Chapter 1

Issues in Reuse and Composition

The goal of this dissertation is to set up a methodology that can fill a void in software engineering and can cross the bridge between current practices in different fields. As on the one hand, this dissertation addresses completely uncovered territory and on the other hand, it tries to combine the benefits of different approaches and is inspired by work in different fields, it is related to a lot of different issues.

This chapter gives an overview of these different issues and simultaneously situates our work. Before doing that section 1.1 discusses some of the conflicts that can occur during evolution and will be the focus of the remainder of this dissertation. Section 1.2 discusses some general issues in component systems, while sections 1.3 to 1.5 focus on the contributions of object-oriented software engineering (OOSE). At the end of each of these sections the relation of reuse contracts with the presented work is discussed. Section 1.6 then closes this chapter by summarising the problems and explaining how reuse contracts contribute to solving them.

1.1 Conflicts with Evolving Components

In the remainder of this section, we first give an example of the kind of conflicts that can occur with evolving components. Second, we give a more systematic overview of possible problems.

1.1.1 An Example

Packet Handling Consider the example of a software simulation of a local area network (LAN), where a packet is passed around a circular LAN. Consider two components `outputserver` and `packet`, as depicted in figure 1.1.

A packet is passed from node to node in the LAN. When an outputserver (which is a special kind of node) receives a new packet through the operation `accept`, it

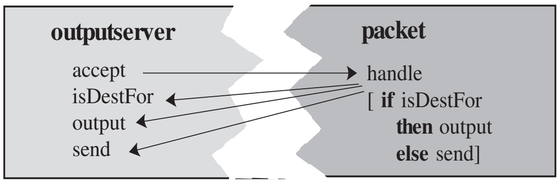

Figure 1.1: Packet Handling in a LAN

invokes the operation `handle` on the packet. The response of the packet is to invoke the operation `isDestFor` on the outputserver, to check whether this outputserver is the destination of this particular packet. If it is, it invokes the operation `output` on the outputserver to perform some action with the packet. If not, it invokes the operation `send` that passes the packet on to the next node.

The communication between these two components describes general packet handling behaviour. Other components, representing different kinds of outputservers and packets, can be created that comply with this contract and perform a particular behaviour by associating a particular implementation with e.g. `output` or `handle`. Special kinds of outputserver can be printservers, where `output` prints the contents of a packet, and fileservers, where `output` saves the contents of a packet. A special kind of packet can be broadcast packets that are to be handled by every node. After introducing multiple variations on `outputserver` and `packet`, it becomes difficult for a developer to assess how — or whether — all these adaptations work together.

Introducing gateways Now consider introducing `gateways` as a new kind of node, as depicted in figure 1.2. When the operation `isDestFor` is invoked on a gateway by a packet, the gateway checks the domain name. When this name is correct it returns yes. As a consequence the operation `output` will be invoked on `gateway` by the `packet`. The implementation of the operation `output` on `gateway` is adapted to send the packet through to the particular domain. Note that the ability to make this variation requires knowledge on the inter-participant communication.

Introducing visitor packets Now consider another adaptation of the packet handling behaviour: the introduction of `visitorpackets`, which have to be passed to all nodes in a LAN to perform a particular action they encapsulate (figure 1.3). An example of such an action could be counting the number of printers in the network.

To achieve this, `visitorpacket` re-implements `handle`, so that it performs an action and then always invokes the operation `send` on `outputserver` and is thus always passed on. In combination with outputservers as defined above this works

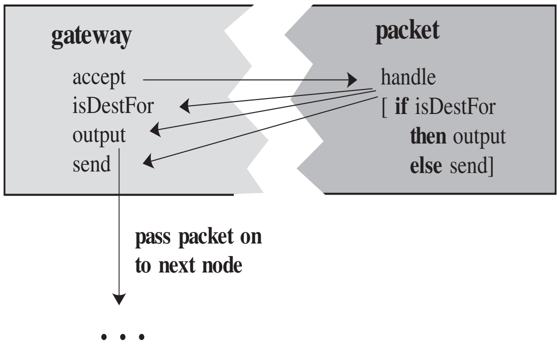

Figure 1.2: Introducing Gateways

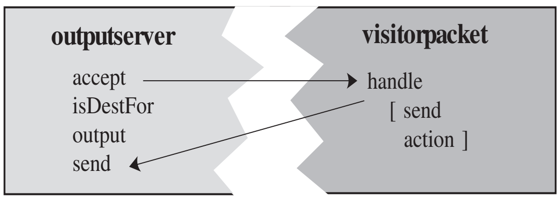

Figure 1.3: Introducing Visitor Packets

perfectly.

Combining gateways and visitor packets *Separately*, the two variations on the general contract — gateways and visitor packets — work perfectly with the other component in the packet. A developer might thus assume that everything will work as planned. Now consider introducing these two variations in one system. Using `visitorpacket` and `gateway` *together* causes a problem (see figure 1.4). `Gateway` uses the operation `output` to send a packet on to the right subnetwork. Since `visitorpackets` are always passed on by means of the operation `send`, `gateways` never receive the operation `output` from `visitorpackets`, and thus `visitorpackets` never get passed `gateways`.

When we investigate the two customisations closer, we see where the problem arises. `Gateway` re-implemented the operation `output`, relying on the *assumption* that packets always send this operation when the referring node is their destination. `Visitorpacket` on the other hand, no longer invokes this operation. It thus breaks the assumption `gateway` was relying on and the behaviour is no longer correct.

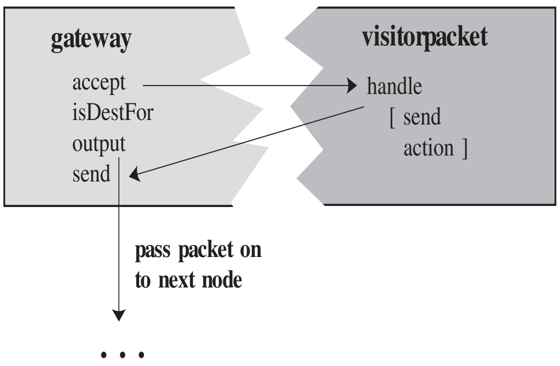

Figure 1.4: Combining Gateways and Visitorpackets

The cause of the problem is that when composing components, one makes assumptions about the way these components will react when certain operations are invoked on them. Here the implementor of gateway assumed that whenever the `isDestFor` operation gets invoked by a packet and returns true, the operation `output` will be invoked by that packet. This was a reasonable assumption, because until then, that was what all packets did. When visitorpackets were introduced however, this assumption was rendered void and so was the co-operation between the two components. This is depicted in figure 1.5. One could argue that the introduction of `visitorpackets` should not have been allowed, because they do not comply to the general contract. However, as we argue further on, a less coercive approach where certain deviations of the general contract are allowed is crucial in the use and development of reusable components.

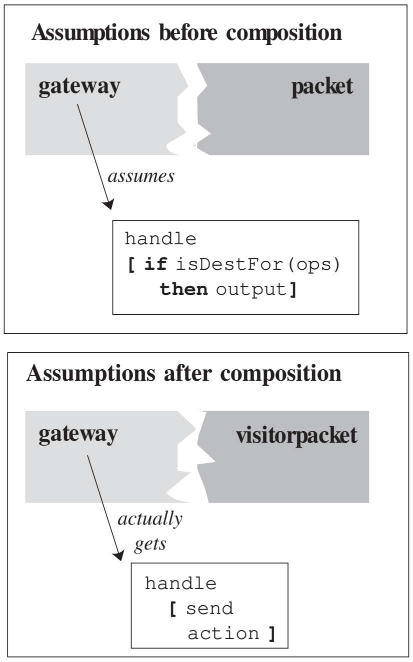

Figure 1.5: Broken Assumptions

Problems with evolving components While this example was presented as a description of what problems can arise when combining different components, the same problems could occur with the evolution of components. One could consider, for example, `outputserver` to be a first version of a component in some dedicated library and `gateway` an enhanced implementation of this component in the following version of the library. If an application developer had implemented visitor packets the way they were described above, problems would occur when upgrading this application to the next version of the library. To take it one step further, the same problem could arise with two adaptations that were independently developed by two developers (or even just one) on a large application, regardless of whether this application was built with components or not.

1.1.2 An Overview of Possible Conflicts

In general, these kinds of problems can occur when changes are made to two different parts of a system, regardless of whether this is achieved through composition, during evolution or by different developers. This section inventorises a number of problems that can occur. We consider two arbitrary modifications that do not cause problems when they are applied exclusively and we ask how these two modifications can be integrated.

