Chapter 5

Reuse Contracts for the UML

Until now, reuse contracts were handled as a fairly abstract way of expressing the structure of a software system. We started by representing reuse contracts on such an abstract level because we believe that the reuse contract approach is scalable and can be applied to different composition mechanisms. While we get back to this scalability issue in the conclusions, as a proof of usability in this chapter we apply reuse contracts to the field of object-oriented class libraries and frameworks. This demonstrates more clearly the concepts and allows for concrete experiments demonstrating how reuse contracts can be applied and how they can help in evolution and composition. Such experiments are discussed in chapters 6 and 7 .

Since we do not see reuse contracts as an entirely new methodology, but rather as an enhancement to existing methodologies, we do not start from scratch but integrate reuse contracts in the Unified Modelling Language (UML)[BRJ97]. The question is then where reuse contracts fit in. The UML presents seven different kinds of diagrams to describe different aspects of an object-oriented software system. Two of those seem immediately relevant: static structure diagrams (i.e., what is usually called class diagrams) and collaboration diagrams. The first concentrates on how different classes are combined structurally, while the second describes the collaboration between different acquainted objects. As we feel reuse contracts can be useful on both levels, we handle both in this chapter. We call these particular versions of reuse contracts *multi-class reuse contracts* and *collaboration reuse contracts*.

While these two diagrams are the most obvious, we by no means feel they are the only ones that should be considered. Sequence diagrams are closely related to collaboration diagrams and could be an obvious next step. [MS96] describes how the reuse contract principles can be applied to OMT's state diagrams [RBP^{+}91], which are very similar to UML's. In a later stage we would like to study applying reuse contracts to notions as diverse as use case diagrams (following Ivar Jacobson's use cases [JCJO92]), or implementation diagrams that describe the general architecture of a system at a higher level, but this is future work[1]. Let us now focus on static structure and collaboration diagrams.

5.1 Basic Static Structure Diagrams

Static structure diagrams in UML are similar to class diagrams in other object-oriented methodologies. Classes are depicted by rectangles which are divided in three parts: one for the name of the class, one for the attributes and one for the operations. Associations between classes are depicted by lines and inheritance by a hollow triangle. Multiplicity constraints and an aggregation sign can be added along the bindings. We do not discuss the entire notation here, but instead introduce new modelling elements as we need them. We start by deciding how reuse contracts can be mapped on a subset of UML static structure diagrams and then proceed by discussing how additional features of these diagrams can be added to this basic mapping. We do not integrate all model elements of UML, but rather focus on features that are crucial to documenting reusable components.

It is clear that participants should map to classes, operations to methods, acquaintance relationships to associations and acquaintance names to role names[2]. Only one item of our basic reuse contract notation is left, operation invocation. In object-oriented systems this naturally maps to message passing. In the UML, message passing is not modelled on the level of classes in static structure diagrams, but only between objects in sequence and collaboration diagrams. We feel however that it is useful to be able to model general message passing sequences between classes. This need becomes apparent, for example, in books describing design patterns, where often small tags with code fragments are attached to associations [GHJV94]. Its usefulness also becomes clear in an experiment we describe in the following chapter. We therefore extend the notation of static structure diagrams with message passing. We denote it the same way as in original reuse contracts, along UML associations.

As an example of multi-class reuse contracts we use the Model-View- Controller framework (MVC) from VisualWorks throughout this chapter, as it is well known and is often named as the first mini-framework[3]. Figure 5.1 illustrates the MVC at the highest level of abstraction, i.e. the classes `Model`, `View` and `Controller`. At that level the way they are connected is specified, plus some basic behaviour. First, it is specified that when a `Model` receives the method `changed:with:` it sends the message `update:with:from:` to each of its dependents[4]. Second, the method

[1]See chapter 8 for an in-depth discussion of future work.

[2]Throughout this chapter we use the UML vocabulary to denote items from object-oriented languages, except for the use of the term method instead of operation, to stress the difference with the basic definitions.

[3]Note that we use the Smalltalk syntax in these examples, i.e. names of methods with colons to denote their arguments. We ignore the arguments here, their inclusion is an extension to the basic model. We just distinguish different method names by means of this notation.

[4]Both for the method `changed:with:` and `update:with:from:` different versions with fewer

`activate` on `Controller` invokes `hasControl` on itself. This is denoted between braces in the client interface, as we have denoted intra-participant behaviour before. We get back to self sends later. While this contract is quite rudimentary and does not specify much behaviour it already gives us a general view. We know, for instance, that every `View` has a `Controller` and vice versa[5].

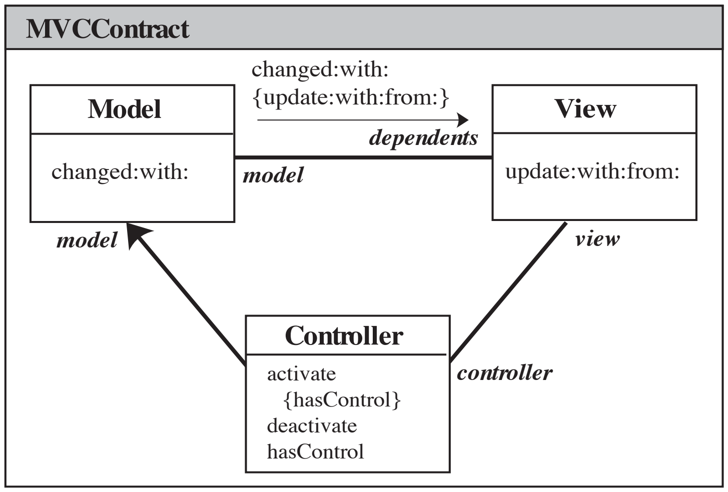

Figure 5.1: Model-View-Controller

Now that we have decided which subset of static structure diagrams reuse contracts can be mapped to, we can reconsider the definition of reuse contracts. In order to obtain multi-class reuse contracts, we start by taking the original definition and replace all occurrences of the word 'participant' by 'class' and all occurrences of the word 'operation' by 'method'.

Definition 5.1 (Multi-class Reuse Contract) A **multi-class reuse contract** is a set of class descriptions, each with

1. a unique name;
2. an acquaintance clause;
3. an interface.

Definition 5.2 (Acquaintance Clause) An **acquaintance clause** is a set of acquaintance relationships $a.c$, associating an acquaintance name a with a class name c.

arguments exist, but these are propagated to these two versions, therefore we only discuss these basic two here.

[5]For `Views` that have no particular interactions an instance of the class `NoController` is attached to it.

We again further define the form of interfaces.

Definition 5.3 (Interface) An **interface** consists of a set of methods each consisting of

1. a method name that is unique within this interface,

2. a specialisation clause.

Finally, we again define specialisation clauses.

Definition 5.4 (Specialisation Clause) A **specialisation clause** is a set of method invocations $a.m$, associating an acquaintance name a with a method name m.

