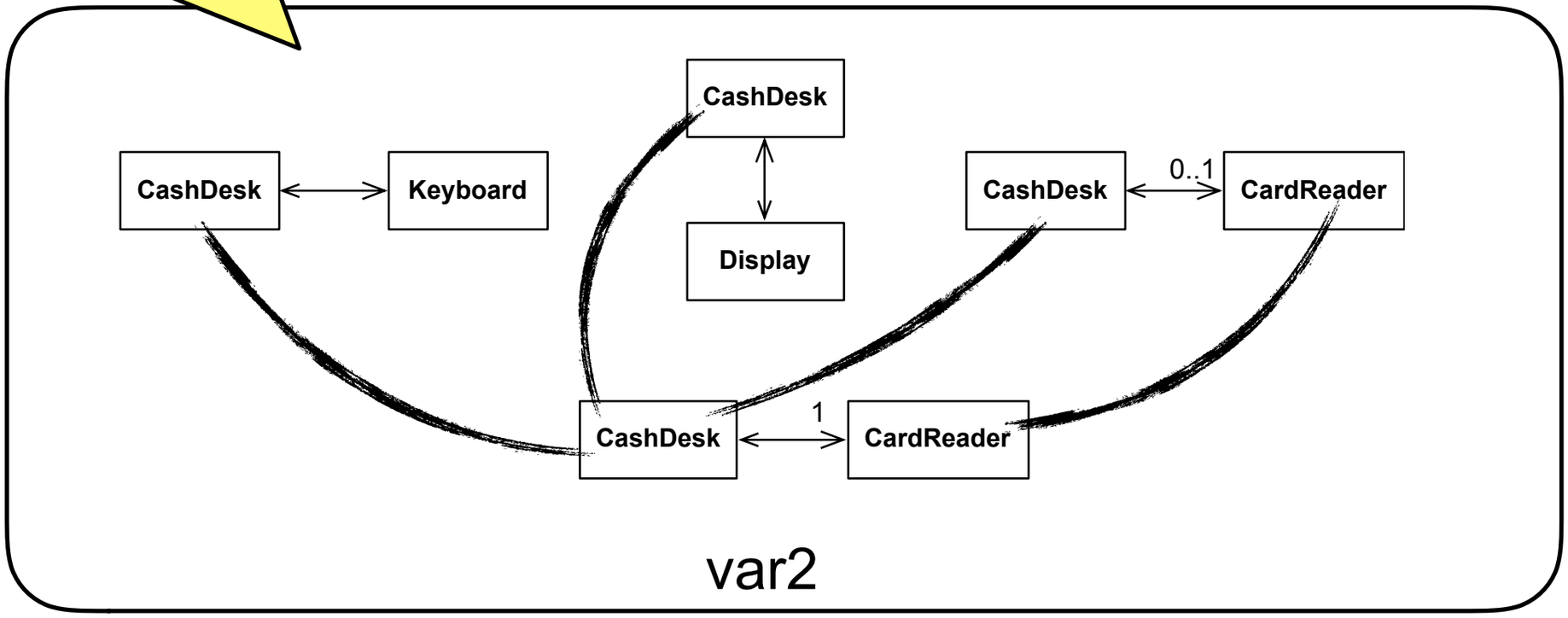


Step 2 (Flatten): Form a flat class diagram from all elements

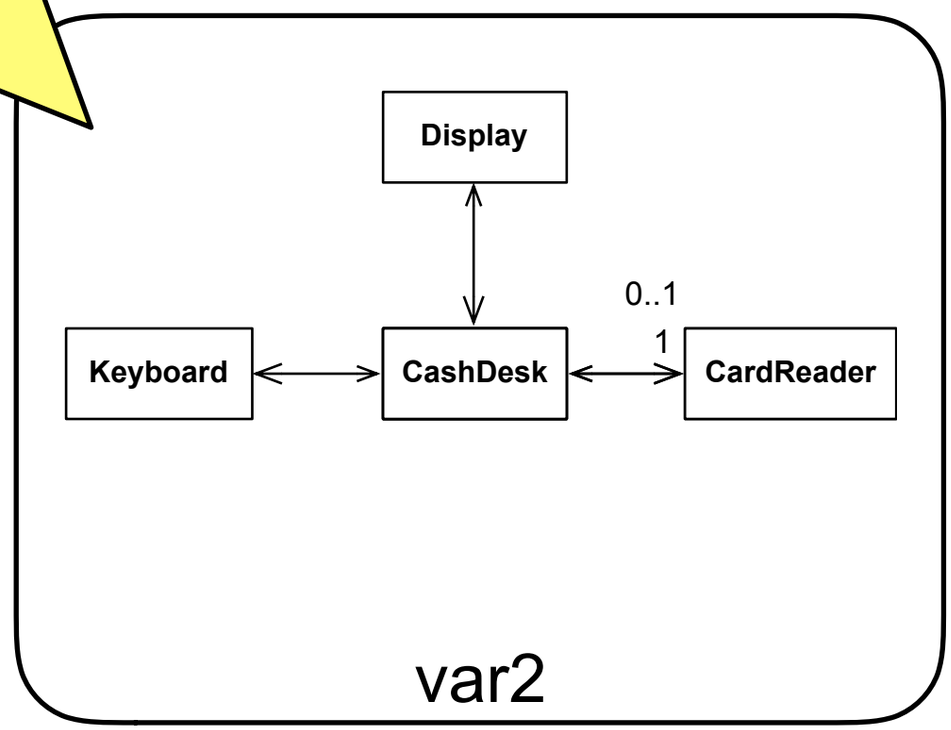