Interface Conflicts

A first set of problems concerns conflicts in the interface, i.e., conflicts of operation names, component names, etc. It is possible that separately two modifications introduce an extra operation or component with the same name. It is possible that both modifications intended to model similar behaviour, but also that they aimed at something completely different. Therefore, such a conflict should be noted on integration, to allow the user to decide how to solve this conflict.

Figure 1.6 demonstrates such a conflict. We depict all conflicts we discuss in this section on an abstract level. A first version of two related components A and B[1] is depicted at the top of the figure. The arrows represent two modifications M_1 and M_2 that result in the two versions at the bottom of the picture. The grey areas in the pictures show where something has changed. In this first example both modifications introduce an operation m on component B. When integrating both modifications, problems might occur. Here the problem is that both modifiers have introduced an operation with the same name on the same component. We call this an interface conflict.

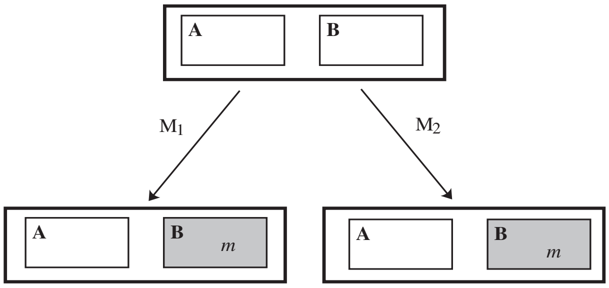

Figure 1.6: Example of an Interface Conflict

Interface conflicts can also occur with annotations made to parts of interfaces. It is not uncommon to attach extra information to items in an interface. In C++, for example, member functions can be abstract or concrete, virtual or not, public or private. We will also add extra information concerning crucial internal dependencies to the external interface. We will amongst other discuss which operations invoke

[1]The fact that they are related is denoted by the rectangle surrounding them. We take abstraction of how they are related.

which other operations. The notation $x[m]$ is used to denote the fact that x invokes m. An example is given in the next section.

Dangling Reference Conflicts

The inverse kind of interface conflicts can occur when one modification removes some item from its interface that another modification refers to. We call these conflicts dangling reference conflicts. This is depicted in figure 1.7, where one modification adds an invocation of m, while the other modification removes m.

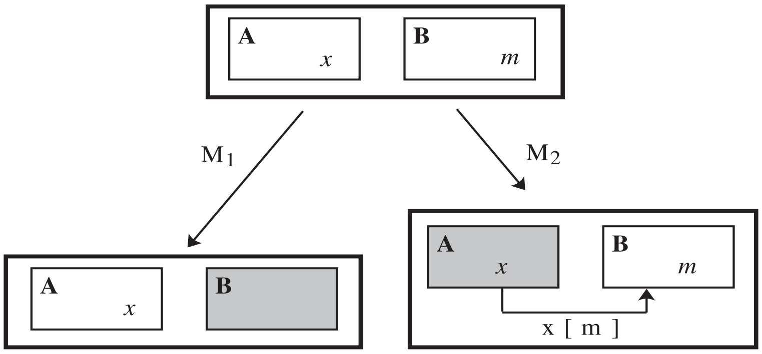

Figure 1.7: Dangling Reference Conflicts

Again, this kind of conflict can occur in different parts of the interface.

Conflicts Concerning the Calling Structure

While the above conflicts might seem simple and a lot of them can be detected by compilers of statically typed languages, other conflicts are more subtle. These conflicts, such as the inconsistent operations in the LAN example, do not necessarily make a system break down, but may result in a working system that does not exhibit the expected behaviour: it does not behave the way the developers *assumed* it would. These are conflicts that concern the assumptions made about the calling structure, i.e., which operations might invoke which other operations. These conflicts now remain undetected.

Operation Captures A first example is operation capture. One of the modifications can provide a new implementation for a certain operation, while the other modification includes extra invocations of the same operation. In that case we say that this operation gets captured by the invocations of the second modification. In figure 1.8 the new implementation of x gets captured by z. This may result in erroneous behaviour, as the first modification (re-implementing x) did not take into account that its operation would be invoked by the other operation (z).

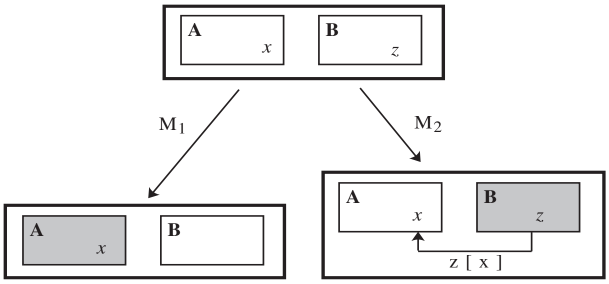

Figure 1.8: Regular Operation Capture

Two cases of operation capture can be distinguished. First, after a modification, an adapted version of an operation that already existed in the original system might be invoked by more operations than before. We call this *regular operation capture*, as it can be foreseen by modifications that any existing operation that they change could be invoked by operations they don't know of.

In the second case, an operation that did not exist in the original system is now introduced and invoked. Figure 1.9 shows how the two modifications have introduced an operation with the same name and M_2 also added an invocation of this operation. Depending on how the interface conflict is resolved[2], the operation of M_1 might be invoked by the operation of M_2. As this could not be foreseen, we call this *accidental operation capture*.

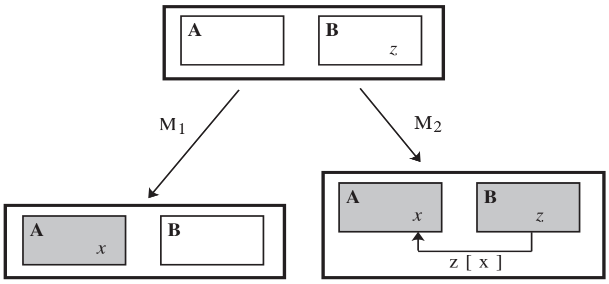

Figure 1.9: Accidental Operation Capture

[2]Note that in the case of accidental operation capture there will also be an interface conflict, because both modifications introduce an operation with the same name.

Inconsistent Operations When method invocations are omitted, the inverse situation of operation capture can arise. If during integration the new system executes less invocations of an operation than before, this might lead to inconsistent behaviour.

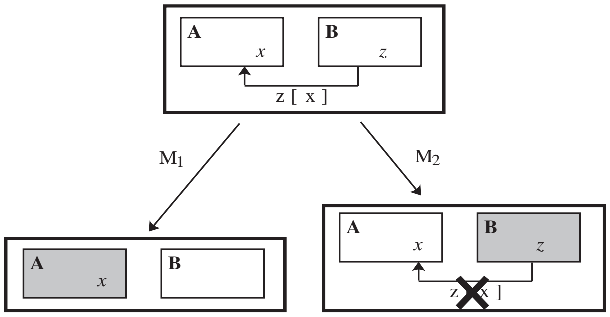

Figure 1.10: Inconsistent Operations

This situation is depicted in figure 1.10. The first modification has adapted the implementation of an operation x, while the second modification has omitted an invocation of this same operation. The first modifier might have assumed that the modification of the operation x would also have an influence on z. After integration with the second modification, this is no longer the case. We then say that the two operations x and z have become inconsistent. This terminology is due to Kiczales and Lamping [KL92]. Note that this is exactly what happened in the `Gateway` example.

Unanticipated Recursion Problems might also arise when inadvertently recursion is introduced. Consider again a component A with an operation x and a component B with an operation z. Now assume that the first modification introduces an invocation of the operation z by x and the second modification introduces an invocation of the operation x by z, as depicted in figure 1.11. This might introduce mutual recursion, which was not foreseen by either one of the adapters.

Other Possible Conflicts

The set of possible conflicts that we discussed here is by no means complete. First, depending on the kinds of components different conflicts occur. Second, depending on how much and which kind of information is provided in interfaces, more or fewer and different problems are detectable. We chose the problems discussed here, because they are relevant (see the two examples and see, for example, [KL92]) and because they are exemplar for the kinds of problems we want to tackle. Further on in the text, when relevant, other possible conflicts are mentioned.