The MVCContract is then denoted as follows:

```
MVCContract =
{ ( Model,
   { dependents.View },
   { (changed:with:, {dependents.update:with:from:}) }
  ),

  ( View,
   { model.Model, controller.Controller },
   { (update:with:from:, {}) }
  ),

  ( Controller,
   { model.Model, view.View },
   { (activate, {self.hasControl}),
     (deactivate, {}),
     (hasControl, {}) }
  )
}
```

The reuse contract contains three class descriptions, named `Model`, `View` and `Controller`. The first has only one element in its acquaintance clause, the other two acquaintance clauses have two elements. The first two class descriptions only have one element in their interface, the last has three elements in it: `activate`, `deactivate` and `hasControl`. While `deactivate` and `hasControl` have an empty specialisation clause, `activate` invokes `hasControl` on self. In the same way as we adapted the basic definitions, we can redefine well-formedness, as well as the operators as described in chapter 2.

5.2 Integrating the Operators

As an example of the result of applying an operator consider figure 5.2. While figure 5.1 describes general MVC behaviour, different refinements and extensions of the contract can be used to describe different specialisations. Different kinds of views with their associated kind of controller are already present in the VisualWorks class library. Figure 5.2 describes the example of `BasicButtonView` and `BasicButtonController`.

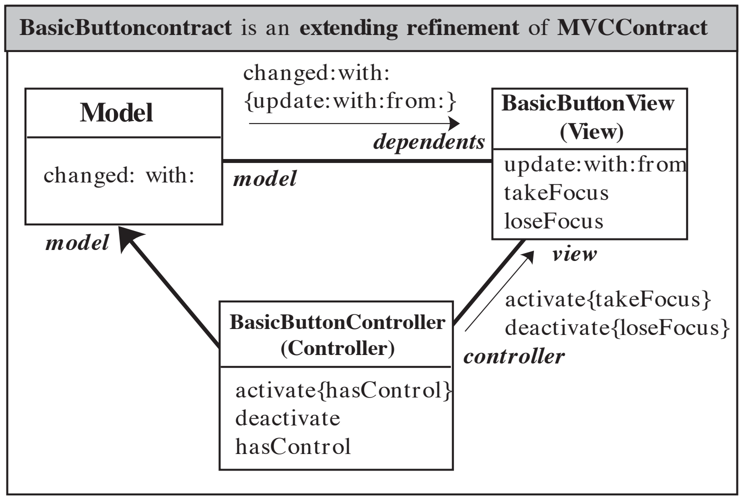

Figure 5.2: MVC for BasicButtonView and BasicButtonController

`BasicButtonController` is a special kind of `Controller`, that sends the message `takeFocus` to its view when it gets activated and the message `loseFocus` when it gets deactivated. As the methods `activate` and `deactivate` already exist on `Controller` and `activate`'s specialisation clause is preserved, this new contract is a participant refinement of the original one. Because the methods `takeFocus` and `loseFocus` are added to `View` it is also a participant extension. In fact, as the only methods that are added are called through the refinement we can decide that the `BasicButtonContract` is an extending refinement of the `MVCContract`.

Now how can the relationship between two models as expressed through the reuse operators be integrated in UML? UML recognises the need to describe a system at different levels of granularity and therefore introduces a refinement relationship. '*The refinement relationship represents the fuller specification of something that has been already specified at a certain level of detail. It is a commitment to certain choices consistent with the more general specification, but not required with it. It is a relationship between two descriptions of the same thing at different levels of abstraction*' [BRJ97]. So refinement in UML shows a relationship between two different views of something. This is exactly what our reuse operators do, only at a finer level of detail!

According to [BRJ97] refinement can include, amongst others, the following kinds of relationships:

- *Realisation:* relation between a type and a class that realises it;
- *Design trace:* relation between an analysis class and a design class;
- *Levelling of detail:* relation between a high-level construct at a coarse granularity and a lower level construct at a finer granularity, such as a collaboration at two levels of detail;
- *Implementation:* relation between a construct and its implementation at a lower virtual layer, such as the implementation of a type as a collaboration of lower-level objects.
- *Optimisation:* relation between a straightforward implementation of a construct and a more efficient but more obscure implementation that accomplishes the same effect.

In the MVC example and the Gateway and Set examples of chapter 1 the reuse operators were used to model levelling of detail and optimisation, but we will see further on that reuse operators can also be used to model different kinds of relationships.

So how do we integrate reuse operators into the notation of UML static structure diagrams? A refinement link can be depicted by a dashed generalisation symbol, i.e. a dashed line with a hollow triangular arrowhead at the end connected to the most general element. It can be either visible within one model, or an invisible hyperlink between models supported by a tool. Furthermore, a refinement relationship in UML can have a specification of how the more detailed version maps into the more abstract version. This can be denoted by a stereotype[6] and a note that is attached to the link connecting both constructs. Instead of attaching just any note to this link, we attach a stereotype describing the kind of reuse operator and possibly a note describing the reuse modifier. Figure 5.3 shows how `MVCContract` and `BasicButtonContract` can be depicted this way. Note that we again use the notation where '+' and '-' signs denote what is added and omitted.

While this notation is very useful, we do not always repeat entire contracts to model changes to particular classes. In object-oriented software engineering variability is most often achieved through *inheritance.* In UML this is called *generalisation*

[6]In the UML stereotypes represent a built-in extensibility mechanism. The general representation of a stereotype is to place the name of the stereotype between matched guillemets, as in ≪foo≫. The stereotype string can be placed above or in front of the name of the model element being described. Stereotypes can be used, for example, to group methods of one class into subgroups or to characterise different kinds of classes, or kinds of relationships between model elements..

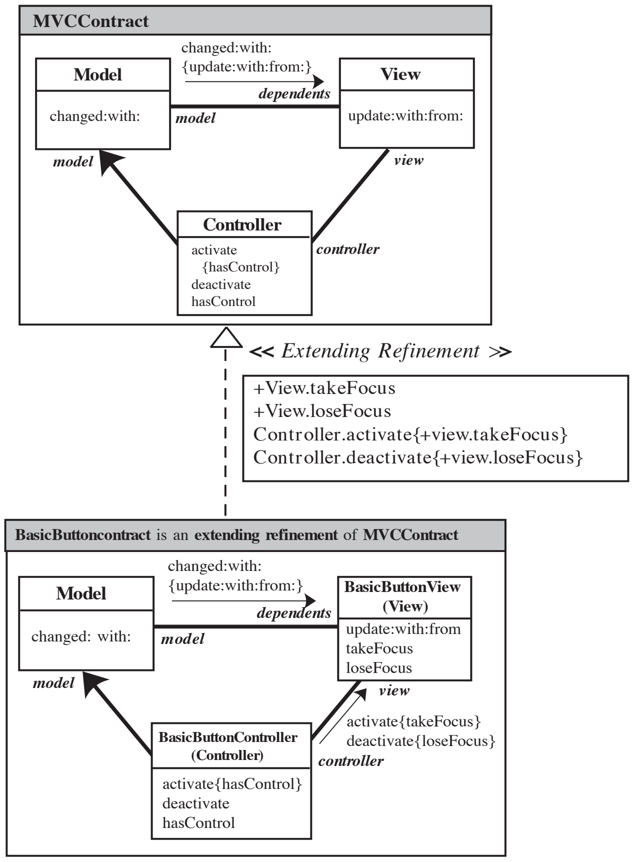

Figure 5.3: Contract Refinement

and it is depicted by a hollow triangle. We also adopt the UML notation for generalisation and add the operator name next to the hollow triangle, as depicted in figure 5.4. The acquaintances to which message sends are added are denoted by their names.

