Chapter 7

Reuse Contracts at Work

In chapters 2 to 5 reuse contracts were described on a theoretical level. The previous chapter already demonstrated reuse contracts at work, assisting in the design and evolution of a framework. In this chapter we give an overview of how reuse contracts can support software engineers in the different phases of the software life cycle and in particular what kinds of tools based on reuse contracts can be conceived to assist in these tasks. We also discuss how reuse contracts can complement other techniques as, for example, design patterns and refactoring.

In sections 7.1 through 7.3 we start by discussing some basic tools that can form the heart of a general development environment. In sections 7.4 through 7.8 we then discuss how tools that assist software engineers in different tasks can be built on top of these. Some of these tools have already been built and experiments have been conducted with them. Other tools have not yet been constructed, but the feasibility of building them is clear for two reasons. First, reuse contracts only use static information. This makes the building of tools for reuse contracts more straightforward than for other approaches that require proof checking or extensive data flow analysis. Second, we provided a formal definition of reuse contracts. The already existing tools were based on a first minimal definition of reuse contracts for class hierarchies. The formal definitions served as specifications for the tools. This approach can again be followed for the reuse contracts as discussed in this dissertation.

The tools were built and experiments conducted in cooperation with other members of the Programming Technology Lab, part of it within a project with an industrial partner, and by students completing their graduation thesis at our Lab. Note also that, as in the previous chapter, our experiments focus on tools for class libraries and object-oriented frameworks. Similar tools can however be conceived for other kinds of systems, as will be discussed in chapter 8.

Most of the tools built, and therefore most of the experiments carried out, are only concerned with what we call *single-class reuse contracts*. These are reuse contracts that only contain one participant, in other words reuse contracts representing classes. The only internal dependencies documented there are self sends. The only operators that make sense on them are the participant versions of extension, refinement and concretisation (and their inverses), specialisation and the combinations of the above. Single-class reuse contracts provide a very clear documentation of class hierarchies.

7.1 Extraction

Extracting Reuse Contracts from Inheritance Hierarchies

A first basic tool is a reuse contract *extractor*. By that we refer to a tool that extracts reuse contract information from source code. The difficulty of building such a tool obviously increases with the complexity of the information we want to include. Recall that we only use static information. When we mention self sends in a specialisation clause we indicate that a self send of a certain method *might* be performed, not that it *will* be performed. Deducing information on actual calls requires extensive data-flow analysis. Here we only deduce what can be seen from the syntax.

A basic tool was built that extracts information on self sends and whether methods are abstract or concrete from Smalltalk source code. Whether a method in Smalltalk is abstract is derived from its body by checking whether it holds the message `self subclassResponsibility`. In languages such as C++ and Java the abstractness of a method is explicitly declared, so obtaining this information in such languages is trivial. The same process that was carried out for Smalltalk can therefore easily be repeated for other object-oriented languages.