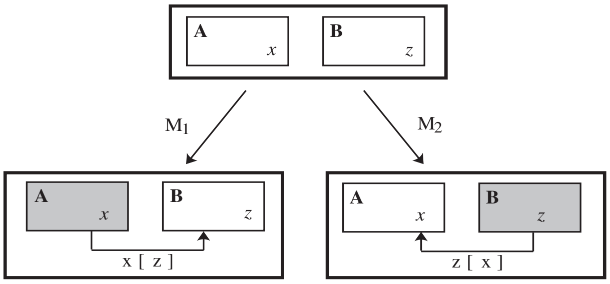

Figure 1.11: Unanticipated Recursion

1.2 The Evolution to Component Software

1.2.1 Benefits and Inhibitors

Since computers first appeared on stage, the increase in software productivity has by no means kept pace with the growing demands or with the evolution of hardware. Despite numerous research efforts concerning programming languages and methodologies, the development of quality software still seems to be an art as much as a science. Among others object-orientation has failed to deliver much of its promises.

Today's software environments react to the ever-increasing customer demands by adding feature after feature to their systems. This process results in large, monolithic applications that are difficult to customise for different users. Moreover, these systems are expensive to maintain and the interdependencies between all these features make it harder and harder to integrate new functionalities. The combination of these problems leads to an ever-rising development cost.

As a consequence and due to the inherent complexity that involves building software from scratch, the software engineering community has focused on the reuse of existing software systems and components. The possible benefits of component systems both on the business and technical side are abundant. [KY95] states, among others, the following:

- faster time to market, since the bulk of an application is already there;
- reduced expenses because applications do not duplicate existing functions and are not written from scratch;
- quick response to changing customer needs and competitive pressures;
- increased customisation, addressing emerging market segments;
- fine-grained integration beyond “Band Aid” interfaces;
- higher quality, reliable systems via pre-tested and pre-approved components;

- simpler testing and debugging, contributing to accuracy and ultimately to user satisfaction.

In order to achieve these goals a large number of questions still needs to be answered. A number of these questions refer to business issues. We again follow [KY95]: How much is a component worth? How does a company account for the development costs? How do vendors set prices? How do companies measure return on investment?

In addition to these business inhibitors — on a technical level — the lack of adequate tools and libraries, the absence of adequate standards, the lack of certification techniques and the focus on language and source code dependent solutions are often mentioned as the main problems. In this chapter we argue that the problems are deeper rooted.

1.2.2 Systematic Reuse: a Paradigm Shift

While a large consensus exists on the possible benefits of component systems it is equally recognised that the establishment of such a component industry, where components are bought and sold and systems are assembled rather than written, is a long way from reality.

According to W. Frakes [Fra94], software engineering is undergoing a paradigm shift, in order to achieve this goal. The new paradigm, called *systematic reuse*, is domain focused, based on repeatable processes, and concerned primarily with reuse of higher level life cycle artifacts such as requirements, designs, subsystems, etc. Systematic reuse shifts from *"viewing software engineering as a discipline concerned with the construction of single systems, to a discipline concerned primarily with constructing related systems that share common parts and vary in certain regular and identifiable ways"*. In other words, at the heart of this paradigm shift is an evolution from the current craftsmanship paradigm, aimed at constructing hand-crafted single systems, to that of an organised industry centred around the production of software components, aimed at building systems much like product lines in e.g. consumer electronics or car manufacturing. This kind of systems is called *component software*.

While the high expectations provoked by the possible benefits of component software can lead to discouragement when advances are not noticed as soon as expected, Frakes stresses that positive change is almost always slow and methodical and that changes usually come in small increments.

1.2.3 Kinds of Component Systems

The most logical approach to reuse is to store reusable components in a reuse library or reuse repository. In the early days, a strong emphasis was put on building reuse libraries of fragments of code. More recently, a shift in emphasis from reuse of code and behaviour to reuse of design is noticeable. [NT95] defines a component as a “static abstraction with plugs”. [NT95] explains: *By “static” we mean that a software component is a long-lived entity that can be stored in a software base, independently of the applications in which it has been used. By “abstraction”, we mean that a component puts a more or less opaque boundary around the software it encapsulates. “With plugs” means that there are well-defined ways to interact and communicate with the component (parameters, ports, messages,etc.)* An entire range of artifacts can possibly be reused: analysis and design models, process models, implementations, and so on. As implementation is only one phase in the software development process, it would not be wise only to consider reuse of code.

All these components, no matter what kind or form, need to be stored in a reuse repository, or *component system*. Component systems can be distinguished from application systems as they are more generic, reusable and customisable and generally require more effort to engineer. Component systems are used to engineer application systems, as well as other component systems. Furthermore, while application systems are delivered outside the business, component systems often are not.

Component systems can range from relatively independent components, to frameworks of collaborating components, to more complex and sophisticated component systems from which complete application systems can be generated. Depending on the kind of component systems they are derived from, different kinds of application systems can be observed. Jacobson et al. distinguish three of them [JGJ97]:

- An *application system suite* is a set of different application systems that are intended to work together to help some actors accomplish their work. An example of an application system suite is the Microsoft Office suite.

- *Application system variants* are used when essentially the same application systems need to be configured, packaged, and installed differently for different users. An example is the Ericsson AXE family of telecommunications switching systems. Sometimes, different variants can be achieved by simply installing and adapting application systems differently. At other times the variability is achieved by engineering different application systems from common components.

- Finally, sets of otherwise fairly independent application systems can be treated as members of a family by building them from the same sets of lower-level reusable components. An example are applications that use the Microsoft Foundation classes.

Unless explicitly mentioned otherwise, when we refer to component systems in this chapter we imply all of these variants, from the simplest single components to full-fledged application systems.

1.2.4 The Development of Reusable Components

When the focus of software engineering is shifted from building single systems to building families of systems, this obviously also influences the life cycle models. The emergence of reusable components is not an automatic side effect of system development. For reuse to be systematic, it needs to be pre-planned and the 'discovery' of reusable components needs to be a major concern in the development process. Reusable components need to be looked for, abstracted and certified. Next to the functional requirements, describing what the system must do, on what platforms and in what order of speed it must do it, non-functional requirements must be considered in order to achieve systematic reuse. These include writing systems that are highly adaptable, contain clearly reusable subcomponents, and are easy to maintain.

Iterative Development

It is generally acknowledged that the classical *waterfall model* does not cope well with these extra requirements. In the waterfall model, tasks are laid out and executed sequentially. The waterfall approach is based on the assumption that everything is known up front, so that well-defined tasks can be executed one at a time. This assumption is often erroneous, as we rarely know all the requirements; even if we could know all the requirements, they would undoubtedly change as the project progresses.

Evolution of living organisms is often used as an analogy with the evolution of software. Just as a living organism adapts to its changing environment, software must adapt to ever-changing requirements. This is true during development, when the parts to be integrated and tested change from day to day, as well as during the rest of the system's life cycle. Therefore a number of *evolutionary models* have been proposed[3]. While each model has its own characteristics, in general evolutionary models contain the same basic steps (analysis, design, integration, ...) as the waterfall model. What's different is how the steps interrelate and how each step is accomplished.

An example of an iterative approach is incremental development. Developers may do some requirements analysis, some design and some coding on a core part of an application and then go on to repeat the same steps on other parts, gradually integrating the different parts. Of course, integration of new layers with the core can call for changes at this core, ... This approach recognises that changes can be caused by decisions at different levels in the process.

There is not only a need for evolution because requirements are constantly evolving and because the best choice for modelling a system might not always be readily apparent. Experience shows that reusable components are generally not designed

[3]Examples are the spiral model, the whirlpool model, the fountain model, the baseball model, ... See, for example, [Wil96] for the whirlpool model.

from scratch, but 'discovered' through an iterative process of testing and improvement [JF88], [Cas94]. While some of the basic identities of components may be discovered early on in the development, these identities will change and improve as they evolve. Reusable components emerge as a result of an evolutionary process. Moreover, the usefulness and thus the true degree of reusability of components can only be tested by actually reusing them.

Domain Engineering

One way to limit the number of required iterations and one of the most successful topics in reuse research is domain analysis [PDe91]. The basic premise is that reuse of components is much easier if not only reusable components are stored in a repository, but also interconnections between the components that provide an architecture or template of how to fit the components together. Specialisation guidance can also be included.