5.3 Impact of Inheritance on the Conflicts

The introduction of inheritance, and more particularly of late binding, also sheds a new light on some of the conflicts. Take, for example, method capture[7]. This conflict occurs when a method is invoked after one modification, while another modification adapted this method unaware of the extra invocation. Until now, we considered

[7]In this and the following chapters we use the terms method capture, inconsistent methods, etc. instead of operation capture and inconsistent operations.

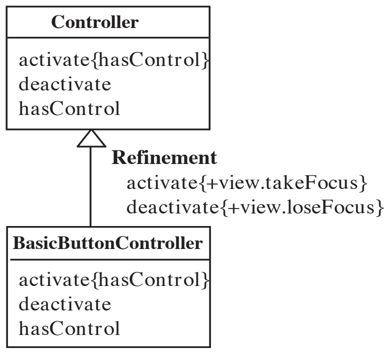

Figure 5.4: Refinement of Controller

method capture to occur with calls between different participants. Now consider the situation where a subclass overrides a certain method and afterwards the superclass is adapted to add an invocation to this method. Late binding will cause method capture to occur.

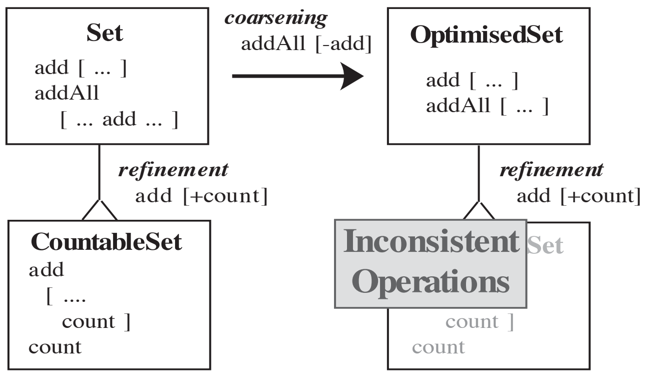

Figure 5.5: Inconsistent Methods on Set

An example of inconsistent methods through inheritance was already given in chapter 1, concerning the Set and CountableSet. The example is repeated in figure 5.5. The inheritor performs a refinement by overriding `add` to invoke `count`, assuming that this will also influence its inherited method `addAll`. The coarsening performed to achieve an optimisation of `Set` broke this assumption, with as result that in the inheritor the behaviour of `add` and `addAll` is no longer consistent. Here we just considered one class and its inheritor, but the depth of inheritance hierarchies and the layered definition of methods makes similar problems hard to detect. This example shows how any change to a method in a superclass that adds or removes method calls might lead to method capture, respectively inconsistent methods.

In general, the rules for conflict detection as given in chapter 3 can be used to detect problems with *parent class exchange*, i.e., possible problems that can occur in subclasses when changes are made to a superclass. When all steps going from one class to its subclasses are documented by means of reuse operators, we can express the changes made to the superclass with reuse operators as well and apply the rules. In the above example, the subclass performs a refinement and the parent class exchange expresses a participant coarsening, which leads to the inconsistent methods conflict. Note that some of the operators can never be expressed through inheritance. Participant cancellation, for example, is not allowed in statically typed languages as C++ and Java. In Smalltalk it can be simulated by the convention of invoking the method `shouldNotImplement`.

While the conflict detection rules can thus be applied to the case of parent class exchange, some particularities of object-oriented languages and software engineering deserve special attention.

5.4 Integrating Late Binding

While the above example already demonstrates to some extent how the reuse operators can assist in clarifying the structure of a framework, it also shows some of the shortcomings. Actually, the original specialisation clause of both `activate` and `deactivate` is preserved on `BasicButtonController` because a super send is performed. In the original definition of reuse contracts nothing was said about self and super sends. Moreover, self sends were added above in the graphical notation, but not in the formal definition. We will therefore now elaborate on the basic definition and enhance it to model self and super sends.

5.4.1 Self Sends

First, until now we have not really incorporated the use of self sends. We therefore add a special section to specialisation clauses to express self sends.

Definition 5.5 (Specialisation Clause Revisited) A **specialisation clause** is a set of method invocations $a.m$, where a is an acquaintance name or the keyword $self$ and m is a method name.

As mentioned before, we graphically depict self sends by adding the methods that are invoked between braces to method signatures in the interface of a class, as depicted in figure 5.6. This figure shows how on class `Model` the different versions of the method `changed` propagate behaviour to each other.

We already mentioned in chapter 2 that it is possible for an acquaintance name to refer to the participant it is defined on. This kind of binding can be represented by a loop. We then argued that operations along such a loop represent intra-participant

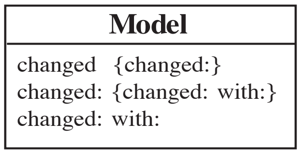

Figure 5.6: The Representation of Self Sends

behaviour. One might think that this could be used to represent self sends, but an association between a class and itself could be used for all kinds of acquaintance relationships between two instances of the same class. These two instances need not necessarily be the same. Take for example the class `VisualPart` (a superclass of `View`), which is used to group different parts of a visual representation together. A `VisualPart` consists of other `VisualParts` and when an instance of `VisualPart` receives the method `update:with:from:`, it passes it on to the `VisualParts` it contains. This is depicted in figure 5.7.

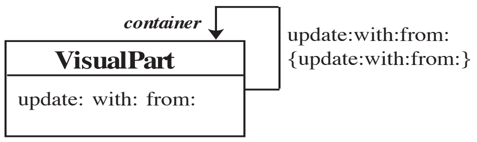

Figure 5.7: Message Sends between Instances of the Same Class

5.4.2 Super Sends: Specialisation

Motivation

We also want to include the possibility to perform super sends. We do not want to extend the definition of a specialisation clause to incorporate this notion explicitly, because each participant in a reuse contract should be self-contained, without needing knowledge of its superclasses to judge well-formedness or to detect conflicts. Therefore, the specialisation clause of a method performing a super send will contain all method names of the specialisation clause of the method it overrides, plus the names of any new methods it may call.

Until now a specialisation clause of a refinement modifier needed to repeat the entire specialisation clause of the operation it was adapting, in order to ensure that the design was respected. When using a super send this is no longer necessary, as a super send always respects the method that is adapted by invoking it. We therefore introduce an extra refinement operator that differs only slightly from participant refinement: specialisation. The only difference is that while in participant refinement the modifier's specialisation clauses needed to be supersets of the original ones, for specialisation the modifier's specialisation clauses only include the new invocations. The example of `BasicButtonView` and `BasicButtonController` already demonstrated the usefulness of this operator.

Definition and Properties

Definition 5.6 (Specialisation Modifier) A **specialisation modifier** is a reuse modifier with modifier tag "specialisation" and a modifier description containing couples (c, int) each consisting of a class name c and an interface int.

The applicability of the modifier is stated in the following definition. Note that the last clause again repeats part of the well-formedness definition.

Definition 5.7 (Specialisable) A reuse contract R is **specialisable** by a specialisation modifier M_{sp} if for each couple (c, int) in M_{sp}:

1. c is a class name in R ;
2. for each method name m in int: m appears in class c in R and m's specialisation clause is disjoint from the specialisation clause of m in c in R ;
3. for each method invocation $a.m$ in a specialisation clause in int :
   (a) a is an acquaintance name in the acquaintance clause of c ;
   (b) m is the name of a method in the interface of the class a refers to.