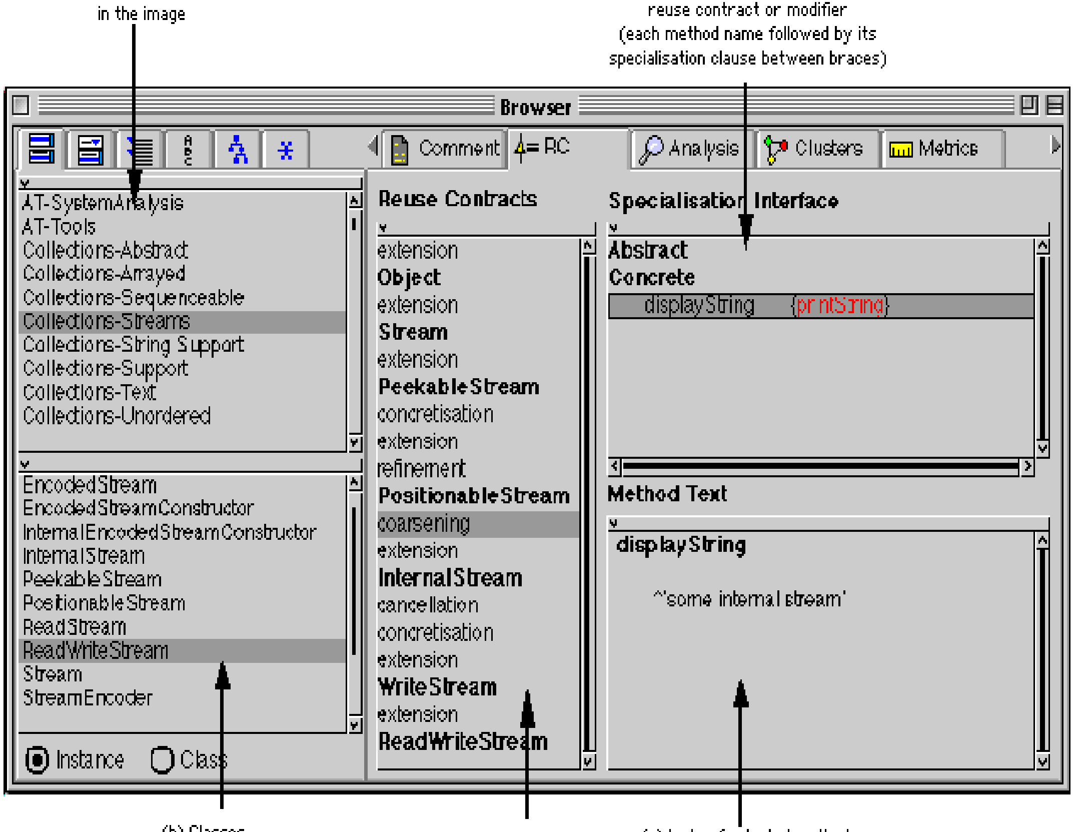
Figure 7.1 gives an overview of the extractor. In panels (a) and (b) the categories and their classes are represented as in a regular browser. In panel (c) the class selected in (b) and its predecessors are depicted (we will discuss the operators between below). Whenever a class is selected in panel (c), panel (d) shows this class's specialisation interface, i.e., all its methods divided into abstract and concrete ones plus their specialisation clauses. These specialisation clauses are again added between braces. When selecting a method signature in panel (d) its specialisation clause is depicted in panel (e) and its body in panel (f). Note that this extractor is just an inspector, it is impossible to make any changes while in the extractor.

Extracting Reuse Operators

Once a reuse contract is deduced for all classes in a branch of the inheritance tree, it is possible to decompose each superclass/subclass relationship into different

(a) Categories in the image

(d) Specialisation interface of selected reuse contract or modifier (each method name followed by its specialisation clause between braces)



```
displayString

    ^'some internal stream'
```