In order for the component developer to be able to add this kind of information, he needs to analyse the problem domain. Domain analysis differs from classical systems analysis in that an entire family of systems and solutions must be considered. [Nei94] summarises domain analysis as follows:

1. Analyse four or more existing systems in the domain using classical systems analysis;

2. Form an analysis model of the union of features of existing systems;

3. Determine and, or, not, existential and universal constraints on acceptable domain feature variations;

4. Present model and variation constraints to domain experts for approval;

5. Continue 1 through 4 until all systems in 1 are variants of model 2 under constraints 3 and approval 4;

6. Provide function implementations under allowed variations.

The result of domain analysis can lead to artifacts as diverse as domain languages, domain algebra's, domain-specific software-architectures or domain-specific kits.

The topic is much more complex than presented here, but a full review is outside the scope of this dissertation (see, for example, [PDe91]).

1.2.5 Relationship with Reuse Contracts

While we acknowledge the contributions of domain engineering, we firmly believe that an evolutionary model for software engineering remains crucial. Reuse contracts aim to form the basis of a new methodology that considers iteration and thus evolution of components as the most crucial action in each part of the life-cycle.

In a later stage the information provided by reuse contracts might also be put to use to ameliorate the reusability and adaptability of reusable components, but that is outside the scope of this work.

1.3 Object-Oriented Reuse

This section examines the contributions of object-oriented software engineering. Object-oriented programming languages are often put forward as promoting software reuse. While features as polymorphism and inheritance do assist in enabling reuse, on their own they are not sufficient. We first look at how these features enable reuse and then discuss where they fail, what possible solutions have been proposed and where these solutions still fall short.

1.3.1 Polymorphism, Protocols and Inheritance

In object-oriented languages operations are performed by sending messages. Message sending causes *polymorphism*, as messages with the same name can be sent to instantiations of different classes and a different method is invoked, depending on the receiver's class. This is called late binding. As a consequence methods and programs can be written for a set of related classes, of which instantiations can be interchanged. A *protocol* is a set of messages that can be sent to an object. Objects with an identical protocol are interchangeable. Moreover, the creation of standard protocols makes it easier to understand programs, as methods with the same name encode similar behaviour. In languages with no polymorphism giving different procedures the same name is discouraged, since they cannot be used in the same program. As a consequence, sets of procedures with the same behaviour and slightly different names occur, which can be confusing. Another feature of object-oriented programming languages that enhances reuse is *inheritance*. Inheritance enhances code reuse by promoting programming-by-difference, where a programmer defines a new class by picking a closely related class and describing the differences between the old and the new class. One of the important benefits of inheritance is that all the subclasses of a class inherit its protocol, creating a family of classes that can be used interchangeably to some level. Furthermore, as these classes share a standard protocol it is easier to understand their behaviour. Another important benefit is that inheritance leaves intact the original code of the class it extends. Thus, changes made by one programmer are less likely to affect another. The code in the subclass defines the differences between classes, acting as a history of the editing operations.

1.3.2 Abstract Classes and Template Methods

While the discussed features of object-oriented languages are beneficial in order to achieve better reusable systems, on their own they do not suffice to create highly reusable code. In order to obtain highly reusable classes, one must design *for* reuse [JF88]. This is, as mentioned above, usually an iterative process resulting in reusable protocols that are represented by abstract classes.

An abstract class is a class that contains a number of abstract methods, i.e., methods without an implementation. Template methods are methods that rely on abstract methods or other template methods in their implementation. Finally, there are concrete methods, which have a full implementation that does not rely on abstract methods. Template methods are the key to reuse. They usually describe the core behaviour of the class. Subclasses inherit the core behaviour by means of the template methods. They only need to override the abstract methods to give them an implementation, or some default methods to adapt their behaviour. Besides implementing a reusable behaviour through template methods, abstract classes provide an interface with which all subclasses must comply. By having these common interfaces, the different subclasses are interchangeable[4][5].

The development of abstract classes is very important, but not easy. There is always a trade-off between flexibility and ease of use. A powerful template method which is only based on a small number of abstract methods requires minimal effort for adaptation. However, if these abstract methods are not sufficient to modify the template method's behaviour, the class is too inflexible and the whole template method needs to be overridden. It is almost impossible to get template methods in an abstract class right from the start and a number of iterations is usually required to improve their reusability.

Abstract classes can encode the design of a set of classes and thus embody domain knowledge by making the commonalities between their subclasses explicit. While with concrete class libraries developers only reuse implementations, with abstract classes they can also reuse designs. Another big difference between reusing concrete and abstract classes is the fact that concrete classes are instantiated by the application and messages are sent to these instance by the application. With abstract classes this is the other way round. Methods from the application are also invoked by methods from the abstract classes. The main domain knowledge is in the abstract classes, not in the application. Frameworks are an elaboration on this principle.

1.3.3 Frameworks

While abstract classes and class libraries can substantially increase developers' productivity, over the last few years it has become clear that to realise more substantial gains one has to go one step further and start reusing multi-class components and

[4]This means interchangeable in the technical sense that no "message not understood"-errors will occur. There are no guarantees concerning behavioural correctness.

[5]In order to be fully interchangeable some requirements concerning argument and return types of overridden methods need to be fulfilled. These requirements differ depending on the language's type system. This is however outside the scope of this thesis.

frameworks. This evolution is closely related to the work on domain engineering. [CP95] gives the following definition of a framework:

> A *framework* is a collection of co-operating classes that make up a reusable design solution for a given problem domain.

Abstract methods and the template methods that rely on them need not necessarily reside in the same class. A framework consists of a set of related abstract classes, with the interfaces between these classes defined in terms of sets of messages. Again, abstract methods need to be overridden to customise the generic design, only now these methods reside in different classes. A framework is not necessarily a working application. All abstract classes must be "filled in" with concrete ones first. Often a library of concrete subclasses is provided that can be used as components in the design.

[CP95] mentions three main differences between frameworks and class libraries :

- *Behaviour versus protocol:* While class libraries essentially provide sets of behaviours that programmers can use in their programs, frameworks also include protocols that describe how these behaviours can or should be combined.
- *Don't call us, we'll call you:* While with class libraries it is always the programmer who instantiates objects and calls their methods, with frameworks this responsibility is divided and it is also possible for framework code to call methods that were provided by the programmer.
- *Implementation versus design:* While with class libraries developers only reuse implementations, with frameworks they also reuse designs.

While the introduction of abstract classes already symbolised a shift from reuse of code to reuse of design, frameworks go one step further by stressing the importance for the design of co-operations between different classes.

Black-box Frameworks versus White-Box Frameworks

Another important feature that frameworks can be characterised with is the distinction between white-box frameworks and black-box frameworks [JF88]. This distinction can be made based on the the way a framework is customised. An application built on a framework typically adds methods to subclasses of the framework's classes. In doing this it must abide by the internal conventions of its superclasses. This is called *white-box* framework reuse because the framework's design and implementation must be understood to use them. The disadvantages of white-box frameworks are that every application requires many new subclasses and that learning to use such a framework can be hard, because learning it is the same as learning how it is constructed.

The alternative is to supply a framework with a set of components that provide application-specific behaviour. The interface between components can be defined by protocol, so the user needs to understand only the external interface of the components. This kind of framework is called a *black-box* framework. Black-box frameworks have the advantage that they are easier to learn to use, but the disadvantage that they are less flexible with respect to unforeseen changes. Mostly the available components provide the most basic and therefore anticipated variations, but users that want to do something slightly different can run into a lot of difficulties.

Furthermore, designing black-box frameworks is much harder than designing white-box frameworks, as a good black-box framework must anticipate all variations users might desire and provide components for these. In white-box frameworks, users that find the behaviour they desire not well anticipated can override methods from the framework's superclasses to adapt this behaviour. This is not the desired approach, but preventing users from wanting to implement unanticipated variations is not realistic. In fact, as a system is iterated over and the design becomes better understood, white-box relationships can be replaced by black-box ones.

1.3.4 Object-Oriented Methodologies

Numerous object-oriented analysis and design methodologies have been developed (see, amongst others, [Boo93], [CY91], [RBP$^+$91], [CAB$^+$94], [BRJ97]). Each of them has a notation for class diagrams. Most methods also have a notation to model object interactions. They have different names, but model similar concepts. There are object diagrams in Booch [Boo93], message connections in OOA/OOD [CY91], data-flow diagrams in OMT [RBP$^+$91], collaboration graphs in CRC [WBW89], object interaction graphs in Fusion [CAB$^+$94] and collaboration diagrams in UML [BRJ97]. All these methods basically define graph structures that represent execution threads between models. However, the main focus of the methodologies is usually on the development of the static model, not of the object interactions.

