Reuse Contracts:
Managing the Evolution of Reusable Assets

Patrick Steyaert, Carine Lucas, Kim Mens, Theo D'Hondt

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
http://progwww.vub.ac.be/

Email: prsteyae@vnet3.vub.ac.be, clucas@vnet3.vub.ac.be,
kimmens@is1.vub.ac.be, tjdhondt@vnet3.vub.ac.be

Abstract. *A critical concern in the reuse of software is the propagation of changes made to reusable artifacts. Without techniques to manage these changes, multiple versions of these artifacts will propagate through different systems and reusers will not be able to benefit from improvements to the original artifact. We propose to codify the management of change in a software system by means of* reuse contracts *that record the protocol between managers and users of a reusable asset. Just as real world contracts can be extended, amended and customised, reuse contracts are subject to parallel changes encoded by formal* reuse operators*: extension, refinement and concretisation. Reuse contracts and their operators serve as structured documentation and facilitate the propagation of changes to reusable assets by indicating how much work is needed to update previously built applications, where and how to test and how to adjust these applications.*

1 Introduction

It has become a well-known fact that the degree to which software reusability is pursued is matched by the extent to which the same reusability fails to fulfil the high expectations. A delicate balance between longer term investments in reusable artifacts and the need to meet deadlines needs to be accomplished. To be properly reusable, artifacts should undergo some form of certification thereby turning them into reusable assets [2]. To be able to leverage on the investment, reusers must be able to benefit from future improvements of the assets they reuse: proper evolution of reused assets should not invalidate previous reuse. In a similar vein, reuse should go beyond the act of copying out code fragments and adapting them to current requirements without regard for the evolution of the reused fragments. This implies the management of some kind of consistency in the evolution of reusable software. The absence of such management mechanisms is recognised as an important inhibitor to successful reuse [2, 12, 15].

In this paper we not only recognise the need for software to evolve both during its initial design and when it is being reused, we actually advocate the development of a methodology for managing change in the process of engineering reusable software. We advocate a kind of reuse that, unlike black-box reuse, allows reusable assets to be adapted before reuse. This makes our approach more akin to white-box reuse: it cannot always be guaranteed that changes to a given reusable asset will propagate without invalidating previous reuse. Unlike white-box reuse, however, reuse is controlled in a way so that reusers know when and how the assumptions they make about reusable assets are broken when this asset is changed.

We propose to codify this management of change in an (object-oriented) software system by means of reuse contracts. *Reuse contracts* are interface descriptions (of, for instance, classes), offering guidelines for reusing assets in some problem domain and recording the protocol between managers and users of a reusable asset. Similar to

the real world where contracts can be extended, amended and customised, reuse contracts are subject to typical *reuse operators*: extension, refinement and concretisation. Together, reuse contracts and operators can be used by the asset producer to document that part of the design that is relevant for reusers. Moreover, they document the assumptions made by reusers about the way an asset is reused. This documentation facilitates the propagation of changes to assets by indicating how much work is necessary to update previously built applications, where and how to test and how to adjust these applications. Reusers thus can benefit from improvements to the assets they reuse and the proliferation of different versions of reusable assets can be kept to a minimum. In a similar vein, reuse contracts can be valuable to the asset developer to assess the impact of changes and to decide whether changes should be made.

As a tangible case in which to explain our ideas, we focus on class abstractions with inheritance as the reuse mechanism. This is by far the best-known and most widely used technique available today for structuring object-oriented software. We use abstract classes, specialisation interfaces [7] and the different reuse operators, as a framework in which to study change management of method overriding. We analyse some of the problems that can arise in inheritors when changes are made to a parent class and show how reuse contracts can help in solving these problems. While the proposed reuse operators are sufficiently expressive for our discussion, it is beyond the scope of this paper to give a formal treatment of their completeness. Interested readers can verify this and other properties in another document [8].

The major contribution of this paper is the identification of a framework for the codification of the evolution of reusable assets. Class hierarchies are used as a specific case in which to explore a consistent set of reuse operators. Our conjecture is that this approach is applicable to other and possibly more general adaptable systems. We even dare to suggest that reuse contracts have a broader scope than managing evolution in reusable assets: they shed light on the

architecture of the asset, can be used as structured documentation and generally assist a software engineer in adapting assets to particular needs. They thus help break down the barriers between asset producers and asset reusers. When adopted, reuse contracts may significantly enhance the way in which software is being built and managed.

2   Conflicts with Evolving Parent Classes

As a concrete case, in this paper we focus on reuse based on abstract classes and inheritance. Because the power of inheritance lies in the ability to override methods that are invoked by methods of the parent class, the possible problems with inheritance become apparent when considering the *calling structure* between a class and its inheritors.

A class hierarchy can evolve in different ways (see Fig. 1). First, a class hierarchy evolves simply by editing the classes in it. Since this corresponds to the replacement of a class with a new, modified version, and this class may already have a number of inheritors, this form of evolution is called *parent class exchange*. Second, classes evolve through inheritance. Rather than just editing it, a class's description gets "modified" by inheritors. This kind of evolution is called *layering*: a class's description evolves through a chain of inheritors, each adding its own layer of modifications. The only difference with parent class exchange is the availability — after modification — of *both* the original class *and* the modified class.
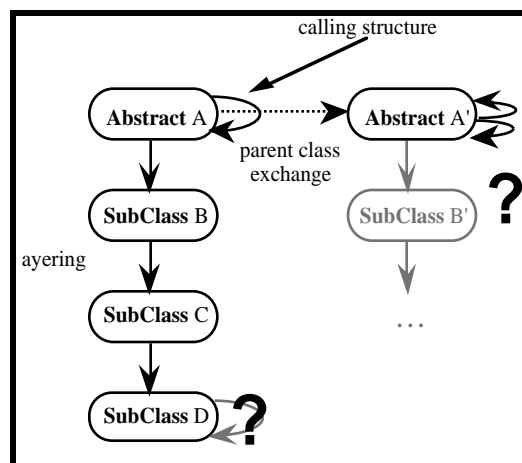

Fig. 1   Problems when Reusing Classes

The problem with class evolution in general, is that modifications made to the calling structure of a parent class can introduce conflicts in already existing inheritors. With layering, the calling structure of a parent class gets modified by a chain of inheritors, which makes it difficult to assess the impact of the modifications made by intermediate inheritors. This is perceived by many users of object-oriented class libraries and frameworks as one of the major obstacles to reuse them. Similarly, with parent class exchange, it is difficult to detect whether changes made to the parent's calling structure introduce conflicts in inheritors. As both problems are similar, we will only focus on the latter. We take a closer look at four specific conflicts introduced by parent class exchange.

2.1 Conflicts in the Method Interfaces

The first problem involves the case where an exchanged parent class introduces a new method, while one of the inheritors had previously introduced a method with the same name. If this method is not invoked by any of the other methods of the parent, at first sight, this causes no erroneous behaviour (the case where this method is invoked by other methods of the parent is discussed in section 2.3). The method of the inheritor will simply override the method of the parent class. But, as a consequence, the intention represented by the adaptation of the parent class will get lost. Moreover, as this causes no technical error, this loss will probably go unnoticed.