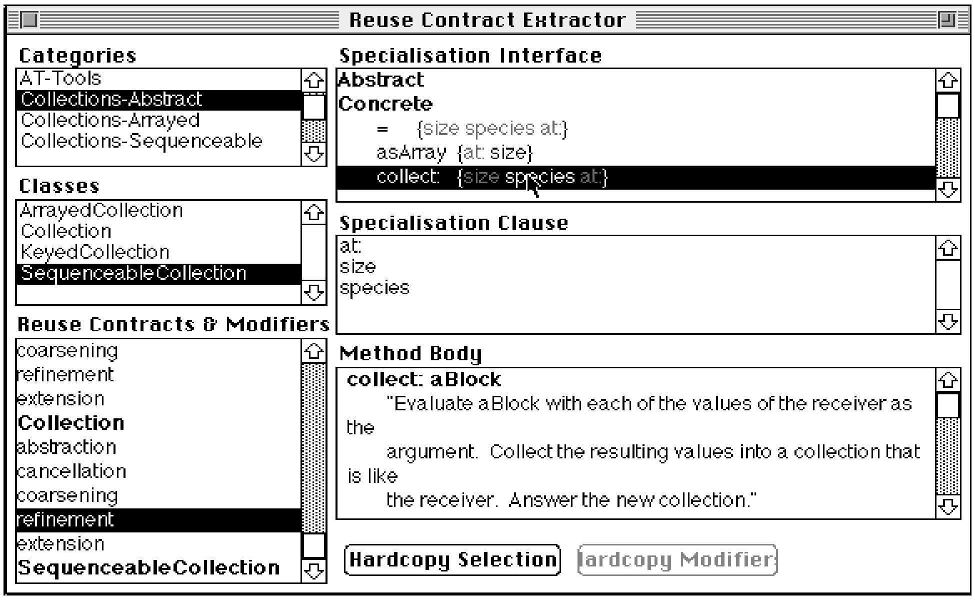
(b) Classes in selected category

(c) Extracted reuse contracts and their modifiers

(e) body of selected method

Figure 7.1: Reuse Contract Extractor

reuse operators. As explained before, different decompositions are possible[1], but when choosing some canonical form, the same decomposition will be returned every time. In this extractor only the basic operators are deduced. Because of their orthogonality, the decomposition is then straightforward. Methods with new names are part of an extension, methods that are overridden with as body "self shouldNotImplement" are part of a cancellation, etc.



```
collect: aBlock
	"Evaluate aBlock with each of the values of the receiver as the
	argument.  Collect the resulting values into a collection that is like
	the receiver.  Answer the new collection."
```

Figure 7.2: Decomposition into Reuse Operators

Figure 7.2 shows the decomposition between `SequencableCollection` and `Collection`. In the previous example a class was selected in section (c) and this class's specialisation interface was shown. In figure 7.2 the name of a reuse operator is selected in section (c) and the corresponding reuse modifier is depicted in section (d). In figure 7.2 refinement is selected and thus section (d) shows all methods from `Collection` that are refined in `SequencableCollection`. The specialisation interface repeats all method names that were already present in the specialisation clauses on `Collection` in black and shows all new method names in green (here grey). Sections (e) and (f) behave as before.

---

[1] As a degenerate case everything can be accomplished by a cancellation and an extension.

Combined Operators

Note that the modification between the two classes is decomposed into a number of operators. The reason for this is that the same method often occurs in more than one reuse operator between two classes. When inspecting the extracted modifiers it can be observed that certain combinations often occur. One example is refinement and concretisation. Therefore we added a combined operator: layered concretisation. Another often occurring combination which we did not define explicitly is refinement and coarsening of the same method. For example, the method `collect:` the refinement of which is depicted in figure 7.2, is also coarsened between `Collection` and `SequencableCollection`. The current extractor can therefore be extended with facilities for detecting all adjustments to one method or for finding combined operators. When including these operators, deciding which decomposition to take might be harder. For example, is the addition of a new method part of an extension or of an extending refinement?

Filtering Design Important Information

Another problem with extracting reuse contract information from source code is that a tool will extract information on *all* methods and all message sends, without distinguishing methods that are important to the design from implementation specific ones. Therefore, it is crucial to the usefulness of extracting tools that filtering mechanisms are included. By filtering we refer to the possibility of removing unnecessary information from the derived reuse contracts. To a certain extent this can be done automatically. For example, one can specify to omit all accessor methods or all methods with a name containing a certain part and all methods referring to these methods. In addition, users can be enabled to manually remove certain methods from the client and specialisation interfaces.

Note that making changes to the reuse contracts can have an influence on the well-formedness of the contracts, as well as on the reuse operators. For example, when one of the operators in a sequence was a participant extension, but all methods added by this extension are filtered away, the extension is no longer valid. Of course, checking of well-formedness and updating of the reuse operators after filtering can again be done automatically.