Note that while the modifiers of specialisation contain only the newly added invocations, in the resulting reuse contract we want to sum up the entire specialisation clause. Our motivation for this is to obtain well-formed reuse contracts. Specialisation of contracts can then be defined as follows.

Definition 5.8 (Specialisation) If a reuse contract R is specialisable by a modifier M_{sp}, then the reuse contract R_{sp} is the **specialisation** of R by M_{sp}, where:

1. R_{sp} contains all class descriptions of R that are not mentioned in M_{sp};
2. for each (c, int) in M_{sp}: R_{sp} contains a class description
   (a) with name c and the same acquaintance clause as c in R ;
   (b) that contains all methods of c in R not mentioned in int ;
   (c) that contains all methods of int with as specialisation clause the union of the specialisation clause of this method on R and the specialisation clause of this method in int .

The following property can be proven about these definitions.

Property 5.1 *A specialisation of a well-formed reuse contract is well-formed.*

Proof

The well-formedness definition imposes 3 constraints.

1. WF1 is not influenced by specialisation;
2. WF2 is preserved by the fact that the specialisation clauses on R_{mr} partially refer to class names that were already present and further by the third clause of the refinability definition;
3. WF3 is preserved by the fact that the specialisation clauses on R_{mr} partially refer to method names that were already present and further by the fourth clause of the refinability definition.

Example

We will not give abstract illustrations of the operators in this chapter. Since they are all fairly straightforward, we immediately give an example. An example of specialisation was already given in section 5.1 with `BasicButtonView` and `BasicButtonController`, as depicted in figure 5.8.

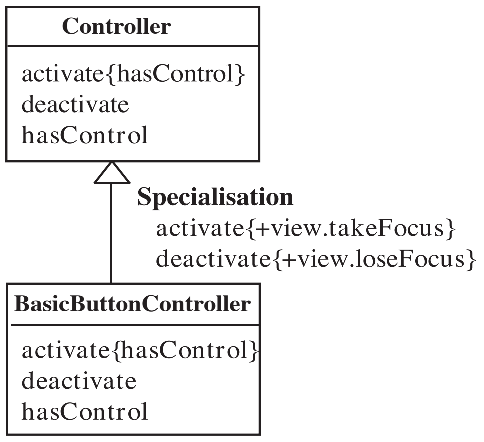

Figure 5.8: An Example of Specialisation

Note that in this definition we only consider super sends of the same method. Performing a super send from one method to another is considered bad design, therefore we do not include the possibility to model this through reuse contracts.

Impact on the conflicts

As specialisation is very closely related to participant refinement it is logical that the conflicts that can occur during specialisation closely resemble those that can occur during participant refinement. Table 5.1 compares the conflicts that can be caused by the two operators.

	part. refinement	specialisation
part. extension	-	-
cont. extension	-	-
part. cancellation	dang. method reference	dang. method reference
cont. cancellation	dang. participant reference	dang. participant reference
part. refinement	*method invocation*, inconsistent methods, regular method capture	unanticipated recursion, regular method capture
cont. refinement	-	-
part. coarsening	*method invocation*, inconsistent methods, regular method capture	unanticipated recursion, regular method capture
cont. coarsening	dang. acquaintance reference	dang. acquaintance reference

Table 5.1: Conflicts with Specialisation

The only difference is that while participant refinement can cause method invocation conflicts in combination with another participant refinement or coarsening, this conflict does not occur with specialisation. Remember that method invocation conflicts occur when both modifications add or remove method invocations from the specialisation clause of the same method. This can cause inconsistencies. This does not occur with specialisation, because the super call in the refined method automatically incorporates all changes that could possibly be made to the original reuse contract through a participant refinement or coarsening of the methods it adapts.

5.5 Abstract Classes and Methods

Until now we have not made a distinction between concrete and abstract methods. The use of abstract classes and methods is, however, a prominent object-oriented reuse technique. An abstract class is a class that contains a number of abstract methods, i.e. methods without an implementation. Template methods invoke abstract methods or other template methods in their implementation. Finally, there are concrete methods, which have a full implementation, that does not invoke abstract methods. Template methods describe the core behaviour of the class. Subclasses only need to override the abstract methods to give them an implementation, or some concrete methods to adapt their behaviour, but they inherit the core behaviour by means of the template methods. The overriding of abstract methods with concrete versions is thus a very important operation. We therefore extend our model with the notion of abstract and concrete methods and with an extra operator, *participant concretisation* and its inverse, *participant abstraction*.

UML provides the possibility to associate certain properties with operations. One of the possibilities is to distinguish abstract methods from concrete ones by representing the abstract methods in italics. As an example, consider `ValueModel`, one of the subclasses of `Model`, which represents all kinds of models holding a certain value. It has two abstract methods, `setValue:` and `value` to mutate and access this value, and a template method `value:` that relies on `setValue:` to mutate and sends the message `changed:` to itself to notify its dependents that its value has changed. The methods `setValue:` and `value` need to be overridden in subclasses of `ValueModel` to provide an implementation suited to the particular kind of value. This abstract class is depicted according to the UML notation for abstract classes as shown in figure 5.9.

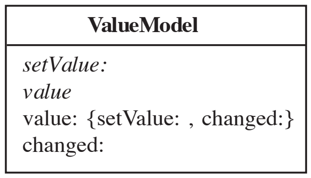

Figure 5.9: An Abstract Class

5.5.1 Extension of the Model

To extend reuse contracts with information about abstract and concrete methods, we need to attach an extra annotation, `abstract` or `concrete`, to each method in the interfaces. We therefore extend our definition of interfaces (see page 130) as follows.

Definition 5.9 (Interface Revisited) An **interface** consists of a set of methods each consisting of

1. a method name that is unique within this interface,
2. an annotation *abstract* or *concrete*,
3. a specialisation clause.

We do not need to adapt the definitions of reuse contract and specialisation clause. Reuse contracts still contain class descriptions consisting of a name, an acquaintance clause and an interface. Only now the interfaces are slightly changed. Specialisation clauses do not need to repeat the annotation of the methods they contain. This information can be found in the concerning interfaces.

Note that we do allow abstract methods to have a specialisation clause, although they do not have an implementation. This might seem awkward, but we want to keep this possibility open, because, for example, during the design one might want to express that a certain method should definitely call some other method, without already giving an implementation. In a lot of examples, however, the specialisation clauses of abstract methods will simply be empty.

The definition of well-formedness does not need to be adapted either. Since nothing changes the uniqueness of the names of methods, i.e. it is not allowed to have a concrete and an abstract method with the same name in one interface, and since abstract methods are allowed to have specialisation clauses the well-formedness definition does not require any extra constraints.

The definitions of the operators can also be preserved to a very large extent. The only changes that need to be made are the addition of considerations concerning the annotation. For example, when adding a new method through a participant extension this new method should also have an annotation. In participant refinement, when a method is refined, the definition should not only state that all methods in the refinement modifier should have the same name as in the base contract, but also that they should have the same abstractness annotation. This will be necessary to detect certain conflicts, just as it was necessary to repeat other redundant information in modifiers. Because the changes to the definitions are straightforward, we will skip the adapted definitions here.