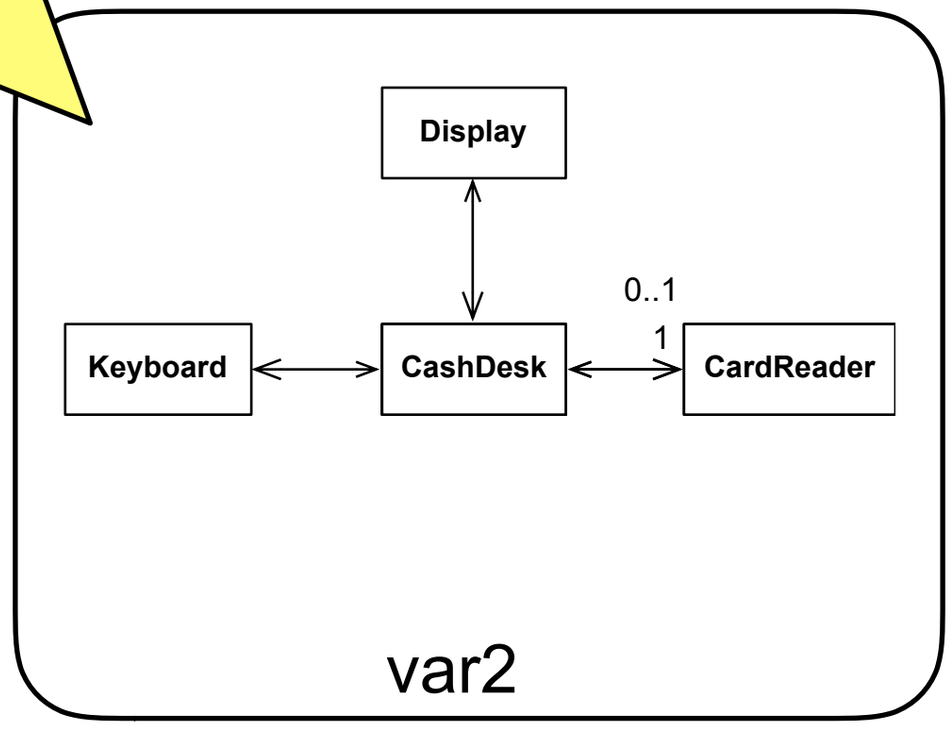
Step 2 (Flatten): Form a flat class diagram from all elements