Two notable exceptions to this situation are CRC [WBW89] and OBA [RG92]. In [WBW89] an approach to object-oriented design is described that focuses on *classes*, class *responsibilities* and *collaborations* among classes (CRC). In an initial exploratory phase, classes, responsibilities and collaborations are defined by means of so called CRC cards. In the subsequent analysis phase, systems are defined, collaborations simplified, hierarchies built containing abstract and concrete classes, and object protocols are defined.

In [RG92] Object Behaviour Analysis (OBA) is presented as an alternative way of performing systems analysis. Upon completion the goal is to have a clear understanding of the behaviours exhibited by the system, the objects that exhibit these behaviours, the relationships among the objects, and how objects interact with one another. They claim that what is important to understand in analysis is the nature of the required information processing, in terms of the services to be performed. This is what should be focused on first, not the search for tangible objects. OBA starts with describing scenarios and deriving scripts from them; only in a later phase, starting from the scripts, objects are derived and classified and relationships between them are identified.

Note that both these approaches are mainly targeted for initial task analysis and capturing of requirements. They both stress the importance of collaborations and both emphasise the need for iterating over the system design before it is built.

1.3.5 Language Extensions

One of the problems with object-oriented software systems is that as methods need to be attached to one class, the application logic is spread over different classes and is often hard to detect and understand. While frameworks stem from the observation that inter-class relationships are crucial to design reuse, as they are usually expressed in classical OOPLs this behaviour is still spread over numerous classes. Several authors have therefore suggested different extensions to the classical object-oriented model, where these object interactions are made explicit.

Interaction contracts exist in two versions. In the first version they were declarative and could thus be used as structured documentation [HHG90]. This version is discussed in section 1.4.6. In the second version contract language constructs can be added to object-oriented programming languages [Hol92]. They thus change the syntactic representation of class implementations as well as the dynamic behaviour of objects. A contract is a 'package of object declarations and object behaviour definitions enclosed by a scope of visibility'. Contracts provide a module-like construct to structure and organise object-oriented programs at a higher level of abstraction, by coupling groups of related class implementations. The syntactic description of class implementations may be split up and distributed over multiple contracts. Contracts thus provide multiple interfaces for one object and also allow multiple implementations for one message on one object. The appropriate implementation is selected dynamically. The work on interaction patterns was performed as part of the Demeter project, of which the main result is adaptive programming.

Adaptive programming [Lie95] is an extension to object-oriented programming that attempts to focus on classes and methods that are essential not only to a particular application but also potentially to a family of related applications. Adaptive object-oriented programs specify essential classes and methods by constraining the configuration of a class structure that attempts to customise the adaptive program, without spelling out all the details of such a class structure. Adaptive programs are specified using propagation patterns. These propagation patterns are general descriptions of how a certain part of the behaviour of a program propagates over a generic class graph. Full-fledged programs are then obtained by customising the propagation patterns with concrete classes that comply with the desired structures. This way the interactions described by means of propagation patterns can be used on several concrete ensembles of classes.

Both interaction contracts and adaptive programming are based on the *Law of*

Demeter [LH89]. The Law of Demeter essentially says that when writing a method, one should not hardwire the details of a class structure into that method. Propagation patterns take this idea further and try to keep class structure details out of entire programs as much as possible. [Hol92] states 'While the Law is geared towards removing excessive behavioural dependencies, the Contract work focuses on the problem of understanding the important behavioural dependencies which result from object interactions'.

[AWB^{+}93] discusses how *composition filters* can be used to abstract object interactions. Composition filters are a mechanism that makes each message that arrives at an object (or is sent from an object) subject to evaluation and manipulation by the composition filters of that object. A single filter specifies a particular manipulation of methods. Various filter types are available. Filters can, for example, be concerned with distribution aspects or error handling. The filters together compose the behaviour of the object. Composition filters can be attached to objects expressed in different languages as C++ [Gla95] or Smalltalk [vDM95] and therefore allow extensions of programs written in different languages.

This is related to other approaches that make interaction protocols between components explicit as separate entities next to the basic objects. Examples of such approaches are gluons [Pin95] and connectors [DR97]. [Dem96] discusses how better tailorable systems can be created by making representations of framework contracts explicit as part of meta-object protocols. Meta-object protocols [KdRB91] can be used to expose design decisions in a controlled way, thereby allowing the behaviour of a system to be customised. [SW96] discusses how using meta-object protocols to implement non-functional requirements allows for a clean separation of concerns while still providing the necessary flexibility for application programmers to customise the system to meet the needs of their particular application.

1.3.6 Relationship with Reuse Contracts

While the reuse contracts approach can be applied to a range of different models and different kinds of systems, in this dissertation we mainly focus on object-oriented designs. Object-oriented systems offer a lot of power to build flexible systems. As the observation that building reusable components requires iterations calls for white-box approaches, it is clear that the need for more disciplined reuse is big.

We do not intend to turn reuse contracts into a new language mechanism but rather to advance engineering techniques for existing languages. The many proposals for language extensions, however, indicate the need for more focus on object interactions. Therefore, reuse contracts will make object interactions that are now hidden in different objects explicit.

1.4 Documenting Reusable Components

A very difficult question to answer when designing reusable components is how much of the internal structure of components should be made public and how much should be hidden. Giving a lot of information makes it easier to specialise, but restricts the range of possible specialisations, because a lot of variations are ruled out up front. Giving less information allows for more flexibility, but requires much more work to specialise. As a possible solution *layering* of protocols is proposed [KL92]. A lot has been said about the design of reusable components, mostly building on the use of template methods and frameworks [Pre94], [JF88]. Although we return to this issue at several points in this dissertation, our approach does not give a methodology that describes how to get a design right. Much will always depend on the situation and experience will remain a crucial factor in the design of reusable components. We do however focus on the kind of interfaces that are necessary to ease the tension between flexibility and extendibility.

1.4.1 Specialisation Interfaces

As was already discussed in the section on frameworks, the distinction between white-box and black-box components plays a crucial role in this trade-off between flexibility and extendibility. Current black-box component interfaces, as, e.g. CORBA IDL [Sie96], are designed towards developers using these components, not towards developers creating specific versions of a component to better fit their needs.

The main problem there is that it is not only the reuser that invokes operations on the reused components, the reused component can also invoke operations that were provided by the reuser. Consider adapting a class to particular needs by inheriting from it. When an inheritor overrides a method in a subclass it is possible that this method will be invoked by methods of the superclass, the structure of which the inheritor might have no understanding. Similarly, in component systems, when a developer inserts a dedicated component in a larger system, this component will not only invoke operations of other components, these other components will also invoke operations of the user's component. This can occur in object-oriented systems through polymorphism and message passing, but also in other paradigms by means of so-called call-backs, or through event propagation mechanisms as in, for example, JavaBeans [Sun97].

This problem was — to our knowledge — first approached by John Lamping's proposal to introduce *specialisation interfaces* [Lam93]. In this paper he discusses how inheritors of object-oriented class libraries need more information on the structure of the class than other clients. More specifically, they need information on which methods invoke which other methods in their implementation in order to assess which methods can be inherited, which methods need to be overridden and how. This idea has been followed by many others. Ivar Jacobson, for example, suggests looking at the protected operations of abstract classes as a separate interface

[JGJ97]. He stresses to be aware of the fact that these interfaces need to be designed as thoroughly as public interfaces. In JavaBeans [Sun97], *design-time interfaces* are distinguished from *run-time interfaces*.

This observation is not only valid for object-oriented programming. Subclasses provide different specialisations of their superclass and a subclass can be used wherever a superclass is expected. In general, the same problem is present in any approach where different versions of one component are desirable that are interchangeable and interact with their environment. This is exactly the goal of component systems.

1.4.2 Contractual Interfaces

However, the interfaces of single components are not the only problem. What is even more crucial is an interface, and an understanding, of how different components interact with each other in a system. One of the general conclusions of the ECOOP '96 Workshop on Component-Oriented Programming was that *interfaces need to be contractual and as such a binding between any pair of provider and client.* [OB97] suggests that in addition to an interface describing the services it provides to its users, a component should also have a *required interface*, describing what services it expects from components it interacts with. [MSW97] agrees that the current way to describe components by providing their interfaces and some additional informal documentation is inadequate. Because they feel that providing full semantic descriptions is not appropriate for all kinds of components, they suggest the introduction of an extra level of information: the Originator Level, between the Interface Level and the Semantic Level. This Originator Level should then provide information to help address the question of which types and versions of components work together, i.e., information on the context in which the component was developed, including which versions of other components it requires.