5.5.2 A New Operator: Participant Concretisation

Motivation

Now that we have extended the model with the extra information, we will introduce an extra operator: *participant concretisation* and its inverse, *participant abstraction*. As mentioned before, an important action in customising reusable classes is the overriding of abstract methods with concrete ones. This is exactly what participant concretisation represents. Intuitively, R_{pc} is a participant concretisation of R if any number of abstract methods from one or more class descriptions in R is 'overridden' by concrete methods in R_{pc}.

Definition and Properties

Definition 5.10 (Participant Concretisation Modifier) A **participant concretisation modifier** is a reuse modifier with modifier tag "participant concretisation" and a modifier description containing couples (c, int) each consisting of a class name c and an interface int , in which all methods have the annotation concrete.

The applicability of the modifier is stated in the following definition.

Definition 5.11 (Participant Concretisable) A reuse contract R is **participant concretisable** by a participant concretisation modifier M_{pc} for each pair (c, int):

1. c is a class name in R ;
2. for each method m in int :
   (a) m has the same name as an abstract method in c in R ;
   (b) m has the same specialisation clause as the corresponding method in c in R .

Note that again we include more information in the reuse modifier than would seem necessary at first. We repeat information on specialisation clauses to be able to detect certain interface conflicts (see below) and we include information on the annotations to be complete.

Definition 5.12 (Participant Concretisation) If a reuse contract R is participant concretisable by a modifier M_{pc}, then the reuse contract R_{pc} is the **participant concretisation** of R by M_{pc}, where:

1. R_{pc} contains all class descriptions of R that are not mentioned in M_{pc};
2. for each pair (c, int) in M_{pc}: R_{pc} contains a class description
   (a) with name c and the same acquaintance clause as c in R ;
   (b) that contains all methods of c in R not mentioned in int ;
   (c) that contains all methods of int .

Following property can again be proven about these definitions.

Property 5.2 *A participant concretisation of a well-formed reuse contract is well-formed.*

Proof
As we discussed above, the introduction of the annotations `abstract` and `concrete` does not influence well-formedness. As this annotation is the only item that is changed through participant concretisation, well-formedness will not be affected by it.

Example

As an example of a concretisation[8] consider the subclass `ValueHolder` of `Value-Model`. In `ValueHolders` the value is simply stored in an instance variable. The class concretises the methods `value` and `setValue:` accordingly. This is depicted in figure 5.10.

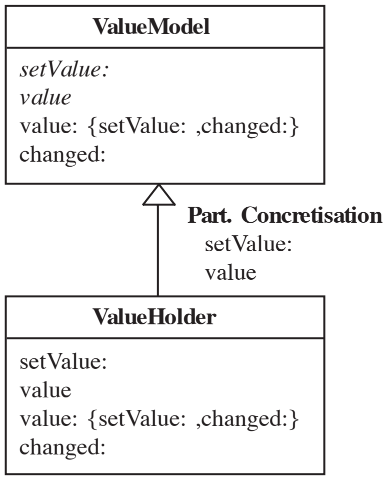

Figure 5.10: Participant Concretisation

Short-Hand Notations

We make a distinction between complete and partial concretisations. A *complete concretisation* concretises all abstract methods in a contract, thereby resulting in the representation of a set of instantiatable classes. A *partial concretisation* leaves some abstract methods, requiring subsequent concretisations.

Notation 5.1 A participant concretisation modifier M_{cc} represents a **complete participant concretisation of** R if
for each class c in R that has at least one abstract method: a pair (c, int) exists in M_{cc} and all abstract methods of c in R are mentioned in int .

Notation 5.2 A participant concretisation modifier M_{pc} represents a **partial participant concretisation of** R if
a class c in R exists that has at least one abstract method and either c is not mentioned in M_{pc} or a pair (c, int) exists in M_{cc} and not all abstract methods of c on R are mentioned in int .

[8]We will often use the names concretisation and abstraction without the prefix participant when no confusion is possible.

Furthermore, we need the classical short-hand notation as before for the rules on change propagation.

Notation 5.3 A participant concretisation modifier M_{pc} represents a **participant concretisation of** m **on** c if
a class c and a method m exist such that (c, int) is an element of M_{pc} and m is an element of int .

Other

The opposite of participant concretisation is called participant abstraction. We will not give a complete definition of this operator as its definition would be exactly the same of that of participant concretisation, only with occurrences of the word `concrete` replaced with `abstract` and vice versa.

Definition 5.13 (Participant Abstraction) R_{pa} is a **participant abstraction** of R iff R is a participant concretisation of R_{pa}

Impact on the conflicts

The introduction of concretisation and abstraction introduces new possible conflicts and also sheds a different light on certain already existing conflicts. We start with the new conflicts.

Annotation Conflicts

As a new element is added to the interface, new interface conflicts are possible. Annotation conflicts occur when two modifications both make the same abstract method concrete, or vice versa. As the annotations of methods can only be altered through concretisation and abstraction, annotation conflicts can only occur when both modifications perform a concretisation, or when both modifications perform an abstraction. This is a new kind of interface conflict, as it concerns a clash between the new piece of information added to the interface.

Rule 5.1 (Annotation Conflict) An **annotation conflict** occurs when a method name m and a class name c exist such that both M_1 and M_2 represent participant concretisations or participant abstractions of m on p .

Note that, as with method invocation conflicts, both concretisation and its inverse abstraction are given. Again, because of the applicability definition the only possible situation are where both modifiers perform the same operator.

The solution to this conflict is to either remove one of the modifications, thereby accepting the concretisation (and according implementation) offered by the other one. When this is not desirable, M_2 can be turned into a refinement or coarsening thereby adapting the concretisation (and according implementation) offered by M_1 to this user's needs. When refining, the design offered by M_1 is fully accepted and extended; when coarsening, the design offered by M_1 is more or less breached.

Mixed Method Interface Conflicts

A second new kind of interface conflicts are mixed method interface conflicts. This problem occurs when a method is on the one hand concretised and on the other hand refined or coarsened. After the refinement or coarsening, the concretisation is no longer valid: in the case of refinement because the specialisation clause was extended, in the case of coarsening because the specialisation clause was narrowed. Note that it is in order to detect this conflict that we added information on the specialisation clauses in the concretisation modifier. Conflicts also occur in the inverse order. In that case the refinement or coarsening is no longer valid after the concretisation, because the refinement or coarsening is defined for an abstract method. There is an essential difference between refining an abstract and a concrete method, because in the first case no implementation will be associated with the modifier, while in the second case an implementation will be there. Mixed method interface conflicts can thus occur when one modification is a concretisation and the other a refinement or coarsening.

Rule 5.2 (Mixed Method Interface Conflict) A **mixed method interface conflict** occurs when a method name m and a class name c exist such that M_1 represents a participant concretisation of m on c and M_2 represents a participant refinement or a participant coarsening of m on c .

Note that consecutively adding new information to the interfaces will lead to different kinds of mixed conflicts.

Incomplete Implementation

It is possible for one modifier to perform a complete concretisation, therefore leading to the believe that the result is an instantiatable class, while another modification adds new abstract methods. We call this last new conflict the *incomplete implementation conflict.* Since this occurs when abstract methods are added, this can only occur after a combination of a participant extension or abstraction and a participant concretisation.

To be able to detect this conflict that we added the definitions on complete and partial concretisations above. Using these definitions we can state the following rule.