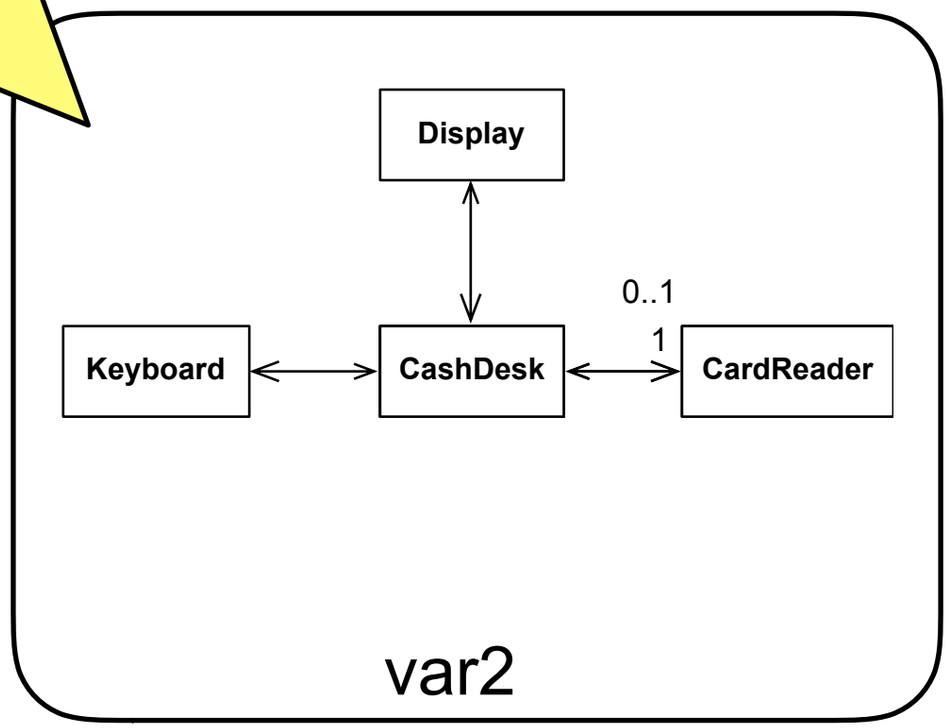
Step 3 (Merge): Merge all elements with the same name together



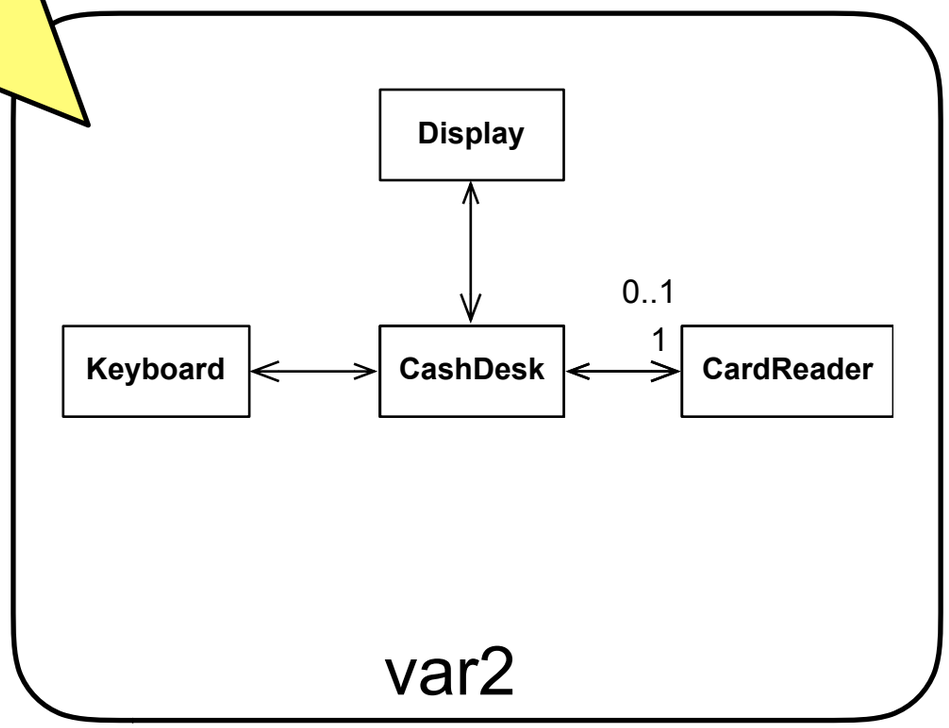
Step 3 (Merge): Merge all elements with the same name together