A related problem is that of conflicts in annotations made to a method's interface. It is not uncommon to attach extra design information to methods. As inheritors rely on this information, this can lead to extra conflicts. For example, methods can be marked as being abstract or concrete. Inheritors that override an abstract method assume that the overridden method has no implementation. This can lead to a conflict in inheritors when the parent is exchanged with a class where this abstract method is already made concrete. Apart from an annotation `abstract` or `concrete`, we will also add other design information to the interface.

2.2 Unimplemented Methods

Another kind of problem is where an exchanged parent class introduces a new abstract method, that might be invoked by other methods of the exchanged parent. When already existing inheritors do not provide an implementation for these newly added abstract methods, this results in incomplete classes.

2.3 Method Capture

The exchanged parent class can include extra method invocations. These could be invocations of methods that are provided by this parent class itself, but also invocations of methods that are implemented by an inheritor and that were not invoked before. In the latter case, we say that these methods of the inheritor *get captured* by the methods of the parent that invoke them. This may result in erroneous behaviour, as the inheritor did not take into account that its method would be invoked by the parent. Two cases of method capture can be distinguished. First, the case where a method that did not exist in the original parent class is now introduced and invoked. An inheritor might have introduced a method with the same name, that will now be invoked by the new parent. As this could not be foreseen, we call this *accidental* method capture. Second, in the exchanged parent, a method that already existed in the original parent might be invoked by more methods than before. An inheritor might have overridden this method, and this implementation will now be invoked by more methods of the parent. As this can be foreseen more easily, we call it *regular* method capture.

A good example of how a similar kind of problems can occur in class hierarchies can be found in the standard Smalltalk-80 class library. The reason why the names of primitive methods (e.g., `basicAt:`, `basicAt:put:`, ...) on the root class (`Object`) are prefixed with 'basic' is exactly to avoid users accidentally introducing a method with the same name in their classes, thus causing a situation similar to method capture.

2.4 Inconsistent Methods

When method invocations are omitted — which is frequently done for performance reasons — the inverse situation of method capture can arise. If on parent class exchange the new parent class performs less invocations of a method than before, this might lead to inconsistent behaviour. We then say that the method that used to perform the invocation has become *inconsistent* with the method that used to get invoked. This terminology is due to Kiczales and Lamping [6].

Consider the prototypical example of a `Collection` hierarchy. A class `Set` defines a method `add` and a method `addAll`, which invokes `add` to add a group of elements to the set simultaneously. A subclass `CountingSet` might override `add` to keep a counter of the number of elements added to the set. If on parent class exchange in a new version of `Set`, `addAll` does not invoke `add` anymore, this will lead to inconsistent behaviour in `CountingSet`, as not all additions will be counted. `addAll` has become inconsistent with `add`.

3 Reuse Contracts

3.1 Documenting Dependencies through Specialisation Interfaces

To be able to detect and solve the above problems it is crucial to have the right kind of documentation. Currently, the main mechanism to document the design of object-oriented software systems is through abstract classes [5]. The client interface of an abstract class sketches the design of future concrete subclasses, by indicating which methods they should provide. However, information about internal dependencies is also a crucial piece of design information to inheritors. This information can usually only be acquired by inspecting the code. To solve this problem, Lamping introduced specialisation interfaces as a means to document the calling structure of a class, by explicitly naming all methods that are invoked *through self sends* in each method [7]. Consider, for example, an abstract class `AbstractView`[1], that describes the general behaviour of a view that

visually represents a certain subject. A view can be drawn and updated. The method `Update` relies on the method `Draw` and on a number of other methods for its implementation.

```
Class AbstractView
  Abstract Draw()
  Concrete Update() [ self.SetPen(2);
                      self.SetRect(40,60);
                      self.Draw() ]
  Concrete SetPen(size)          [ … ]
  Concrete SetRect(height, width) [ … ]
End Class
```

The specialisation interface of `AbstractView` is shown below. The information on the self sends is provided by enumerating the names of the referenced methods between curly braces after the method's signature.

```
Specialisation Interface AbstractView
  Abstract Draw
  Concrete Update  { Draw,SetPen,SetRect }
  Concrete SetPen  { … }
  Concrete SetRect { … }
End Specialisation Interface
```

While specialisation interfaces are an important step towards achieving actual reuse of design, their main drawback is that they document the internal dependencies of a class by listing *all* self sends in a class. They do not distinguish between implementation level and design level dependencies and thus do not allow hiding of implementation details. While providing too little information about internal dependencies makes it impossible to define large applications in an implementation independent way, exposing too much implementation detail restricts the ability to evolve or to allow different implementations. It is thus very important to expose only those parts of the internal structure that are crucial to the design of a class. Reuse contracts are introduced in the next section as a mechanism that distinguishes between that part of the specialisation interface that is implementation dependent and that part that is crucial for inheritors. They thus hide those dependencies on which inheritors should not rely.

A remark must also be made about the kind of information that specialisation interfaces can provide. Specialisation interfaces can be specified either by listing method dependencies purely based on names, or by including type information,

---

[1] Throughout this text we use a part of a GUI library as an example. The example is inspired by [4] and [6].

or by including semantic information that specifies, for example, the order in which methods should be invoked. The art is in finding the right balance between descriptions that are easily understood and expressed, and descriptions that capture enough of the semantics of possible adaptations. Reuse contracts will only indicate which methods rely on which other methods, by enumerating the names of methods that are invoked through self sends. Although reuse contracts provide only syntactic information, this is enough to firmly increase the likelihood of behaviourally correct exchange of parent classes.

For simplicity we have restricted the reuse contracts we propose here to include only documentation on the internal dependencies among a class's methods. The dependencies among the methods of one class and the methods of its acquaintances are at least as important. Just as our current reuse contracts are based on specialisation interfaces, reuse contracts could be developed based on descriptions of interclass relationships. Such descriptions have already been studied in the form of *contracts* [3, 4], but will not be discussed here.

3.2   Definition of Reuse Contracts

A reuse contract is a set of method descriptions that is divided into two subsets: the abstract and the concrete method descriptions. Reuse contracts are only interfaces: they never contain actual methods, only descriptions of methods. Each method description consists of a method name together with a specialisation clause and an annotation `abstract` or `concrete`. For reasons given in the previous paragraph, the specialisation clauses only list the methods that are crucial to the design of a particular method. Methods that are listed in specialisation clauses are called *hook methods* and can be abstract as well as concrete. This term should not be confused with the term *template methods*, which is usually used to describe concrete methods that invoke abstract methods in their implementation. Finally, similarly to classes and inheritance, new reuse contracts can be derived from existing ones. Unlike inheritance we will have different operations to obtain a *derived*

reuse contract; e.g., a reuse contract can be *an extension of* another one. We will discuss the meaning of the various possible relationships in section 4. Summarising, we a reuse contract can be defined as follows:

A **reuse contract** is an interface, i.e., a set of method descriptions each consisting of
• a unique name,
• an annotation `abstract` or `concrete`,
• a (possibly empty) specialisation clause.