1.4.3 Documenting Frameworks

Let us now see how these documentation requirements are met with respect to object-oriented frameworks. According to [Joh92] the documentation of a framework has three purposes. Framework documentation must describe

- the purpose of the framework;
- how to use the framework;
- the detailed design of the framework.

The framework user must become familiar with its design, that is, the design of its individual classes, as well as the interactions between these classes. As was discussed in the section on white-box and black-box frameworks, there is a general understanding that a framework should be used predominantly in a pre-defined way: the basic framework structures should not be violated. The rules that describe how to do so remain, however, largely informal documentation. [CP95] describes the documentation of the Taligent frameworks as follows:

> Taligent provides new CommonPoint developers with a series of developer guides that cover frameworks and services and include code examples. Taligent also provides a programming tutorial, sample code, documented sample applications, human interface guidelines, white papers, on-line documentation and a variety of resources.

Some more structured form of documentation that is used concerns cookbooks and design patterns.

1.4.4 Cookbooks

The Model-View-Controller construct is often cited as the first mini-framework and this framework was already documented by a "Cookbook for Model-View-Controller" [KP88].

Cookbooks contain recipes that guide reusers step by step in using a framework for application development. They describe in an informal way how to use a framework and usually do not explain the internal design and implementation details of the framework.

Besides being informal, recipes also suffer from the problem that they are overly coercive, i.e., they describe the typical ways that the framework will be used and can thus only guide reuses that were specified up front. A good framework can, however, be used in ways that its designers never conceived. The MVC cookbook therefore also included an informal description of the design of the framework, in addition to the instructions on how to use it.

1.4.5 Design Patterns

One way to additionally document frameworks is through design patterns [GHJV94], [Pre94]. [GHJV94] defines design patterns as

> Descriptions of communicating objects and classes that are customised to solve a general design problem in a particular context.

In general, design patterns try to record experience in designing object-oriented software in a form such that people can effectively reuse it. Their first aim is to make it easier to reuse successful architectures and designs, make these designs more accessible to developers of new systems and help in choosing between design alternatives. For that purpose catalogues of design patterns are put together [GHJV94], [CS95], [VCK96]. In addition to that, design patterns can also be used to document existing systems. For example, Ralph Johnson documented HotDraw by means of design patterns [Joh92] and Wolfgang Pree documented ET++ by means of his meta-patterns [Pre94].

A design pattern, in general, has four essential elements

- a name;
- a problem description;
- a description of the solution;
- a discussion of the consequences.

These four elements are all described through informal textual notations, graphic notations (as OMT, UML,...) and programming language samples. Design patterns suffer from the same lack of formal underpinnings as cookbooks, code examples, sample applications, etc.

While most catalogued design patterns describe domain or problem-specific designs, *meta-patterns* [Pre94] provide a more advanced abstraction, by categorising patterns according to possible class/object composition attributes as the multiplicity of associations or whether two classes in a design are descendants or not. This provides a more systematic approach, but still lacks a formal model behind it.

1.4.6 Interaction Contracts

Interaction contracts were not presented specifically in the context of frameworks, but more generally as a program tool for object-oriented programming. As a program extension, they were discussed in section 1.3.5. In their declarative version they can be used as structured documentation [HHG90].

Interaction contracts then describe object interactions, i.e., a group of objects which interact via message passing to accomplish some system task or cooperate some system invariant. Interaction contracts aim at providing an explicit formal textual representation of object interactions. An interaction contract contains:

1. a *name*;
2. a *refinement clause* stating from what other contracts this one is a specialisation;
3. a *participant list*;
4. *obligation definitions* describing what each participant contributes to the contract. These can be instance variables, methods or method interfaces;
5. *include clauses*. Contracts can include other contracts;
6. *invariant definitions*, defining predicates which must be maintained by the participants.

Conformance declarations then need to be provided that specify the relation between class implementations and participant definitions.

While interaction contracts were the first approach stressing the importance of interclass co-operation in documentation of reusable object-oriented systems and were thus instrumental in the current evolution, they never found wide-spread use. We feel that the main reason for this is their high level of formality, which tends to scare of developers. While the ideas were important, we think that in order to be widely used a more intuitive notation is necessary.

1.4.7 Interface Definition Languages

More recently, interface definition languages (IDLs) were introduced as part of CORBA's (Common Object Request Broker Architecture) effort to achieve language and platform-independent composition [OH97], [Sie96]. The CORBA architecture separates the interface, written in IDL, from the implementation, which must be written in some programming language. IDLs provide language neutral and totally declarative descriptions of components. IDLs do not define implementation details and provide operating system and programming language independent interfaces to all services and components that reside in a CORBA repository. An interface definition specifies the operations the object is prepared to perform, the input and output parameters they require, and any exceptions that may be generated along the way. Clients of the object then use the same interface definition to build and dispatch invocations.

1.4.8 Architecture Description Languages

Architecture description languages [GS93], [KC94b], [GS96] are used to describe high-level structures or designs of software applications. The focus is on software modules and the interactions and communications between them, rather than on the functional or algorithmic detail within a particular source module. A software architecture is usually defined as the organisational structure of a software system including components, connections, constraints, and rationale [GS93]. Components can be small pieces of code (such as modules) or larger chunks (such as stand-alone programs). Connections are abstractions of how components interact in a system (e.g. procedure calls, pipes, remote procedure calls). An architecture has various constraints and rationales associated with it, including the constraints on component selection and the rationale for choosing a specific component in a given situation.

Recently a lot of work has been carried out in this area. Researchers have identified different architectural styles found in existing software systems [GS93]. Others have classified different features in existing ADLs in order to choose a good ADL for a given situation and as a basis for developing improved and specialised ADLs [KC94a]. Other emphasis was put on developing domain-specific software architectures (DSSAs), i.e., architectures for a family of application systems in a domain (see for example [LRD+88]).

1.4.9 Relationship with Reuse Contracts

Recent literature has clearly stressed the need for documentation both on how a component is structured internally and on how components interact in a system. This documentation can describe systems at different levels of abstraction, e.g. specialisation interfaces versus ADLs, and at different levels of formality, e.g. cookbooks versus interaction contracts. In this dissertation we will focus on lower-level descriptions as ensembles of classes. The application of this work on higher-level artifacts as architecture descriptions is a next step. Reuse contracts will have a formal basis, but will only include information of which compliance with the code can be automatically checked. This means that no invariants as in, for example, interaction contracts will be included.

Chapter 8 will briefly discuss possible extensions to the model in this dissertation, both in adding more information and describing other and higher-level constructs.

1.5 Evolution of Reusable Components

In addition to the technical motivations for an iterative approach to the development of reusable components, there are also some purely economic reasons. As the discovery and abstraction of reusable components has to be actively pursued, this requires a substantial investment. Fully planning all reuse requires a very large up-front investment. Therefore, a more gradual approach where reusable components are a concern at several steps of the development process, is more realistic. A delicate balance between longer term investments in reusable components and the need to meet deadlines must be found. To be properly reusable, components should undergo some form of certification. To be able to leverage on the investment, reusers must be able to benefit from future improvements of the components they reuse: proper evolution of reused components should not invalidate previous reuse. If reusers do not upgrade their applications to newer versions a proliferation of versions occurs. While reuse of components should lead to a reduction of the maintenance effort, the presence of all these different versions makes maintenance only harder.

To inhibit proliferation of versions it should be made easier for reusers to update their software to new versions of the components they reuse. This implies the management of some kind of consistency in the evolution of reusable software. To guarantee the systematic detection of problems a formal understanding of change propagation is essential. Programming environments should provide assistance in updating applications to new versions of reusable components, based on a formal understanding of change propagation. The absence of such management mechanisms is recognised as an important inhibitor to successful reuse [GR95], [Pan95], [You94].

We first give a short overview of work that has been done in this area and then discuss what kind of conflicts on change propagation we try to tackle with reuse contracts.

1.5.1 Binary Compatibility

One related topic is release-to-release *binary* compatibility. The major concern there is that when the evolution of the class library does not require changes to the application source, the application source should not require recompilation. The de facto standard way to do this was developed by the designers of SOM and is based on a number of transformations that are guaranteed to be compatibility preserving 'for all reasonable functional specifications' [FCDR95]. Transformations were set up for procedural as well as object-oriented programming that are behaviour preserving in this way. Object-oriented programming requires a more extensive set of transformations, because of the possibility for OO applications to subclass the classes of the library. Our work focuses on changes that are outside these transformations.