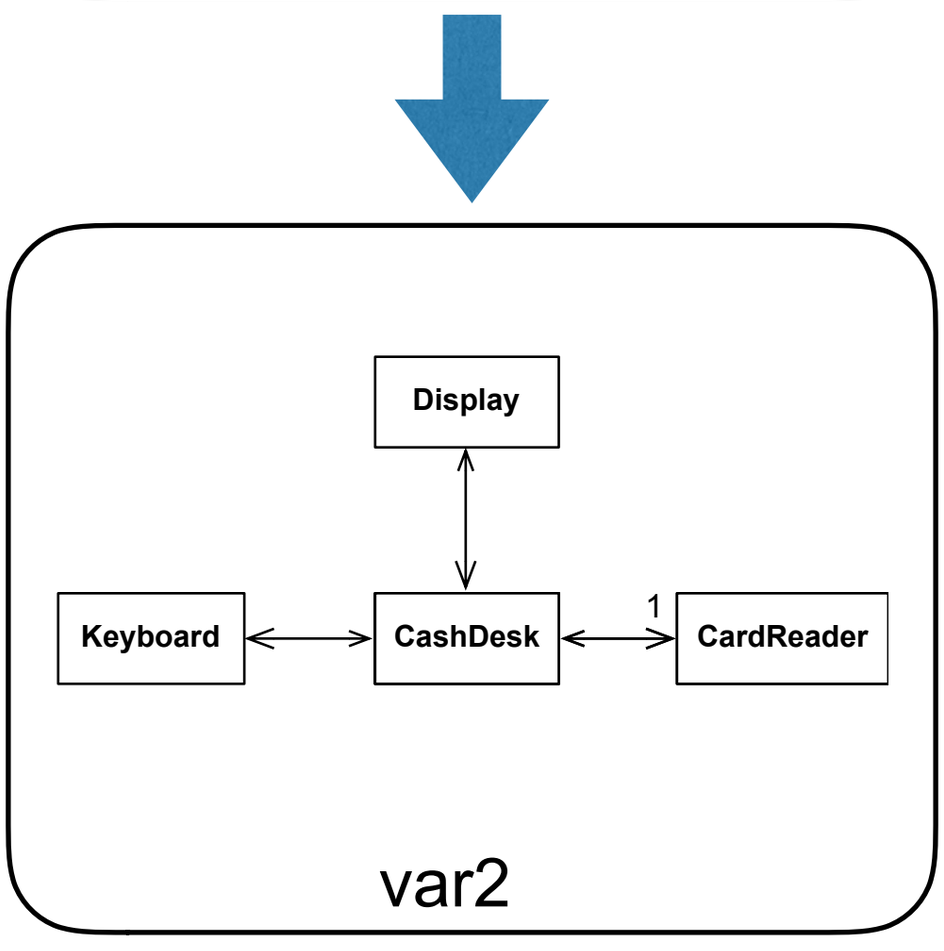


Step 3 (Merge): Merge all elements with the same name together