Furthermore, for a reuse contract to be well-formed it needs to satisfy certain conditions:

A reuse contract is **well-formed** if every name occurring in one of the specialisation clauses corresponds to a method description appearing in the reuse contract itself;
Well-formed reuse contracts can be explicitly related to other reuse contracts by one of the following operators: concretisation, extension, refinement, abstraction, cancellation, coarsening.

Unless explicitly stated otherwise, from now on when we use the term "reuse contract" we mean a well-formed reuse contract. As a first example, consider a reuse contract `View` describing the abstract class `AbstractView` from section 3.1.

```
Reuse Contract View
  Abstract
    Draw
  Concrete
    Update      { Draw }
End Reuse Contract
```

As discussed, this reuse contract is subdivided into an abstract and a concrete section. While `AbstractView`'s specialisation interface enumerated *all* methods invoked through self sends in `Update`, in the reuse contract `View` only `Draw` is mentioned in the specialisation clause of `Update`, as the invocation of this method is the only one crucial to the design. The other methods that were invoked through self sends in `Update` were pure implementation methods and are therefore not included in the reuse contract. Not only did we leave `SetPen` and `SetRect` out of the specialisation clause of the `Update` method, we also did not

include them in the list of method descriptions in the reuse contract `View`.

3.3 Implementing Abstract Classes that Comply with Reuse Contracts

Since reuse contracts only describe interfaces, implementations must be associated with them. As these implementations should satisfy the design imposed by a reuse contract, the classes should comply with the reuse contracts in some way: we say that a reuse contract *is implemented by* an (abstract) class when the conditions below are fulfilled.

A reuse contract R **is implemented by** a class C if
(1) C provides an implementation for all concrete method descriptions of R;
(2) C provides a signature, but no implementation for any abstract method description of R;
(3) for every name n in a specialisation clause of a concrete method description m of R, the method in C with name n is invoked through a self send by the method corresponding to m (or by a method of C that is directly or indirectly invoked by m).

According to this definition the abstract class `AbstractView` is a possible implementation of the reuse contract `View`. The definition also immediately implies that reuse contracts containing abstract method descriptions are implemented by abstract classes.

The need to explicitly declare reuse contracts as well as the compliance of classes to reuse contracts could be criticised for too much verbosity. Some remarks can be made about this comment. In languages that promote the separation of interface and class hierarchies, reuse contracts can be introduced as an extension to these interfaces. In that case, reuse contracts do not introduce too much overhead, since interfaces must be specified anyway. For languages that do not include interfaces, reuse contracts can be managed by the programming environment. In that case, reuse contracts could be semi-automatically constructed on the basis of the calling structure (the programmer only has to delete the descriptions

and names of methods that should not be exposed). Moreover, since reuse contracts are design concepts, they should actually already have been constructed during the design phase.

A problem can occur with classes that include implementation-specific methods not specified by the reuse contract. As these methods are not specified in the reuse contract but only in the implementation, it is possible that a user later accidentally introduces a method with the same name in the class implementing a derived reuse contract. As discussed in section 2.3., method capture might occur. The classes that implement the reuse contracts should therefore encapsulate the invocation of their implementation-specific methods (whether they are public or private) from the derived reuse contracts and their implementations. Different approaches are possible to achieve this. Kiczales and Lamping suggest using a package system such as the Common Lisp package system or information hiding facilities such as provided in languages like C++ [6]. Another possibility is that a tool checks for this kind of errors and requires the user to rename one of the methods involved. Yet another option is provided by languages that offer the possibility to explicitly encapsulate self sends [1, 14]. In such languages, the self sends to implementation-specific methods can be explicitly encapsulated so that they are invisible to future inheritors.

Another problem that may occur with implementation-specific methods is that the specialisation clauses might demand that a method m performs a self send to some method n, while this self send is not made in m itself, but only in one of the implementation-specific methods invoked by m. Therefore, when verifying whether an implementation satisfies a reuse contract, it is actually necessary to look at the *transitive closure* of all self sends made from within methods. This is the motivation behind the phrase: "or by a method of C that is directly or indirectly invoked by m" in clause 3 of the above definition.

Note that even when a self send appears in the implementation of a method, it is not always

possible to check — without extensive data flow analysis — whether this invocation will actually be performed at run-time. It could for instance occur in the body of a conditional expression. As discussed in the previous section, reuse contracts do not aim at fully specifying the design of classes at a behavioural level, only through descriptions.

4 Operators on Reuse Contracts

Reuse contracts are only a first step towards solving the problems concerning parent class exchange from section 2. Without reuse contracts it is difficult to detect problems such as method capture because specialisation interfaces are not explicitly available. But even making specialisation interfaces explicit does not suffice in order to detect problems on parent class exchange. More information is needed both on the assumptions made by inheritors about their parent classes and on the way the parent classes are actually reused. Consider exchanging a parent class with a new parent class that introduces a new method m. When looking at plain inheritors, it is not always clear whether a method with the name m in the inheritor was intended to *override* the method m of the parent class or whether it was intended as a *new* method leading to unintended method capture. These two cases can only be differentiated by meticulously comparing the reuse contracts of the old parent class, the new parent class and the inheritors. This is neither practical (in practice the old parent class might not even be available anymore), nor intuitively compelling. We propose a methodology that is more intuitive for both inheritors and developers of reusable classes and guides them in managing changes to these classes. It is based on a categorisation of the typical actions undertaken by the designers of both abstract classes and inheritors and the changes these actions cause in the calling structure. We essentially distinguish three different logical operators on reuse contracts: concretisation, refinement and extension and their inverse operators: abstraction, coarsening and cancellation. Although not the only operators imaginable, they do coincide with the typical ways

to use abstract classes. By examining the interactions between these operators and by investigating which operators respect the design, rules can be proposed that facilitate exchange of reuse contracts of parent classes. In the above example, had it been clear that the inheritor intended to perform an extension (i.e., introducing a new method), it would have been easier to detect possible problems.

So, in order to be able to correctly assess the impact of parent class exchange, reuse contracts must be labelled with how they are derived from other reuse contracts in terms of the basic operators. For the programmer, the work effort involved is obviously not the adding of an extra keyword (i.e., extension, concretisation, ...), but rather the careful construction of the derived reuse contract corresponding to the inheritor by means of the basic operators. While this puts an extra burden on the programmer requiring making more conscious decisions during the design process, the pay-off of this work will become clear in the next sections. Moreover, supporting tools can be constructed that automatically decompose the derived reuse contract of an inheritor in terms of the basic operators. Such tools can also be an extra help in understanding how an inheritor differs from its parent.