Rule 5.3 (Incomplete Implementation Conflict) An **incomplete implementation conflict** occurs when

- M_1 represents a complete participant concretisation of R;
- M_2 represents a participant abstraction of R or a participant extension by a method m , where m is abstract;

Note that this is the second conflict, after unanticipated recursion, that cannot be detected by solely looking at the modifiers. We need to take the base reuse contract into consideration to know whether the concretisation is complete or not. This could be avoided by introducing two different operators *partial concretisation* and *complete concretisation* and thus let the user better document his assumptions. This brings us back to the trade-off between flexibility and the possibility to detect conflicts we discussed in chapter 3. Here, we chose not to make operators too fine-grained.

Method Capture and Inconsistent Methods

Earlier on, method capture and inconsistent methods were described to occur when one modification was a participant refinement, respectively a participant coarsening and the other either one of these. The idea behind it was the addition or removal of a method call by the first modifier of a method that was adapted in some way by the second modifier. Since concretisation and abstraction also adapt a method in some way these operators can also lead to method capture, respectively inconsistent methods.

Interface Inconsistency Conflicts

The same line of thought can be followed concerning interface inconsistency conflicts. These conflicts all occur when in one modifier a reference is made to an item that is removed by the other modifier. When a method is concretised or abstracted, it is possible that another modifier removes either a method it invokes, the acquaintance relationship along which it invokes it or even the very class it is defined on (and thus also the method itself). This leads to respectively dangling method, dangling acquaintance and dangling participant conflicts.

Summary

Table 5.2 gives an overview of all conflicts that can occur with concretisation and abstraction.

	participant concretisation	participant abstraction
part. extension	incomplete implementation	-
cont. extension	-	-
part. cancellation	dang. method reference	dang. method reference
cont. cancellation	dang. participant reference	dang. participant reference
part. refinement	mixed method interface, regular method capture	mixed method interface, regular method capture
cont. refinement	-	-
part. coarsening	mixed method interface, inconsistent methods	mixed method interface, inconsistent methods
cont. coarsening	dang. acquaintance reference	dang. acquaintance reference
part. concretisation	annotation	incomplete implementation
part. abstraction	incomplete implementation	annotation

Table 5.2: Conflicts with Participant Concretisation and Abstraction

5.5.3 A Combined Operator: Layered Concretisation

Motivation

As we stated before, abstract methods are allowed to have specialisation clauses, but in a lot of cases these will simply be empty. Concretising a method will therefore very often go together with additions to the specialisation clause. This will also often be accompanied by the introduction of auxiliary methods. We will therefore introduce a new combined operator that is a combination of concretisation and extending refinement: layered concretisation.

Definition and Properties

As in the previous chapter, all we have to do in order to introduce a new combined operator, is to define the structure of the combined modifier.

Definition 5.14 (Layered Concretisation Modifier) A **layered concretisation modifier** is a combined reuse modifier with modifier tag "layered concretisation" and a modifier description containing a sequence (M_c, M_{er}), where:

1. M_c is a participant concretisation modifier and M_{er} is a participant extending refinement modifier;

2. the methods refined by M_{er} are a subset of the methods concretised by M_c;

Note that we speak of a subset, but this need not be strict. In other words, we are also allowing to refine every method that is concretised.

Definition 5.15 (Layered Concretisation) R_{lc} is a **layered concretisation** of R by M if

1. M is a layered concretisation modifier;
2. M is applicable to R ;
3. R_{lc} is the result of applying M to R

Example

We used the class `ValueHolder` as an example of a concretisation: values were simply stored in an instance variable and no extra behaviour was necessary. There are however other subclasses of `ValueModel` that have a more complicated behaviour. Take, for example, the class `ProtocolAdapter`. This class also concretises the methods `value` and `setValue:`, but it also adds two extra method invocations to `setValue:`'s specialisation clause. It are invocations of two methods newly added to `ProtocolAdapter`, namely `target` and `setValueUsingTarget:to:`[9]. Figure 5.11 depicts this layered concretisation.

Impact on Conflicts

The set of conflicts that can occur after the application of layered concretisation is the union of those of its parts.

5.6 Implementing Reuse Contracts

We will not introduce any other features of UML static structure diagrams here. We have shown how to integrate some of the most important features; other elaborations are future work. We will see in chapters 6 and 7 that this model is already rich enough to be very useful in a lot of situations.

Instead, now that participants, acquaintance relationships, etc. are mapped on tangible software items, we can discuss what it means for a software system to implement a certain reuse contract, or what it means for a reuse contract to model a software system.

A multi-class reuse contract is implemented by a set of classes. Each class complies to one class description in the contract. To correctly implement a class description a class has to comply to a number of criteria. This is stated by the definition below.

[9]It also adds an extra call to `value`, namely to the message `valueUsingSubject:`, which already exists on `ValueModel`. We omitted this in order not to clutter the example.

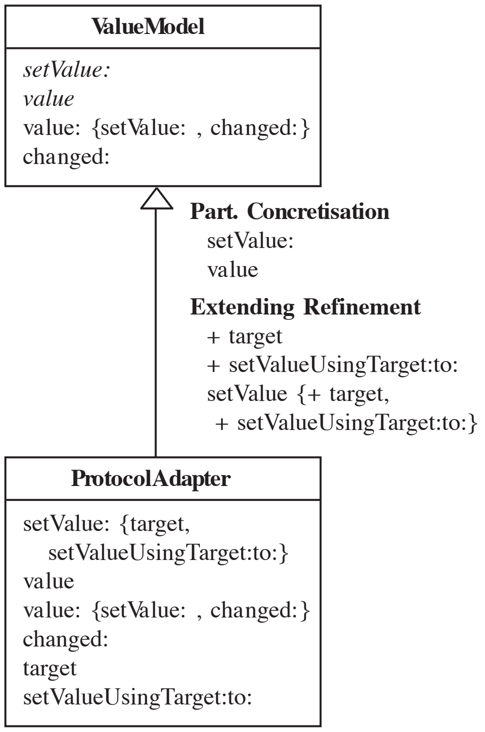
ValueModel
setValue:
value
value: {setValue: , changed:}
changed:
Part. Concretisation
setValue:
value
Extending Refinement
+ target
+ setValueUsingTarget:to:
setValue {+ target, + setValueUsingTarget:to:}
ProtocolAdapter
setValue: {target, setValueUsingTarget:to:}
value
value: {setValue: , changed:}
changed:
target
setValueUsingTarget:to:

Figure 5.11: Layered Concretisation

Definition 5.16 (Reuse Contract Implementation) A class description c **is implemented by** a class c_{imp} if

1. c_{imp} provides an implementation for all concrete methods that appear in the client interface of c ;
2. c_{imp} provides a signature, but no implementation for all abstract methods that appear in the client interface of c ;
3. for every concrete method m in the client interface of c :

   for every method invocation $a.n$ in the specialisation clause of m , where a refers to a class (with implemented by d_{imp}):

   (a) a method named n exists on d_{imp};

   (b) if the implementation of m in c_{imp} does not contain a super send: the implementation of m in c_{imp} sends the method n to d_{imp}.

   (c) if the implementation of m in c_{imp} contains a super send: the implementation of m in c_{imp} or in its superclass sends the method n to d_{imp} (possibly again through a super call).