Extracting Multi-Class Reuse Contracts

The extraction of multi-class reuse contracts from code is of course a much more difficult task, as different kinds of acquaintance relationships must be taken into account. The task will be easier to perform in statically typed languages such as C++ or Java than in Smalltalk. Even then it is hard to decide which acquaintance relationships to include in a contract and which ones not. One class might participate in a number of different contracts with different other classes and it is not feasible to automatically distinguish which relationships should be coupled in one contract and which separated into different contracts. Assistance from the user is therefore even more crucial here.

An interactive approach can be adopted where an extractor provides some information, but a user can add constraints or ask for other information, thus steering the extraction process. A user can, for example, start by asking to extract information about all dependencies involving a certain class. First information about all methods can be given or the user can immediately include a number of methods which he wants information about. The tool can then check what calls are made from within these (or all) methods and infer possible acquaintances from this information. In statically typed languages these acquaintances can be automatically deduced, in dynamically typed languages again assistance from the user is required. After providing this information, the user can see which other classes are involved and select some of these and ask the extractor to show only the dependencies involving these particular classes. By repeating this process for different class combinations, different contracts which a class participates in can be deduced. In another scenario a user can immediately select a number of classes and ask for information on contracts between them. Information on the calling structure can be deduced and again reuse contracts containing only the desired information can be obtained through filtering.

7.2 Compliance Checking

A tool complementary to an extractor is what we call a *compliance checker*. Here, the inverse way of working is followed. The user gives reuse contracts and operators that model the design of (part of) his code and declares which (groups of) classes should comply to which reuse contracts. The checker then verifies whether the declarations made by the user are correct. This includes checking the well-formedness of the given reuse contracts, the compliance of the code with the reuse contracts, as well as the validity of the operators.

An experiment concerning *single-class* reuse contracts was conducted in Java [Cor97]. Java interfaces were extended to hold reuse contract information, making it possible to verify whether Java code complies with the given reuse contracts. As an illustration the Java Abstract Windowing Toolkit (AWT) framework was documented, since it is the most generally distributed Java framework. We use this example here to explain the experiment.

As a first part of the experiment, the `interface` construct of Java was extended with specialisation clauses[2]. Part of the reuse contract for `Component`, the root class in the hierarchy can then be described as below. We only show a part here because

---

[2] Whether methods are concrete or abstract is already included in Java interfaces.

the `Component` class contains over 70 methods that are not all related. We will show further on how the documentation with reuse contracts helps in assessing the design of `Component`. For now, we just use a fragment as an example.

```
public interface ComponentRC {
  public Dimension preferredSize();
  public Dimension minimumSize();
  public void layout;
  public void validate ()
    { layout() };
  public void invalidate();
}
```

In the example, only one method, `validate`, has a non-empty specialisation clause. It is again denoted between braces. As these specialisation clauses only concern self sends, the name of the method suffices. Furthermore, we see that all these methods are concrete, as in the Java interface syntax the keyword '`abstract`' is added when a method is abstract and otherwise nothing is mentioned. We followed this syntax.

As a second step in the experiment, next to the Java keyword `extends`, that can be used to express that a particular interface is deduced from another one, a set of keywords `refines`, `concretises`, ... expressing our reuse operators was added. The keyword `expands` was used to model reuse contract extension, because we wanted to preserve the standard Java interfaces. As an example, consider the reuse contract for the class `Container`, a direct subclass of `Component`; it is an extension and a refinement of `ComponentRC`.

```
public interface ContainerExtRC expands ComponentRC {
  public LayoutManager getLayout ();
  public void setLayout(LayoutManager m);
}

public interface ContainerRC refines ContainerExtRC {
  public void layout ()
    { + getLayout };
  public Dimension preferredSize()
    { + getLayout };
  public Dimension minimumSize()
    { + getLayout };
  public void setLayout(LayoutManager m
    { + invalidate() }
}
```

Note that this is partially a refining extension as the newly added method `setLayout` is immediately refined with a call to `getLayout`. The extension of the interface construct and the addition of the operators enables users to describe their designs by means of reuse contracts.