In this section we will define the different operators, in the next section their interactions will be examined and we will discuss how this information can be used to detect and solve problem cases. While the definitions of the operators and the discussion of their interactions were developed on a much more formal level [8], they are presented in this paper in a way that makes their intuition more clear. For every operator a definition, a description of the prerequisites and an inverse operator are given. Each operator will be defined in terms of a modifier M. This modifier is necessary to discuss parent class exchange, as for an inheritor having its parent class exchanged comes down to applying the same modifier that was applied to the

former parent, to the new parent[2]. The prerequisites for each operator will describe the exact form of these modifiers, as well as the conditions they must comply with in order to be correct. Modifiers themselves are reuse contracts, although not necessarily well-formed ones.

Every R and R_x in the following definitions represent a well-formed reuse contract. Every M and M_x represent a *reuse modifier*, i.e., a reuse contract that is not necessarily well-formed. In the examples we will not explicitly mention the modifiers, as they will always be clear from the context.

4.1 Concretisation

Concretisation is an operation that is typically performed by an application developer, when customising assets to obtain applications tailored to a certain domain. It makes an asset more concrete, by overriding (some) abstract method descriptions with concrete ones.


R_c **is a concretisation of** R **with** M_c if
(1) R is concretisable with M_c (defined below);
(2) R_c contains only method descriptions with the same name and specialisation clause as in R;
(3) every abstract method description of R listed by M_c becomes concrete in R_c;
(4) all other method descriptions in R remain unchanged in R_c.

When we are not explicitly interested in the modifier M_c, we simply say that R_c *is a concretisation of* R. As concretising a reuse contract can happen in several steps, we distinguish *complete* concretisations from *partial* concretisations. Whereas the former yield new reuse contracts containing only concrete method descriptions, the latter result in reuse contracts that still contain abstract method descriptions and need subsequent concretisations. The effect of a concretisation is depicted in Fig. 2. Note that a concretisation does not change the calling structure, nor does it add new method descriptions.

---

[2] This is similar to the way inheritance is achieved in languages with mixin-based inheritance [1].
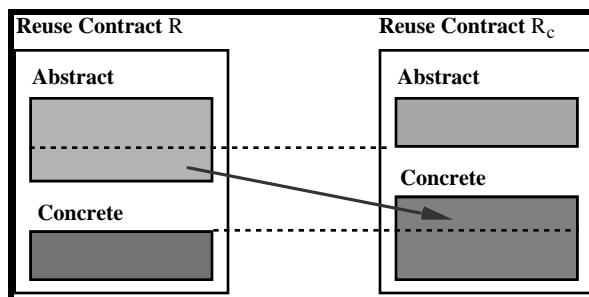
Fig. 2 Concretisation

As concretisations merely transform some abstract method descriptions of R to concrete ones, at first sight a correct concretisation modifier M_c should only mention their names. Although including information on the specialisation clauses as well seems to be redundant, it is necessary in order to avoid conflicts in the method interfaces, as will be explained in section 5.1. On the other hand, since only abstract method descriptions can be concretised, including the annotation is not really necessary. Nevertheless, to be consistent with the other definitions we include this redundant information.

R **is concretisable with** M_c
if every method description of M_c
(1) is concrete;
(2) has a name corresponding to an abstract method description of R;
(3) has the same specialisation clause as in R.

The following reuse contract `ViewPort` is an example of a complete concretisation of `View`. It transforms the only abstract method description `Draw` of `View` into a concrete one.

```
Reuse Contract ViewPort
is a concretisation of View
  Concrete
    Draw
    Update      { Draw }
End Reuse Contract
```

As there are no abstract method descriptions in this reuse contract, the keyword **Abstract** was left out.

The opposite of a concretisation is called an *abstraction*. Abstraction will not often be used, but might, for example, be wanted by a library developer to add a more abstract layer to a class library.

<table>
<tr><td>R_a **is an abstraction of** R</td></tr>
</table>

R_a **is an abstraction of** R
iff   R **is a concretisation of** R_a

This means that R_a is an abstraction of R if some method descriptions that were concrete in R are made abstract in R_a and no other changes occur.

4.2 Extension

An extension adds new method descriptions to a reuse contract. An application developer might use extension to introduce new method descriptions to express a certain behaviour particular to the application; the developer of reusable assets might use it to enhance an asset's functionality.

R_e **is an extension of** R **with** M_e if
(1)   R is extendible with M_e (defined below);
(2) R_e contains all method descriptions of R plus all method descriptions of M_e.

An extension is called *concrete* if no abstract method descriptions are added, otherwise it is called *abstract*. A legal extension modifier M_e must fulfil the following conditions:

R **is extendible with** M_e if
(1) M_e contains no method description with the same name as a method description in R
(2)   the specialisation clauses of method descriptions in M_e contain only names of method descriptions occurring in M_e or R.

Condition 1 is included to make the different operators more orthogonal. Just as a concretisation cannot add new method descriptions (condition 2 of concretisable) and thus perform an extension, an extension cannot affect any existing method descriptions.
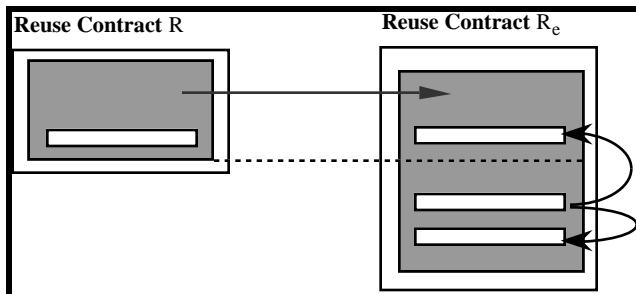
Fig. 3 Extension

The reuse contract `DragableView` below is an example of a (concrete) extension of the reuse contract `View`. An extra method description `Drag` is added, representing a method that has to invoke `Draw` to redraw the view at its new position.

```
Reuse Contract DragableView
is an extension of View
  Abstract
    Draw
  Concrete
    Update      { Draw }
    Drag        { Draw }
End Reuse Contract
```

The opposite of extension is called *cancellation*. Cancellation will mainly be used by an asset developer to remove unnecessary behaviour from the asset.

R_c **is a cancellation of** R
iff   R **is an extension of** R_c

A cancellation merely removes existing method descriptions. Of course, this operation can only be performed if the method descriptions that need to be removed are not listed in the specialisation clauses of any other methods (unless these are removed as well).

4.3 Refinement

Finally, refinement is the operation of overriding method descriptions in order to refine their design. It can be performed by an asset developer to model the evolution of an asset, or by an application developer to make reuse contracts specific to some application domain, thus creating a more layered design. This is achieved by adding extra hook methods to the specialisation clauses of the original method descriptions. Since the already existing hook methods are maintained, the design of the original reuse contract is preserved. By adding hook methods, it is refined.