Note that this is a very generic definition that needs to be tuned to specific languages. For example, clause 2 specifies that for all abstract methods a signature without an implementation should be provided. In Java or C++ this can be achieved by explicitly declaring the method to be respectively abstract or pure virtual. In Smalltalk this is not possible, but there is a convention to use the body "self subclassResponsibility".

Similarly, in clause 3 we assume to know to which classes acquaintance relationships refer. This is known, for example, in statically typed languages as Java and C++ in the case where the acquaintance relationship is represented by an instance variable or an argument. In dynamically typed languages or for other kinds of acquaintance relationships this is not always so clear and extra assistance from the user, providing explicit mappings for acquaintance relationships might be necessary. Moreover, we say m sends the method n to d_{imp}. It is obviously not as simple as that. The message needs to be sent along the right acquaintance relationship to the right instance. Again, user assistance might be required.

We will discuss a first experiment of checking compliance of Java code with reuse contracts in chapter 7.

5.7 Collaboration Diagrams

As announced at the beginning of this chapter, we will now consider collaboration diagrams. In the UML the convention is taken that for all type-instance pairs, for example classes and objects, associations and links, parameters and values, the type and the instance are modelled by the same geometric symbol. For classes and objects this is a rectangle. The designator strings (instance name, colon, type name) of instance elements are underlined. The basic notation of collaboration diagrams is thus the notation as adopted in chapter 2 with the possibility to include type names, here class names.

An example of a collaboration diagram with different objects of the same class is given in figure 5.12. This reuse contract depicts how composite visual structures can be represented. A `CompositePart` can refer to different `Views` or other visual items through `Wrappers`. Behaviour such as releasing is propagated from the composite through the wrappers to the subparts.

It was already mentioned in the UML specifications that one of the possible uses of the refinement relationships would be to express the connection between '*a high-level construct at a coarse granularity and a lower level construct at a finer granularity, such as a collaboration at two levels of detail*'. It is useful to apply all operators defined so far on static structure diagrams on collaboration diagrams. We thus again want to make an instantiation of reuse contracts, now dedicated to collaboration diagrams. We will not repeat all the definitions again, we will just focus on the important adaptations.

A first extension to the model is the possibility to include types. We therefore

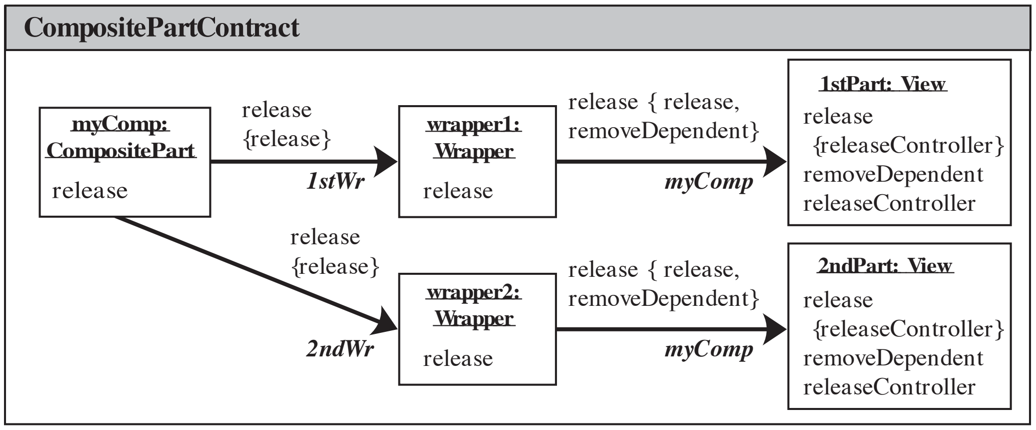

Figure 5.12: A Collaboration Diagram

need to extend the definition of basic reuse contracts.

Definition 5.17 (Collaboration Reuse Contract) A **collaboration reuse contract** is a set of object descriptions, each with

1. a name that is unique within this object description, with optionally an associated type name;
2. an acquaintance clause;
3. an interface.

We will not repeat the definitions of acquaintance clause, interface and specialisation clause here, as these are again straightforward. Note that in UML usually no interface is attached to object representations. It is however already mentioned in [BRJ97] that for special reasons a compartment containing operations is permissible.

We now need to perform some extra checks in the well-formedness definition to ensure that different objects of the same class do not give contradictory information about the interfaces of this class. For example, different occurrences of the same method need to have the same abstractness annotation. Furthermore, we take the option that all occurrences of the same method either have the same or an empty specialisation clause. For example, in the `CompositePartContract` both occurrences of `release` on `View` had the same specialisation clause, as well as both occurrences of release on `Wrapper`. It is not always necessary to repeat the specialisation clause, therefore we leave the option open to have empty specialisation clauses as well. More sophisticated rules could be developed, but for now we opt for this simpler solution.

Definition 5.18 (Well Formedness) A collaboration reuse contract R is **well-formed** if for every object description o

1. for each acquaintance relationship $a.q$ in the acquaintance clause of o : an object with name q exists in R (WF1);
2. for each method invocation $a.m$ in a specialisation clause in o :
   (a) a is an acquaintance name in the acquaintance clause of o (WF2);
   (b) m is the name of a method of the object referred to by a (WF3).
3. for every method name m , occurring in the client interface of different object descriptions with the same type name (WF4):,
   (a) all non-empty specialisation clauses of these methods are identical;
   (b) the abstractness annotation attached to these methods is equal on all occurrences.

The operators as discussed before can be preserved, but again in the modifier and applicability definitions extra conditions need to be added to ensure the newly introduced well-formedness constraints. For example, on concretisation one needs to ensure that all occurrences of the same method are concretised simultaneously and on participant refinement, one needs to ensure that there will not be two different occurrences of the same method with a different non-empty specialisation clause.

We will not elaborate on all possible extensions to this basic collaboration diagram model, but will focus on one important aspect that is also prominent in UML's collaboration diagrams: different kinds of acquaintance relationships.

5.8 Acquaintance Relationships

Until now we have acted as if all acquaintances were life-time, i.e., when a class or object is acquainted to another class or object in a contract, we pretend it always is. A distinction should be made between acquaintance relationships that are life-time and volatile relationships. In object-oriented systems life-time relationships will generally be implemented by instance variables. The relationship with self is also life-time. Volatile relationships can be modelled by method arguments or local or global variables.

In collaboration diagrams in UML five 'implementation stereotypes' are distinguished:

1. ≪*association*≫ this is the default in UML;
2. ≪*parameter*≫ a procedure parameter;
3. ≪*local*≫ a local variable of a procedure;

4. ≪*global*≫ a global variable;

5. ≪*self*≫ the link of an object with itself.

These stereotypes can be denoted along the acquaintance relationships. We add an extra stereotype ≪*result*≫, to depict that an object is acquainted with another object that was the result of one of its methods. The fact that a method returns a result will be depicted by a dashed line.

5.8.1 Extension of the Model

We again have to extend our original definition to include the possibility to add implementation stereotypes along acquaintance relationships. We therefore adapt the definition of acquaintance clauses.

Definition 5.19 (Acquaintance Clause) An **acquaintance clause** is a set of acquaintance relationships $a.o.s$, associating an acquaintance name a with an object name o , and optionally an implementation stereotype s.