1.5.2 Refactoring and Restructuring

A lot of work in the OOSE community has been carried out on refactoring (see for example, [Opd92], [Cas94]). Refactorings are automatic program restructuring operations that preserve the behaviour of a program. These refactorings aim to assist in subsequent design iterations by helping to extract reusable components and improve the consistency among components. Examples of low-level refactorings are moving instance variables and methods from one class to another and renaming an instance variable. Higher-level refactorings try to find and create new abstract classes or capture aggregations.

A related approach is adapted in the Famous (Fully Automated Method and Object Update System) framework [HS95]. This framework couples a change avoidance approach based on Adaptive Software (see section 1.3.5) with a change management mechanism for automatic evolution. They see the foundation of an object-oriented system to be its schema, i.e., the structure of classes and their interfaces, and aim at automatically maintaining consistency and behaviour of a system during schema evolution. Their approach is based on a number of basic, behaviour-preserving class graph transformations, such as the addition of a concrete class or abstraction of a common reference.

These refactoring techniques can be helpful in, for example, assessing the design of a system that has gradually drifted from its original architecture and make suggestions about and assist in enhancing it. On the other hand, the suggested changes are not always true clarifications of the design. Not every method that occurs on two sibling classes should necessarily be pulled up. Furthermore, these refactorings do not generally assist a developer when making more substantial changes to its reusable components. They only assist in making predefined, frequently appearing changes.

More recently, the FAMOOS project [DDN$^+$97] has presented the need for restructuring on different levels. On a lower level, restructuring operations like class refactorings are proposed. On a higher level, the use of anti patterns [Koe95] to detect problems and the application of design patterns to solve them is proposed. At the time of writing this work the FAMOOS project is still in an early stage and its approach has not yet been tested on large case studies.

1.5.3 Programming by Contract and Formal Methods

Another approach to correctness of programs is the use of formal methods as Z [Spi89], VDM [Jon90] or Larch/C++ [Lea95]. Similar principles are applied in "programming by contract" as practised in Eiffel [Mey88].

In such methods the relationship between a software component and its clients is viewed as a formal agreement, expressing each party's rights and agreements. A big part of most formal approaches is the semantic specifications of operations. The behaviour of an operation is expressed by means of preconditions and postconditions. Preconditions express the properties that must hold whenever the operation is called; postconditions describe the properties that the operation guarantees when it returns. Another part is the use of state invariants which are more general and must be preserved by all operations.

While these methods have presented good results [Lin94], [Hal96], in this dissertation we want to investigate a completely different approach, where we don't focus on declarative behavioural specifications, but see how operational behaviour information can aid in managing the evolution of software systems.

1.5.4 Consistency of Class Libraries

This follows the work of some authors who have more recently focused on particular problems with evolving class libraries and frameworks [Str86], [KL92], [Lam93], [KKS96] instead of focusing on full behavioural correctness. [Mez97] divides these problems in vertical and horizontal evolution conflicts. Horizontal conflicts occur when changes to a base class invalidate inheritors, while vertical conflicts occur when the base class is extended by an inheritor in a way that was not anticipated by the base class designer.

[KL92], [Lam93] and [Str86] give an overview of some problems , but they offer no particular solutions. They all emphasise the importance of clear interface descriptions, the first two stressing the importance of structural information. [Mez97] suggests an automatic consistency maintenance system, where designers are enabled to formulate properties of the base module to be propagated to the inheritors during composition. Changes to the base module are monitored to filter out alterations that may invalidate already existing inheritors. The composition behaviour of classes is then adjusted such that before the integration/reintegration of the corresponding modules is performed, the maintenance of the base module properties and the protection of the inheritor is ensured. The specifications about the design and implementation specifics of the base class to be propagated to inheritors are explicitly formulated by the base module designer in a simple description language, called CCL (cooperation contract language). In this language it can, for example, be specified that certain methods are functional or that all methods changing a certain instance variable should also call a certain method. A meta-programming approach is then proposed to alter the behaviour of linguistic constructs responsible for the evolution.

1.5.5 Relationship with Reuse Contracts

Refactoring approaches help in making a number of frequently occurring changes. The work from [KKS96] helps in automatically checking the validity of a number of design constraints, while Mezini's work additionally helps in automatically solving some problems that were specified by designers. No mechanisms exist to generally assist developers in iterating over their implementations. When a developer wants to make a change to a reusable component it is hard to assess where and how that change influences the applications built on this component. Similarly, when an application developer wants to upgrade his application to a new version of the components he reused, it is hard to assess whether and where problems might occur. Reuse contracts aim to address this problem.

1.6 Our Approach: Reuse Contracts

1.6.1 Summary: the Problems

In section 1.2 we argued that there are some fundamental technical issues that need to be tackled in order to establish systematic reuse. Now that we have given an overview of the state of the art, we can clearly distinguish those problems.

1. There is a trade-off between black-box and white-box interfaces. While black boxes restrict flexibility, white boxes are harder to use and more fragile to change. The problem with using white boxes is that in order to reuse them, their entire implementation needs to be understood in order to find the necessary information. Therefore, we plead for a more *selective white-box* mechanism[6]. Such interfaces make only those implementation details visible that are required for reusers, while shielding unnecessary details. As a consequence, adapters only rely on the information given in the interface and therefore are more brittle to change.

[6] Term coined to us by Karl Lieberherr [Lie96].

2. As it is not easy to decide what information is needed by reusers, and thus what information should be made public, a mechanism for *layering of interfaces* is called for. Adapters making planned changes can limit themselves to considering the most abstract interfaces, while others can consult the interfaces that show more detail.

3. Much of what is currently being proposed for object-oriented software reuse (cookbooks, design patterns,...) lacks *formal* notation and rules. When using informal descriptions of reusable artifacts only, reasoning about, for example, how to reuse an artifact, effort estimations, or the impact of changes to reusers can only be done by error prone informal reasoning. No discipline in the reuse of artifacts can be enforced. However, as we want components to be reused by as many applications as possible, the guidance should not be *too coercive*.

4. The lack of support for *impact analysis* and *change propagation* leads to a proliferation of versions, which increases the maintenance problems. A number of problems can be detected by explicitly documenting the assumptions system parts make about each other. Disciplined reuse requires models expressed in a formal notation and formal rules describing how changes to reusable components affect reusers.

5. Yet other problems are related to the life cycle of reusable components. Too much burden is placed on the user of reusable components. Every user must detect possible reusable components, adapt them to his needs and integrate them. Furthermore, when new versions appear he is largely on his own to perform the upgrade. It would be beneficial to shift some of this burden back to the developer of reusable components. On the other hand, feedback from reusers on how components were adapted is crucial in order to build truly reusable components. This relates to the need for a *contractual interface* between component providers and reusers.

1.6.2 Another Example

Consider the example of a Collection hierarchy. A class `Set` defines a method `add` to add one element to a set and a method `addAll` to add a set of elements simultaneously. When creating a subclass `CountableSet` of `Set` that keeps a count of the number of elements in the set, we need information about which methods call what other methods, in order to decide which methods need to be overridden. In other words, we need a specialisation interface. For example, if we know that `addAll` calls `add` in its implementation, we know it is sufficient to override the method `add` to perform the counting. This is depicted on the left-hand side of figure 1.12.

Suppose now that we want to make an optimised version `OptimisedSet` of `Set`. In this version `addAll` stores the added elements directly rather than invoking the `add` method to do this. When `Set` is replaced by `OptimisedSet`, this leads to

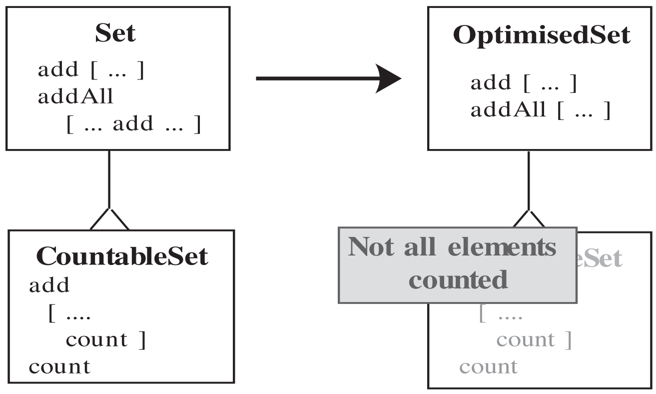

Figure 1.12: Evolution of Set

inconsistent behaviour in `CountableSet`. Not all additions will be counted. This is depicted in figure 1.12.