As an example, consider the reuse contract `Button` below which is a refinement of `View`. It refines the method descriptions `Update` and `Draw`. Where `Draw` originally had an empty specialisation clause, it now lists a new abstract method `Geometry`. Obviously, this new method description also needs to be added to the reuse contract. Because a button is always visually represented as 'on' or 'off', `Update` is also refined to depend on this status. To represent this behaviour, `Update` must rely on

Choose and UnChoose, which in turn rely on Refresh to draw the button in either status.

```
Reuse Contract Button
is a refinement of View
  Abstract
    Geometry
    Draw            { Geometry }
  Concrete
    Update          { Choose, UnChoose }
    Choose          { Refresh }
    UnChoose        ( Refresh }
    Refresh         { Draw }
End Reuse Contract
```

As the example illustrates, refinements can also "extend" a reuse contract with new method descriptions. This is the only place where the functionalities of the operations partially overlap. Condition 6 below restricts this overlap by stating that new method descriptions can only be added by a refinement if they are (directly or indirectly) called by one of the refined methods. This is illustrated in Fig. 4. Otherwise, adding method descriptions is an extension.
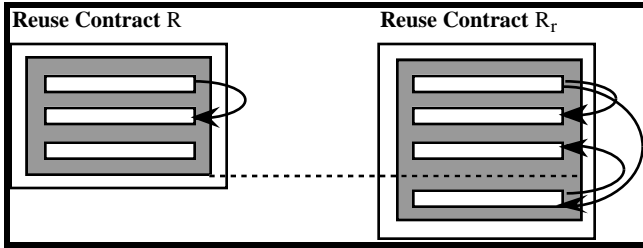
Fig. 4   Refinement

Note that while the old specialisation clause of Update listed Draw, the new one only lists Choose and UnChoose. This does however indirectly lead to Refresh (because Choose and UnChoose list Refresh), and eventually to Draw (because Refresh lists Draw). When these indirections are taken into account, it is clear that Update's new specialisation clause is indeed an augmentation of the old one. Therefore, it is necessary to work with the transitive closure of the specialisation clauses. Summarising all this we can define a refinement in terms of an overriding modifier M_o and an extending modifier M_e, as below.

A refinement is called *concrete* if none of the added method descriptions (described by M_e) are abstract, otherwise it is *abstract*.

R_r **is a refinement of** R **with** (M_e, M_o) if
(1) R is refinable with (M_e, M_o) (defined below);
(2) R_r contains all method descriptions of M_e and M_o as well as all method descriptions of R not corresponding to a method description in M_o.

For a modifier pair (M_e, M_o) to express a correct refinement, the following constraints need to be satisfied:

R **is refinable with** (M_e, M_o) if
(1) M_e contains no method descriptions with the same name as method descriptions in R;
(2) M_o contains only method descriptions with the same name as method descriptions in R;
(3) the method descriptions in M_o have the same annotations as the corresponding method descriptions in R;
(4) the specialisation clauses of method descriptions in M_e or M_o contain only names of method descriptions occurring in M_e, M_o or R;
(5) the transitive closures of specialisation clauses in M_o are augmentations of the transitive closures of the corresponding specialisation clauses in R;
(6) the name of every method description in M_e must occur in at least one of these augmented specialisation clauses.

Note that a refinement does not change the annotation abstract or concrete attached to method descriptions. Nevertheless, again to avoid method interface problems, this information is included.

The opposite of a refinement is called a coarsening. We discuss coarsening a bit more in depth than the other inverse operators, because it is an important operator in practice. Coarsening is achieved by omitting hook methods from the specialisation clauses of method descriptions. Although this means partially ignoring the design of an asset, this is often done for performance reasons or because some parts of the design are irrelevant to certain domains.

Coarsening is defined as:

R_{coarse} **is a coarsening of** R   if
(1) every method description in R_{coarse} corresponds to a method description in R;
(2) these method descriptions are exactly the same as in R except that the transitive closure of their specialisation clause in R_{coarse} can be smaller than in R;
(3) method descriptions with names that were mentioned in specialisation clauses in R but are no longer mentioned in *any* specialisation clause in R_{coarse} *can* be removed from R_{coarse}.

The motivation behind 3 is twofold. First, to avoid errors, only methods corresponding to method descriptions that are not invoked anymore can be removed. Second, if these method descriptions were not invoked in R either, then the operation concerned would be a cancellation rather than a coarsening.

To keep things simple, the above definition of coarsening was not given in terms of reuse modifiers. An actual coarsening modifier should associate a coarsening clause with each method, indicating which method invocations are removed. This information will be used later on to detect the problem of inconsistent methods.

4.4   Implementation of Reuse Operators

All operators on reuse contracts that are defined in this paper can be achieved on their implementor classes through inheritance. We could, for example, have an abstract class `GeneralView` implementing the reuse contract `View` and a concrete class `MacintoshView` implementing the reuse contract `ViewPort`. While `ViewPort` is a concretisation of `View`, `MacintoshView` is a subclass of `GeneralView`.
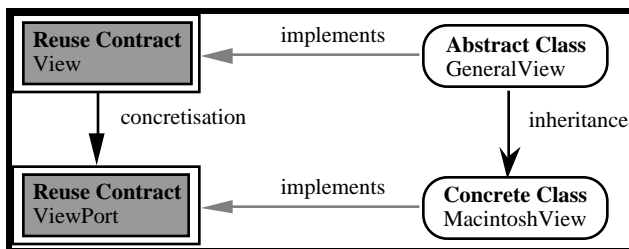

Fig. 5 Implementation of Reuse Operators

Note that this example is a simplification. First, because the techniques to encapsulate invocations of implementation-specific methods as discussed in section 3.3. must be applied here. Second, because the relation between reuse contracts and the classes that implement them is presented as one-to-one. This might not always be the case. Different reuse contracts can provide different views on a single class. Conversely, a single reuse contract can be implemented by a chain of classes, rather than a single class. In such inheritance chains, the implementor can resort to plain code reuse. Obviously, the responsibility to avoid method capture and other problems within these inheritance chains is then left to the implementor.

5   Managing Parent Class Exchange through Reuse Contracts

In this section we discuss how reuse contracts help in managing parent class exchange. Rather than plainly examining exchange of parent classes, we will investigate what the effect is of exchanging the associated reuse contracts. We will call this *base contract exchange*. Reuse contracts corresponding to parent classes are called *base reuse contracts* (or short, base contracts), and reuse contracts corresponding to the inheritors are called *derived (reuse) contracts*. This is depicted in Fig. 6.
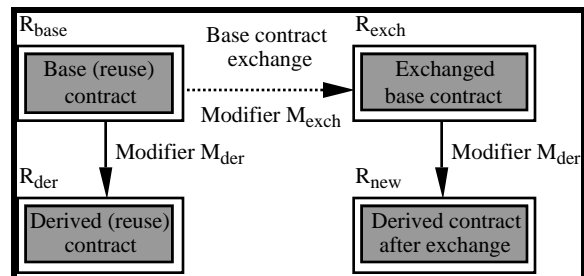

Fig. 6   Base Contract Exchange

To examine the effect of base contract exchange, we investigate what happens when the same modifier M_{der}, that created a derived contract from a base contract, is also applied to the exchanged base contract. Unlike plain parent class exchange, base contract exchange allows the detection of conflicts by a set of simple rules. It is sufficient to check whether the modifier M_{der} is still applicable to the exchanged base contract (i.e., concretisable,

extendible or refinable). If so, we can safely conclude that no assumptions made by the reuser about the base contract are violated. Non-applicability indicates the existence of a conflict between the exchanged base contract and these assumptions. This is further explained in the next section.