We follow the same convention as UML, namely that when no implementation stereotype is included, we consider the acquaintance relationship to be an association. Note that we need to attach a method name to some implementation stereotypes, because it is not sufficient to know only the stereotype, we also need to know from which method a participant is a parameter, variable or result. We therefore have to define what kinds of stereotypes are included in the model.

Definition 5.20 (Implementation Stereotype) An **implementation stereotype** is either one of the keywords: $association$, $global$ or $self$ or a pair $(keyword, m)$ where $keyword$ is either $parameter$, $local$ or $result$ and m a method name.

We again have to add an extra clause to the well-formedness definition, to ensure that when an acquaintance relationship has the implementation stereotype ≪*parameter*≫, ≪*local*≫ or ≪*result*≫ attached to it, the method name it concerns is present on the participant.

Definition 5.21 (Well-Formedness) A collaboration reuse contract R is **well-formed** if for every object description

(WF1) . . . (WF4)

(WF5) for every object o : for every $a.p.s$ in its acquaintance clause, where s is a pair $(keyword, m)$: a method m is part of the interface of o .

The inclusion of this information could allow us to define well-formedness even more precisely. Until now, it was allowed to send a message to another participant as long as there was an acquaintance relationship and as long as the called message was part of the client interface of the receiving participant. This could be further restricted. For example, when we know that an acquaintance relationship has the stereotype $\ll$*parameter*$\gg$, we know that this acquaintance relationship is only accessible from within the method of which it is an argument. We could therefore add an extra constraint to the definition of well-formedness. In order to do that we would however need to take sequences of method invocations into account[10]. We will not do this here, because that would lead us too far. We will see further on that even this rather minimal model allows to express a lot.

Since we do not add these extra constraints, the operators are also not so much affected by this extension of the model. The context operators are again affected, because they need to take the possibility to add implementation stereotypes into account. We do however add one extra operator: context concretisation.

5.8.2 A New Operator: Context Concretisation

Motivation

In a first description of a design one can leave out specifications on what kind of acquaintance relationship a binding represents. In a second stage, one needs to make decisions about this and add implementation stereotypes. As this gives a more concrete representation of the collaboration, we call the addition of the implementation stereotypes *context concretisation.*

Definition and Properties

Definition 5.22 (Context Concretisation Modifier) A **context concretisation modifier** is a reuse modifier with modifier tag "context concretisation" and a modifier description containing pairs (o, acq) each consisting of an object name o and an acquaintance clause, where each acquaintance relationship in acq has an associated implementation stereotype.

Definition 5.23 (Context Concretisable) The **context** of a collaboration reuse contract R is **concretisable** by a context concretisation modifier M_{cc} if for each pair (o, acq), for each $a.p.s$ in acq:

1. $a.p$ also occurs in the acquaintance clause of o in R , but without an associated implementation stereotype;

2. if s is a pair $(keyword, m)$: a method m is part of the interface of o in R .

10 Because a method to an acquaintance representing a result can only be sent after the method of which it is a result has been invoked.

Definition 5.24 (Context Concretisation) If the context of a collaboration reuse contract R is concretisable by a modifier M_{cc}, then the collaboration reuse contract R_{cc} is the **context concretisation** of R by M_{cc} where:

1. R_{cc} contains all object descriptions of R that are not mentioned in M_{cc};
2. for each pair (o, acq) in M_{cc}: R_{cc} contains an object description
   (a) with name o and the same interface as o in R ;
   (b) with an acquaintance clause that contains the union of acq and all acquaintance relationships of the acquaintance clause of o on R that are not mentioned in acq.

Context concretisation also preserves well-formedness.

Property 5.3 *A context concretisation of a well-formed collaboration reuse contract is well-formed.*

Proof
The well-formedness definition now has 5 clauses. WF1 to WF4 are not affected by this operator. WF5 is preserved by the second clause in the context concretisability definition.

5.8.3 Implementing Collaboration Reuse Contracts

In section 5.6 we discussed what it means for a software system to implement a multi-class reuse contract. A simple mapping was defined between classes and class descriptions. Here a similar definition can be given, only now a class should not only comply to the specifications concerning one object, but to those on all objects of the same class.

One could try to go a step further here and also try to check the existence of the right kind of acquaintance relationships. Whether this is checkable and how is however strongly language-dependent, so we will not give a general definition here.

5.9 Conclusion

We will not continue with the integrating of reuse contracts in UML here, but rather go on with a number of experiments. We applied the basic reuse contracts model to the field of object-oriented class libraries and frameworks by integrating it in UML for two purposes. First, the two instances of reuse contracts presented here — multi-class reuse contracts and collaboration reuse contracts — allow us to apply reuse contracts to object-oriented systems, examples of which are given in the following chapters. Second, the approach followed in this chapter could also be followed to apply the basic model to other application areas. The steps that need to be taken are:

1. Map the items from reuse contracts to features in the application area. Investigate how this influences well-formedness, the reuse operators and the conflicts. For example, in our case the presence of late binding shed a new light on some conflicts.

2. Add additional features from the application area to the model. We added, among others, information about abstract and concrete methods.

3. Adapt the existing model, the well-formedness and operator definitions to cope with the features added in step 2. For example, after adding information about abstract and concrete methods the definition of participant refinement needed to be adapted to take this information into account.

4. Introduce additional operators to cope with the features added in step 2 and optionally provide additional combined operators. Each basic modelling element corresponds to a number of operators. For example, we introduced participant concretisation and abstraction.

   The new operators should be defined in the same way as the basic operators (modifier, applicability and result definitions and well-formedness property). Consequently, they can again be composed with other operators, in order to form coarser-grained ones. We thus introduced layered concretisation.

5. Investigate how the new operators of step 4 influence the conflicts and detect what possible new conflicts could arise. For example, method capture can also occur after participant concretisation and the new annotation conflict was identified.

In this chapter we integrated two diagrams from UML in the reuse contract notation. We did not consider all possible features of these diagrams, but concentrated on a number of crucial elements. We could, for example, have introduced other method annotations as private/protected/public, static, final, ... next to the annotation abstract/concrete to methods as we did here. We chose to incorporate the annotation abstract/concrete because the use of abstract methods is important to achieve reuse in object-oriented systems. It is however possible to repeat the same exercise with other annotations.

[Cor97] discusses the incorporation of the visibility annotations of Java in reuse contracts. This requires a considerable adaptation of the well-formedness definition, because subclasses cannot rely on private methods of their superclass. It also influences the operators, because it is not possible to override a method with a more private one and therefore some refinements are illegal. Based on this observation one can see how the use of private methods can be used to prevent, for example, method capture. We will not go into details concerning these other annotations here. We feel the introduction of the annotation abstract/concrete has demonstrated the feasibility of extending the model.

Note that the integration of extra features from the UML diagrams into our basic model required some adaptations to the basic model. Extra features cannot always be added orthogonally. On the one hand, a reason for this can be found in the complexity of the object-oriented model itself. On the other hand, further research might enable us to make the basic model more orthogonal in order facilitate extensions.

Next to method annotations, further elaborations to the model could be performed by further exploiting the typing information, by adding arguments and result types, by adding information on the order in which methods are invoked, etc. We will shortly touch these issues in the section on future work in chapter 8, but we will not introduce any more features here. Instead, in the next two chapters we take a look at how reuse contracts can be used in various phases of the software life cycle.