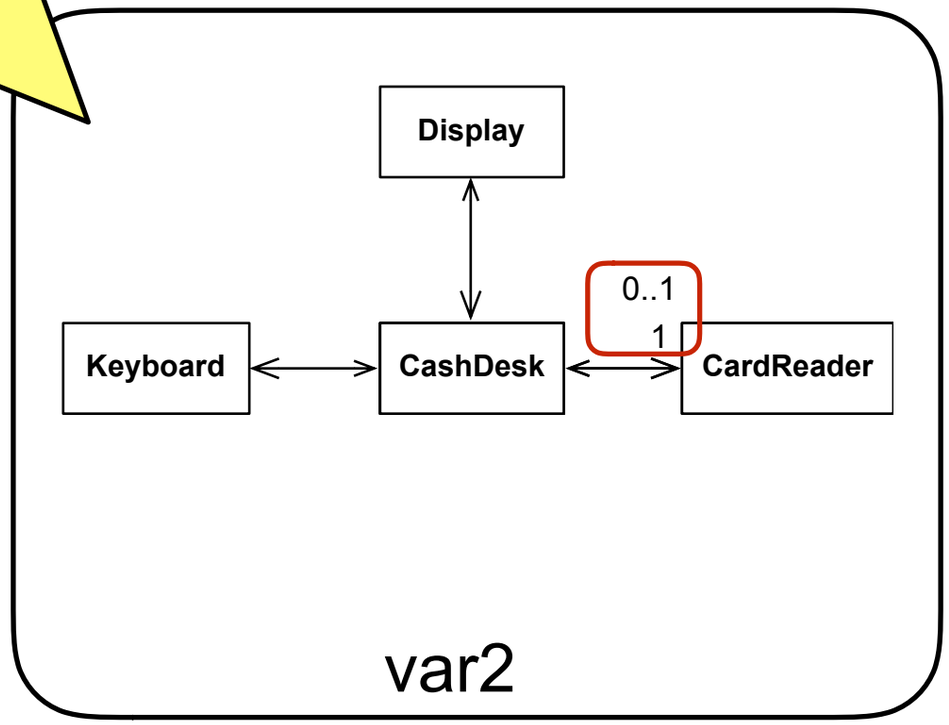


Step 3 (Merge): Merge all elements with the same name together