More subtle conflicts, such as method capture and inconsistent methods, cannot be detected without further information on how the calling structure in the exchanged base contract has changed. Exactly this information is documented in the exchange modifier M_{exch} (see Fig. 6). Rules to detect these conflicts are based on the interaction between M_{exch} and M_{der}.

As we will discuss the conflicts that may arise by combining modifiers one by one, the results must be iteratively applied for base reuse contract exchanges that involve several modifiers. It is proven in [8] (and illustrated in a prototype tool for Smalltalk classes) that even if these modifiers are not explicitly known, they can always be computed by comparing the concerned reuse contracts.

5.1 Conflicts in the Method Interfaces

A first set of problems concerns conflicts of method names or annotations, or specialisation clauses attached to a method. These conflicts can be detected by checking whether the modifier that created the derived contract is still applicable to the exchanged base contract. Non-applicability indicates that the reuser's assumptions have been violated. Moreover, by taking a closer look at the particular operations involved we can differentiate between different kinds of conflicts.

Name Conflicts: Extension versus Extension

When both the exchanged base contract and the derived contract are created through an extension of the original base contract (i.e., in Fig. 6 both M_{exch} and M_{der} are extensions), name conflicts might occur. More specifically, a name conflict occurs when M_{der} introduces a new method description with the same name as a method description introduced by M_{exch}. This can be detected by checking whether the exchanged base contract is extendible with M_{der}.

Note that name conflicts can also occur when M_{exch} or M_{der} (or both) are refinements that introduce new methods. To detect this, only the extension part of the refinement needs to be taken into account.

These problems are comparable to problems concerning multiple inheritance and can be solved with similar techniques. In languages without multiple inheritance they have to be solved through renaming or hiding.

Annotation Conflicts: Concretisation versus Concretisation

When both M_{exch} and M_{der} are concretisations, a conflict may occur when the same method gets concretised twice. This can easily be detected as the exchanged base contract will not be concretisable anymore with M_{der} (condition 2 of concretisability will be violated, as the method is already concrete). Only when both concretising modifiers M_{exch} and M_{der} manipulate a disjoint set of method names is there no problem. An analogous reasoning holds when both M_{exch} and M_{der} are abstractions.

Although such annotation conflicts will not cause a problem for the corresponding classes technically, the knowledge that M_{der} is no longer a correct concretisation indicates that there might be a problem on the behavioural level, because the concretisation (and the corresponding implementation) given by the exchanged base contract will be ignored. The reuser can solve this problem either by removing M_{der}, thus accepting the concretisation M_{exch}, or by turning M_{der} into a refinement which combines both concretisations, or by turning M_{der} into a coarsening which (partially) ignores the concretisation M_{exch}.

Specialisation Clause Conflicts: Refinement versus Refinement

When M_{exch} and M_{der} are both refinements that refine the same method a conflict might arise. This is the case when the specialisation clause of this method in M_{exch} contains more names than the corresponding specialisation clause in M_{der}. In that

case, M_{der} will not be a correct refinement of the exchanged base contract. The reuser can solve this problem by changing the M_{der} modifier into a coarsening thereby indicating that he has no interest in "respecting" the design of the exchanged base class. A more elaborate solution is to re-implement the conflicting method as a correct refinement.

Mixed Conflicts: Concretisation versus Refinement

Mixed conflicts concerning the specialisation clauses, as well as the annotations `abstract` or `concrete` can occur when M_{exch} is a refinement and M_{der} is a concretisation or vice versa.

In the case where the base contract is exchanged for a refined version and a derived contract was made by performing a concretisation of the base contract, the concretisation M_{der} will not always be applicable to the exchanged base contract. Consider, for example, the reuse contract `ViewPort` which concretises the `Draw` method of `View`, and the reuse contract `Button` which refines the `Draw` method of `View` to invoke a newly introduced method `Geometry`. Despite the orthogonality of concretisation and refinement, the concretisation that created `ViewPort` from `View` cannot be applied to `Button`. Applying this concretisation could lead to incorrect behaviour as the implementation of `Draw` corresponding to `ViewPort` is not required to perform any self sends, while the implementation of `Draw` corresponding to `Button` is obliged to

invoke `Geometry`. For this reason, a concretisation can only be applied to reuse contracts where the methods to be concretised have the same specialisation clauses. This is the motivation behind condition 3 in the definition of concretisable.

Similar problems occur when the derived contract is a refinement and the base contract is a concretisation of the original base contract. Information on the annotations `abstract` or `concrete` in a refinement modifier is necessary as refining an abstract method is essentially different from refining a concrete method, since in the first case no implementation is required while in the second case there is. This is the motivation behind condition 3 in the definition of refinable.

The only remedy is to update the conflicting parts (and the associated reuse contracts), so that they do take the extra design information into account.

Summary

The continued applicability of the reuse modifiers indicates to reusers which parts of their applications can be trusted, and which parts might introduce behavioural problems. This set of problems can be further subdivided, depending on the kind of operations involved, as summarised by the table below.

For refinements, a distinction is made depending on whether refinability fails due to a conflict in its extension part or in its overriding part.

Base contract exchange *Operation to create derived contract*	Concretisation	Extension / Extension Part of Refinement	Overriding Part of Refinement
Concretisation	annotation conflict	no method interface conflict	mixed conflict (1)
Extension / Extension Part of Refinement	no method interface conflict	name conflict	no method interface conflict
Overriding Part of Refinement	mixed conflict (2)	no method interface conflict	specialisation clause conflict

5.2   Unimplemented Methods

The second kind of problem, somewhat related to method interface conflicts, is that of unimplemented methods. This problem occurs, for

example, when a base contract is exchanged with an extended or refined version that adds new abstract method descriptions. A concrete reuse contract that was derived from a base contract

through a concretisation is not concrete anymore when derived from the exchanged base contract and needs additional concretisations. Only when the extension or refinement is concrete (i.e., when only concrete methods are introduced) can it be guaranteed that no unimplemented methods will be introduced. This is summarised by the following property:

If	R_{der} is a *complete* concretisation of R_{base} with M_{der}
and	R_{exch} is a *concrete* refinement or extension of R_{base}
and	R_{new} is a concretisation of R_{exch} with M_{der}
then	R_{new} is a *complete* concretisation of R_{exch} with M_{der}.

This implies that extensions and the extending parts of refinements can be performed freely, without having to worry much about possibly existing concretisations. Either the concretisations will still be complete or else it is easy to determine how the partial concretisations can be completed. In general, an extra concretisation needs to be performed on all abstract method descriptions added through the refinement or extension. This is a significant result as it means that users that only use an asset by concretising it, can easily switch to new versions of the system, as long as this new version is obtained by only making correct refinements and extensions of the old one. However, problems of method capture and inconsistent methods might still occur.