The reason for this failure is again that the implicit assumption made by `CountableSet`, that `addAll` invokes `add`, is broken in `OptimisedSet`. As discussed in section 1.1, we say that `addAll` and `add` have become inconsistent operations. Although in this simple example the conflict can easily be derived from the code, in larger examples this is not so trivial. In practice it should be possible to detect such conflicts without code inspection. This requires information about dependencies between components and about the assumptions made during reuse to be made explicit. The problem for this example is depicted in figure 1.13. While the developer of `CountableSet` assumed that the adaptation of `add` would also affect `addAll`, this is no longer true after the upgrading of `Set` to `OptimisedSet`. Allowing reusers to explicitly document the assumptions they make about the components that they reuse can therefore assist in detecting such problems.

1.6.3 Reuse Contracts

As an answer to these problems we propose to base the co-operation between the component developer and the component reuser on an *explicit reuse contract* that satisfies the requirements that were enumerated in section 1.6.1 and is based on explicitly documenting assumptions as described above.

Reuse contracts provide interface descriptions that partially document the internal structure of components. This is depicted in figure 1.14 by denoting `add` between braces next to `addAll` in the interface of `Set`. Reuse contracts thus offer guidelines for reusing and adapting components. For this example, the reuse contract specifies that `addAll` *relies on* `add`. This implies that `addAll` *can* call `add`, not that it always *will* call `add`. For a class `Set` to comply to the reuse contract `SetRC` as depicted

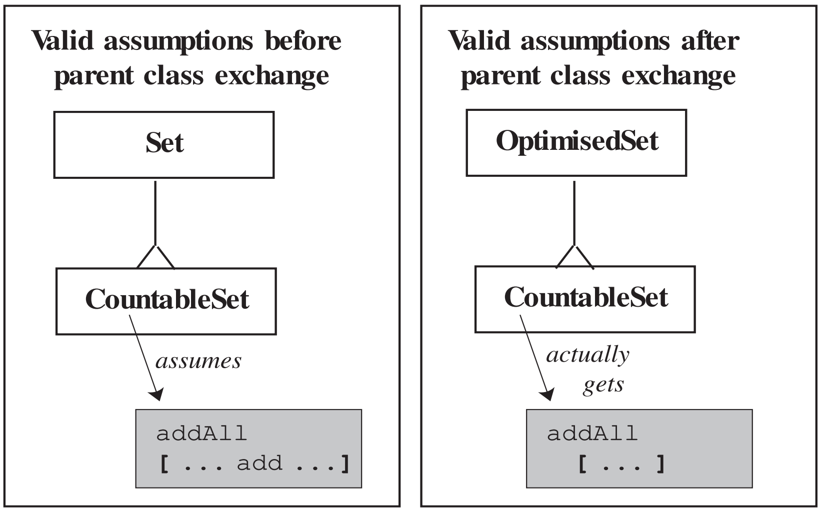

Figure 1.13: Broken Assumptions

in figure 1.14, `addAll` needs to invoke `add` somewhere in its implementation. This does not mean it will always be invoked at run-time. The reason for this approach is that we want to adhere to simple models that can be automatically processed. Checking whether a certain method will always be called would require extensive data-flow analysis. The information given in the reuse contract is also limited in that it gives no information on how a method can be called, in what order methods can be called, etc. The reuse contract information will thus not always be sufficient to know how a component can be reused. It does however provide a very useful view on the code allowing to get an understanding of its structure and behaviour in a fast and intuitive manner.

Reuse contracts can only be adapted by means of *reuse operators*. These record the protocol between developers and users of a reusable component. Similarly, they can record the relationship between different layers of specifications of one component. A basic set of reuse operators is defined, that is complete in that the operators allow expressing all changes to the basic reuse contract model. The extension operator adds new elements to an interface, while its inverse, cancellation, removes elements. The refinement operator adds extra interdependencies to an interface, while its inverse, coarsening, removes them. For example, the addition of a call to `count` in the implementation of `add` is a refinement. This is depicted on the left of figure 1.14. The reuse operators thus express a contract between a component provider and reuser[7].

[7]Note that a reuse contract denotes *two* component descriptions related by *operators*. Sometimes however the term contract will also be used to denote the description of the opeartional behaviour

The formal documentation of a contract between providers and reusers assists in handling change propagation. The key to this approach is the observation that on composition developers make *assumptions* about how the components will behave and interact. This is what makes them fragile to change. When replacing a component with another version some of these assumptions can be broken. The interface part of reuse contracts allows documenting what assumptions can safely be made about a component, for example, `addAll` relies on `add`. The reuse operators allow specifying what assumptions are actually relied on during reuse. For example, when refining `add` this implies affecting `addAll`. This documentation allows checking whether these assumptions are broken after a reusable component is adapted, which facilitates the propagation of changes to components by indicating where conflicts might occur. In the example, it is then clear that when `Set` is replaced by `OptimisedSet`, the assumption that `addAll` relies on `add` is broken. This is depicted in figure 1.14 by expressing the upgrading from `Set` to `OptimisedSet` through a coarsening. A set of rules can be developed that identify conflict situations based on reuse operators. For example, one rule states that when one modifier performs a coarsening removing a call to one operation, while the other modifier performs a refinement of the same operation the operation of which the call is removed and the operation from which it is removed become inconsistent. In the example, `add` and `addAll` become inconsistent. These conflict detection rules do not guarantee full behavioural correctness, as they work with limited models. They are however very easy to use and provide developers with useful information on change propagation based on minimal effort. They not only indicate where a problem might occur, but also what problem, which allows suggesting possible solutions.

Reuse contracts thus help in managing the evolution of components. Reusers can benefit from improvements to the components they reuse and the proliferation of different versions of reusable components can be kept to a minimum. In a similar vein, the reuse operators can be valuable to the component developer in order to assess the impact of changes and to decide whether changes should be made. Moreover, the reuse operators provide developers with a vocabulary to discuss reuse and evolution, that assists them in better understanding the structure and behaviour of the systems they work with.

While the use of reuse operators encourages disciplined reuse, the presence of inverse operators allows for making changes that break the original contract. This leaves room for unanticipated changes in a way that more coercive approaches don't.

The main contributions of the reuse contract approach are:

- Reuse contracts can be used as structured documentation of reusable components and generally assist a software engineer in adapting components to particular needs.

of one component.

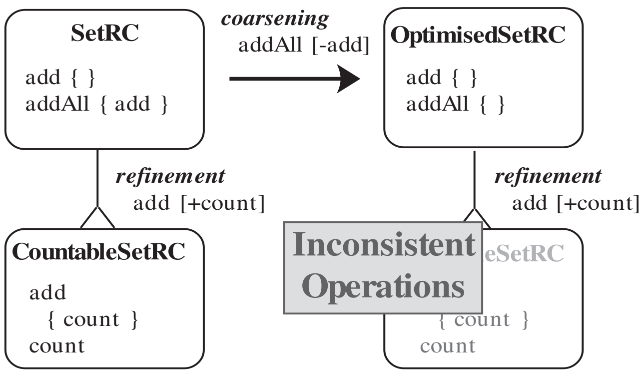

Figure 1.14: Inconsistent Operations on Set

- Reuse contracts encourage disciplined reuse, without being too coercive and provide a vocabulary and notation to discuss reuse. They do this through simple models that are easy to learn and intuitive in use.
- On evolution, reuse contracts assist in assessing how much work is necessary to update previously built applications, where and how to test and how to adjust these applications.

Reuse contracts thus help in breaking down the barriers between component producers and component reusers.

1.6.4 Structure of the Dissertation

In chapter 2 a basic definition of reuse contracts and the basic operators are introduced. In chapter 3 a number of rules based on these operators, that allow conflict detection upon evolution, is given. Chapter 4 describes how the basic operators can be combined to form bigger, more realistic operators and how the conflict detection rules behave under this composition. These first few chapters give a general definition of reuse contracts, that could be applied to different kinds of reuse mechanisms. In chapter 5 we demonstrate how reuse contracts can be applied to object-oriented class libraries and frameworks. Chapter 6 then discusses a concrete project where reuse contracts were applied, while chapter 7 gives an overview of how reuse contracts can be used in the different stages of the software life cycle. Finally, chapter 8 concludes.