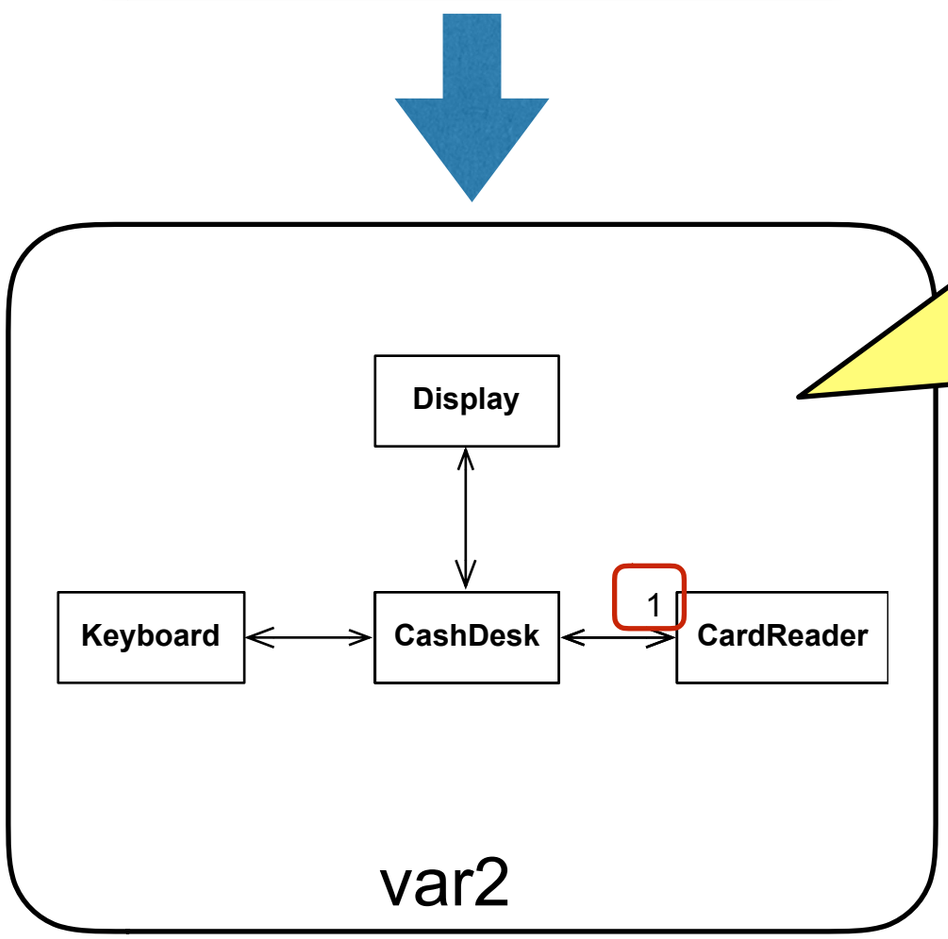


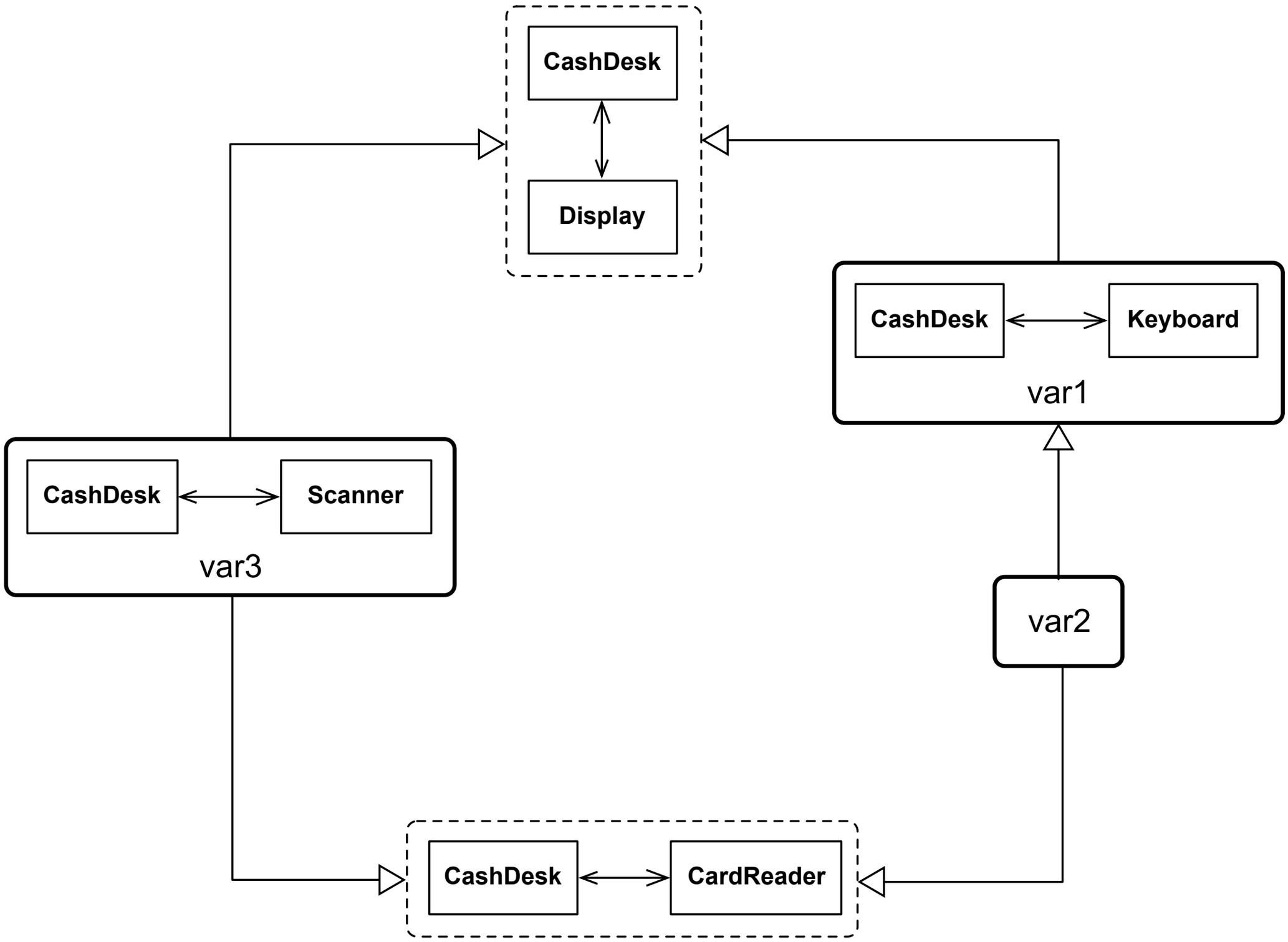