Note that the problem of unimplemented methods can also occur when, for example, M_{exch} is a cancellation and M_{der} is a refinement that introduces this cancelled method in its specialisation clause. The reuser depends on a method that has been removed. The only solution therefore is to incorporate the old implementation into the reuser's implementation.

5.3 Conflicts in the Calling Structure

Method Capture

While applicability of reuse modifiers can check method interface conflicts, for the detection of more subtle conflicts, such as method capture and inconsistent methods, the specialisation clauses of the modifiers must be taken into account.

A *method capture* occurs when a hook method m that is added by exchanging a base contract is also added or changed by a derived contract (Fig. 7). It is *accidental* when the method m did not yet occur in the original base contract.
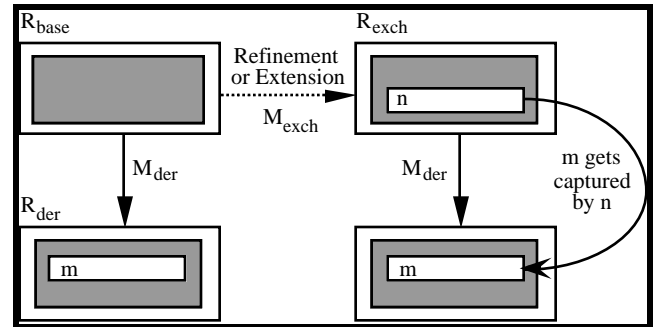
Fig. 7 Definition of Method Capture

Thus, accidental method capture only occurs when *both* reuse modifiers are extensions or refinements that introduce the method m in their extension part, thereby causing a name conflict.

This is illustrated in the following check for accidental method capture that adds an extra condition to the rule for name conflicts.

|	
If a method m is the cause of a name conflict when applying a reuse modifier M_{der} after M_{exch}	
and	m occurs in the specialisation clause of a method n in M_{exch}.
then	m gets accidentally captured by n

Regular method capture is more complicated to check as it does not introduce a method interface conflict (i.e., M_{der} remains applicable after base contract exchange). Reuse contracts in their current form only allow detection of regular method capture by comparing the specialisation clauses from the original base contract with the specialisation clause of the exchange modifier (to find newly added hook methods), and the interface of M_{der}. The detection of regular method capture thus happens directly on the level of specialisation clauses (and not on the level of applicability checks as is the case with the other rules) and by taking the original base contract into account. This can be amended by making explicit in the M_{exch} modifier which part of the

specialisation clause is new. We did not do so because we are not entirely convinced that regular method capture is indeed a conflict. Regular method capture can be entirely anticipated by the developer of the exchanged base contract.

In any case, the user should be notified of occurrences of accidental method capture. Sometimes these captured methods have the expected behaviour on the corresponding classes, so that it is still (behaviourally) correct to apply M_{der} to the adapted base contract R_{exch} although as another operation. When this is not the case, the problem can be solved by encapsulation techniques, as was discussed for implementation-specific methods in section 3.3.

Inconsistent Methods

While method capture occurs when augmenting the specialisation clauses in a base contract R_{base}, inconsistent methods are created when parts of the design are omitted by narrowing these specialisation clauses. Fig. 8 illustrates this.
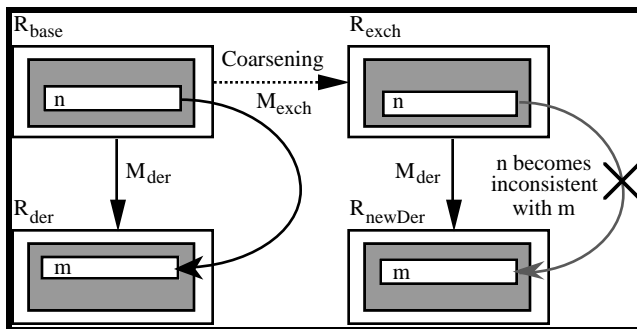
Fig. 8 Definition of Inconsistent Methods

Notice that inconsistent methods can only appear when the set of hook methods removed by exchanging the base contract and the set of names of method descriptions changed or added by the reuse modifier M_{der} are not disjoint. Therefore inconsistent methods can be detected as follows:

Assume that R_{base} is a base contract, and that R_{exch} and R_{der} are derived from R_{base} by applying modifiers M_{exch} and M_{der} respectively.

If M_{exch} is a coarsening where the coarsening clause associated to some method n contains m,

and m also occurs in the interface of M_{der}

then n becomes inconsistent with m

Indeed, inconsistent methods can only be introduced by coarsenings or cancellations, since these are the only operations that narrow (or delete) specialisation clauses. Cancellation however does not create inconsistencies, as the method description that omitted the reference from its specialisation clause simply does not exist anymore. Therefore only coarsenings can create inconsistent methods.

Notice that, similar to method capture, two forms of inconsistencies between methods exist, depending on whether the methods that have been removed from the specialisation clauses are also cancelled. In the latter case, an extra conflict will arise since the M_{der} modifier in Fig. 8 will not be applicable after base contract exchange.

Once inconsistencies are detected, the solution is straightforward. On base contract exchange, all method descriptions n that have become inconsistent due to a coarsening might need to be adapted by the derived reuse contract R_{newDer} as well. Whereas they used to rely on a method m to implement their behaviour, they do not anymore in the exchanged base contract. This might lead to inconsistent behaviour with respect to derived reuse contracts that depended on this information. Recall the example from section 2.4. After exchanging the class Set with a version of addAll that does not invoke add anymore, the subclass CountingSet needs to override addAll as well to avoid inconsistent behaviour.

5.4 Evaluation

The previous sections gave simple rules to detect problems when exchanging parent classes. Most of the possible conflicts are directly expressed in terms of reuse contracts and operators rather than on the level of interfaces and calling structures. This allows developers to reason about change in more intuitive terms and on a higher level than previously possible.

Because conflicts upon change can be easily detected, reuse contracts help to predict the work effort to update existing applications. Because they document what aspects possible reusers can rely on, they can also be used by developers of a reuse library to decide whether making a certain

change to the reuse library is a good idea or not. For example, when adding method descriptions to specialisation clauses, a distinction can be made between adding new method descriptions and adding already existing method descriptions. When adding already existing method descriptions to specialisation clauses, a developer of reusable assets knows that there is a great chance that method capture will occur. When these method descriptions are abstract this is even a certainty, since these method descriptions must be concretised by derived contracts. Therefore the developer might try to avoid doing this or at least pay extra attention that the captured method is only used for what it was originally defined for. In the same vein, the rules from the previous section can be used to guide application developers in understanding where testing is needed when the reusable asset has changed and how to fix the problems.