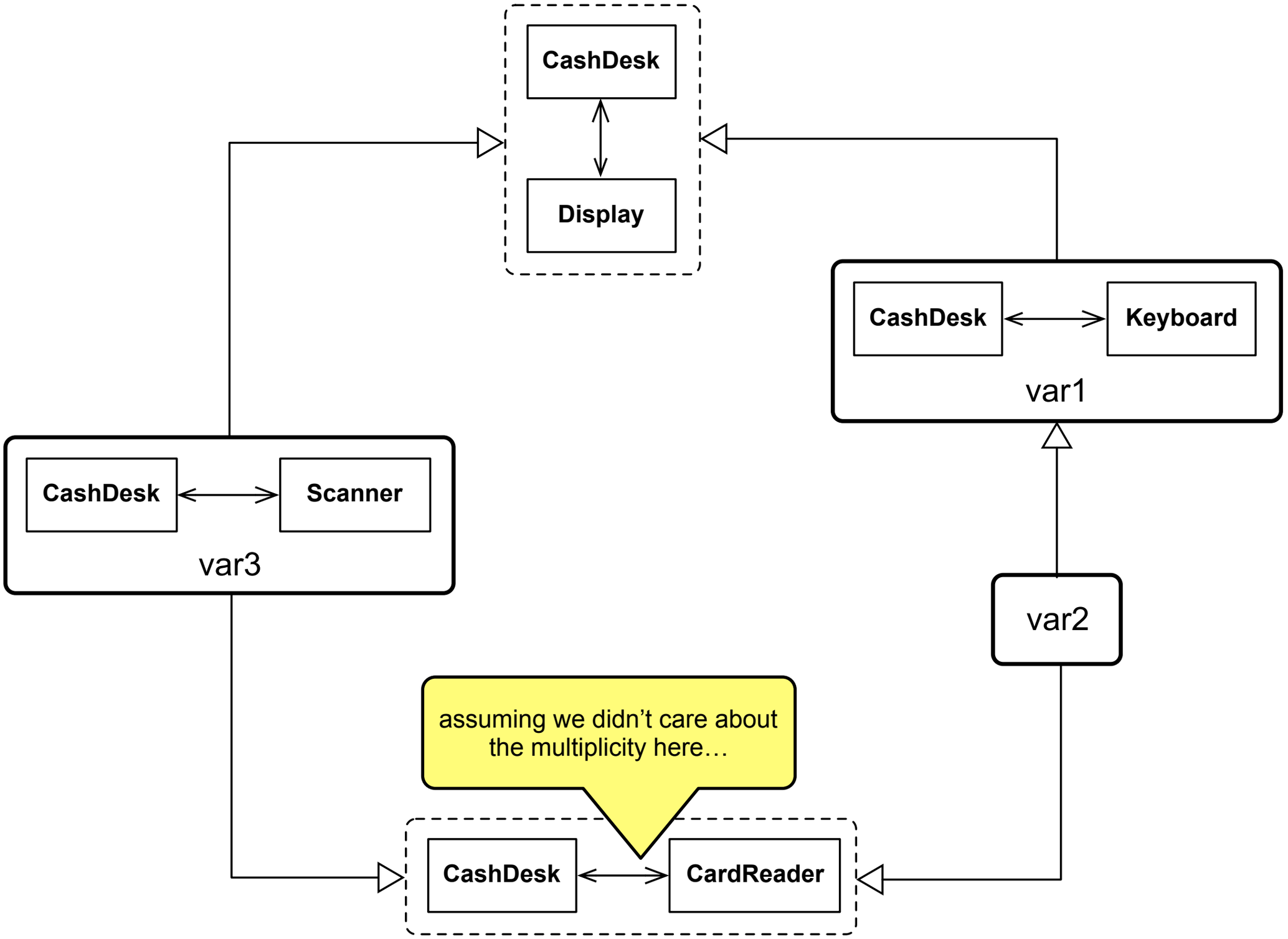
Step 3 (Merge): Merge all elements with the same name together