We have mainly explained how reuse contracts can be useful in the context of reusable asset evolution. Reuse contracts are also an important aid in making the layered structure of classes more explicit. As mentioned in section 2, the problems involved are similar to those in evolution of class hierarchies. Associating reuse contracts with classes in a class hierarchy helps in solving these problems by classifying different inheritors by the operators with which the associated reuse contracts are derived and pointing out possible conflicts when methods are overridden in the class hierarchy.

6   Future Work

Extensions to the model

Although we restricted ourselves to only three operators on reuse contracts, others are imaginable. For example, in the course of changing an abstract class a frequent operation is that of refactoring existing methods by introducing intermediate methods. The method `Update` in the reuse contract `Button` is such an example. In our approach this was implicitly achieved through refinement.

One could also think of predefining frequent combinations of operators on reuse contracts. An interesting example of such a combined operator on reuse contracts is turning an abstract hook method into a concrete method that invokes newly introduced abstract hook methods. This is a typical operation to add a more concrete layer to an abstract class. Currently, such an operator is a combination of a refinement and a concretisation.

A more serious extension of reuse contracts is including interclass relationships. In their current form, reuse contracts only document the internal dependencies among a class's methods. Part of our future work is studying how reuse contracts can be extended to include interclass dependencies as well, yet on a less behavioural level than contracts [3, 4].

Tools

When necessary, reuse contracts and the relationships between reuse contracts must be deduced. We already dispose of a prototype implementation of such a tool[3]. This tool also checks the correctness of explicitly declared relationships and signals possible problem situations as discussed throughout the text.

Such tools can also assist in the synchronisation of reuse contracts and their implementations. Two situations can be distinguished. In those parts of the reuse library that have a stable design, the implementation must be forced to comply to the reuse contract. In those parts that are still subject to major redesign, it should be possible to make changes to both implementation and reuse contracts independently. The environment could discretely issue warnings, but should not become a hindrance.

Analysis and Design Notations

Reuse contracts provide design information that is complementary to what conventional object-oriented design notations provide. We are actively investigating how reuse contracts can be integrated

---

[3] A prototype tool in Prolog can be found via the URL http://progwww.vub.ac.be/prog/pools/rcs/rc.html. We are also currently working on a "reuse contract"-extractor for Smalltalk classes.

with existing design notations and how the same principles could be applied to analysis and design specifications instead of classes.

7 Related Work

The work of Kiczales and Lamping forms the basis for this work by describing the problems involved in the specification of class libraries and by stressing the importance of internal dependencies [6, 7]. Lamping approaches this from a library specification angle and uses the interfaces primarily as documentation. Stata and Guttag extend the idea of specialisation interfaces to incorporate full behavioural specifications [13]. Ossher and Harrison discuss the combination of independently developed inheritance hierarchies [10]. They suggest a new way of system building, where systems are not adapted by subclassing or modifying code, but by combining existing hierarchies with merge operators. Their proposal can however only handle non-conflicting parallel extensions (two extensions are non-conflicting if the order in which they are combined has no importance). They emphasise the importance of the exploration of conditions different from non-conflicting that ensure that separate extensions "work correctly together" when merged. In their further work, they present composition rules to combine different independently developed "subjects" [11]. This work is more focused on the implementation level than on the design level.

Another way to handle evolving frameworks is refactoring [9] of class hierarchies. This work is, in our opinion, complementary to our work. Refactoring aims at transforming entire class hierarchies in order to make them more reusable, for example, by abstracting common behaviour into abstract classes. It is based on code analysis and transformation. Our work also starts from class hierarchies, but deals with how to manage design changes that propagate through these hierarchies.

8 Conclusions

Although recently important advances have been achieved in object-oriented software engineering, reusability still fails to fulfil its high expectations.

Two of the most important inhibitors to successful reuse are the lack of adequate documentation and the absence of mechanisms to manage the propagation of changes to reusable assets through applications that have been built on them.

Reuse contracts and reuse operators solve these problems by recording the protocol between producers and users of reusable assets. They not only document the design intentions of the asset producer, but also the assumptions made by reusers about the assets they reuse. When changes are made to assets, this documentation allows to identify which of these assumptions are no longer valid and thus where the applications built on the assets should no longer be trusted.

A detailed study of the reuse operators and their interactions led to rules that describe just that. These rules indicate to reusers where and how to test and adjust applications, when the assets on which they were built undergo changes. In a similar vein, they assist asset developers in assessing the impact of changes they make.

9 Acknowledgements

10 References

[1] Bracha, G. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*, PhD Thesis, Dept. of Computer Science, University of Utah, 1992.

[2] Goldberg, A., Rubin, K. *Succeeding with Objects: Decision Frameworks for Project*

Management, ISBN 0-201-62878-3, Addison-Wesley Publishing Company, 1995.

[3] Helm, R., Holland, I., Gangopadhyay, D. "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", *Proceedings of Joint ECOOP/OOPSLA '90 Conference,* pp. 169-180, ACM Press, 1990.

[4] Holland, I. *The Design and Representation of Object-Oriented Components*, PhD thesis, Northeastern University, 183 pages, 1992.

[5] Johnson, R., Foote, B. "Designing Reusable Classes", *Journal of Object-Oriented Programming*, 1(2): 22-35, 1988.

[6] Kiczales, G., Lamping, J. "Issues in the Design and Specification of Class Libraries", *Proceedings of OOPSLA '92, Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 435-451, ACM Press, 1992.

[7] Lamping, J. "Typing the Specialisation Interface", *Proceedings of OOPSLA '93, Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 201-215, ACM Press, 1993.

[8] Mens, K., Lucas, C., Steyaert, P. "ARC: an Algebra of Reuse Contracts", *Tech-report* ftp-able at: progftp.vub.ac.be/tech_report /1996/vub-prog-tr-96-03.ps.Z.

[9] Opdyke, W., Johnson, R. "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems", *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications* (SOOPPA), 1990.

[10] Ossher, H., Harrison, W. "Combination of Inheritance Hierarchies", *Proceedings of OOPSLA '92, Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 25-40, ACM Press, 1992.

[11] Ossher, H., Kaplan, M., Harrison, W., Katz, A., Kruskal, V. "Subject-Oriented Composition Rules", *Proceedings of OOPSLA '95, Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 235-250, ACM Press, 1995.

[12] Pancake, C. "Object Roundtable, The Promise and the Cost of Object Technology: A Five-Year Forecast", *Communications of the ACM*, October 1995, Vol 38(10), pp. 32-49, ACM Press, 1995.

[13] Stata, R., Guttag, J. "Modular Reasoning in the Presence of Subclassing", *Proceedings of OOPSLA '92, Conference on Object-Oriented Programming, Systems, Languages and Applications,* pp. 200-214, ACM Press, 1995.

[14] Van Limberghen, M., Mens, T. "Encapsulation and Composition as Orthogonal Operators on Mixins: A Solution to Multiple Inheritance Problems", *Object Oriented Systems Journal*, Volume 3, Number 1, Chapmann&Hall, 1996.

[15] Yourdon, E. *Object-Oriented System Design: An Integrated Approach*, Yourdon Press Computing Systems, Prentice Hall, 1994.