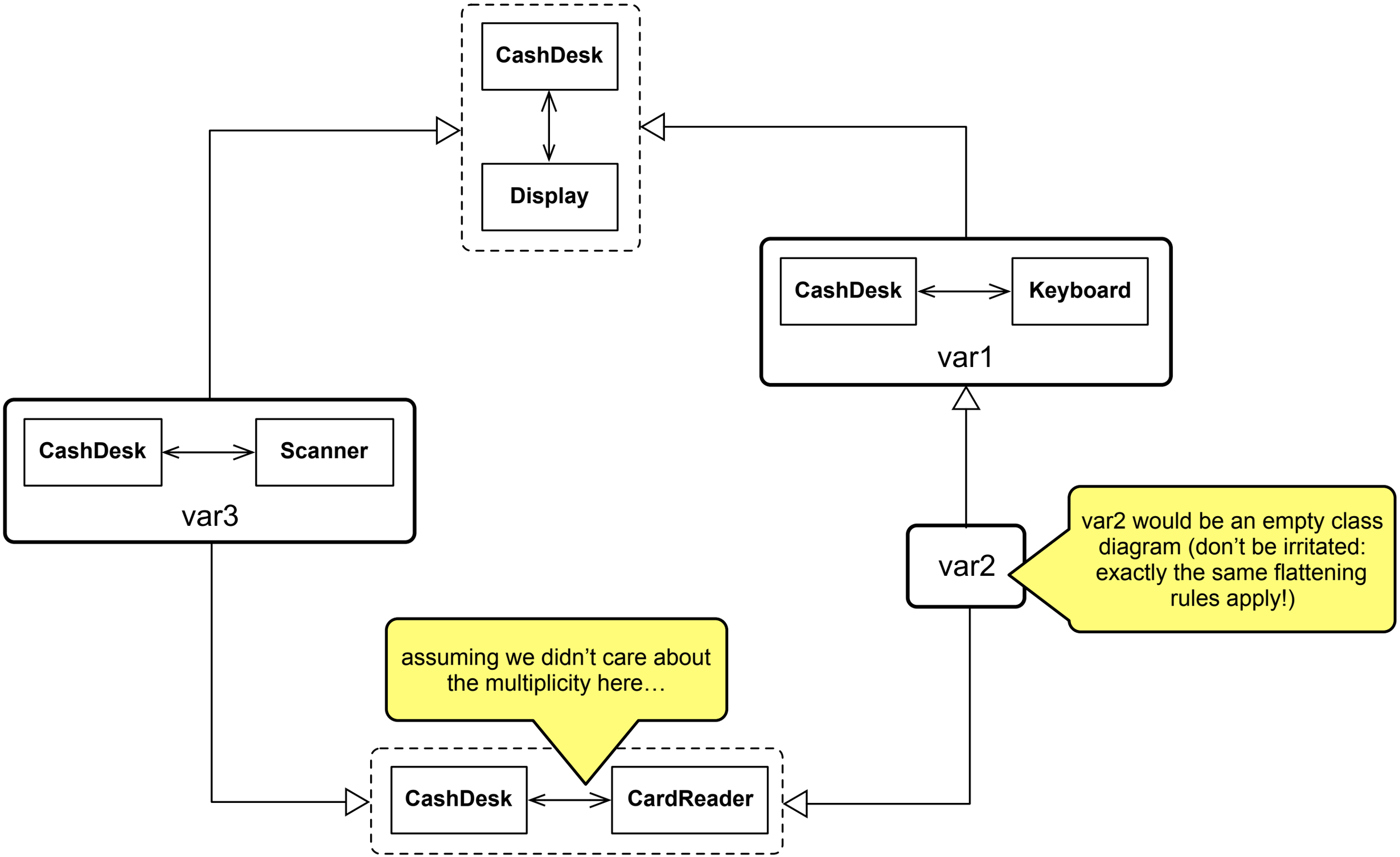


Step 4 (Resolve): Resolve any conflicts (subtypes win, multiplicities lower down in the refinement hierarchy win)




CashDesk
Display
CashDesk
Keyboard
var1
CashDesk
Scanner
var3
var2
CashDesk
CardReader


CashDesk
Display
CashDesk
Keyboard
var1
CashDesk
Scanner
var3
var2
assuming we didn’t care about
the multiplicity here…
CashDesk
CardReader


CashDesk
Display
CashDesk
Keyboard
var1
CashDesk
Scanner
var3
var2
var2 would be an empty class diagram (don’t be irritated: exactly the same flattening rules apply!)
assuming we didn’t care about the multiplicity here…
CashDesk
CardReader