As a third step in the experiment, the checker of the `implements` relationship, which declares that a class complies to one or multiple interfaces, was extended to include checking of self sends. This required extraction of information on which self sends are possibly performed by a method. This is similar to extraction, only here we know what we are looking for. As in this example, the user most likely only includes certain method descriptions and only denotes those self sends in his reuse contracts that are crucial to the design, while leaving out implementation specific calls. Compliance of a class with a reuse contract can then be declared as below and subsequently checked. Note that the `Container` class implements the second of the two reuse contracts declared above for it.

```
public class Component implements ComponentRC
{
  // code for the class Component
}

public class Container implements ContainerRC
{
  // code for the class Container
}
```

The checking process consists of several parts. If a complete reuse contract is given, its well-formedness is checked. If the modification is given, the applicability of the reuse operators as defined by the user is checked and the result is computed. Third, it is verified whether the class complies with the reuse contract. This implies checking whether all declared method descriptions are present, whether the abstractness annotation is correct and whether the declared self sends are part of the code.

Checking compliance of code with *multi-class* reuse contracts will be less difficult than extracting them, because the reuse contract states the important acquaintance relationships and what kind of relationship they are. Furthermore, the user indicates which classes map onto each participant and optionally also which implementation features to which participant relationships. Again, this task will be easier in typed languages such as Java or C++ than in Smalltalk, as the types of the instance variables and arguments will help in verifying whether the described acquaintance relationships are present in the code.

7.3 A Drawing Tool

A final basic tool is a drawing tool as present in any CASE tool. As the graphical representation of reuse contracts closely resembles notations from classical object-oriented methodologies such as OMT and UML, it is clear that extending such a drawing tool with reuse contract information will be straightforward. What should not be forgotten is that all the well-formedness constraints have to be built into the tool. As these are all straightforward constraints on the notation however, this will not pose too much difficulties.

The three basic tools as described above can form the basis for a general development environment with reuse contract support. While each of them can be useful on its own, their real value only becomes apparent when using them in combination or when building other features on top of them. This is what we will discuss in the following sections.

7.4 Documentation

A first obvious use of reuse contracts is the documentation of the structure of software, be it entire systems, frameworks, components or class libraries.

7.4.1 Dependencies between System Parts

First and foremost, reuse contracts document the way different parts of a system depend on and interact with each other, thus documenting the assumptions one part makes about the others.

A first remark that should be made is that because of the possibility of building extractors and compliance checkers, it is much easier to keep this form of documentation consistent when the software evolves. The difficulty in building these tools is that when assisting in, for example, incremental development, one does not want the tool to be triggered all the time. The different kinds of evolution and how the tools can assist in the different cases will be further discussed in section 7.7.1.

7.4.2 Assistance of the Software Engineer

The information provided by reuse contracts is crucial to software engineers in the customisation process. The kind of information that is now provided by reuse contracts is only available in natural language in manuals accompanying frameworks and libraries. Here it is closely linked to the source code.

When looking at the documentation of existing frameworks [Obj91], [Tur91] and when reading the accompanying manuals, it is clear that a lot of information for the customiser is given in natural language. The description of how to make your own application is completely described in natural language. A general description of the framework is given and then an overview of all classes and member functions with a description of what the function does, if you are allowed to override them and what you have to ensure when overriding. While this kind of application frameworks are usually well-structured and a lot of applications can be built by following the instructions, a number of things are clear.

First, in order to make complex applications an extensive understanding of the structure of the framework is necessary. This can more easily be acquired through reuse contracts, than through the current natural language specifications.

Second, open any manual and you will find sentences as 'to adapt this you need to override that method' and 'be sure when overriding method x to call method z'. This is typically the kind of information that can easily be captured by reuse contracts and operators. The last statement above, for example, can then be expressed as 'refine x with z'. If we manage to encode this information in a more structured form, we will also be able to provide better customisation tools.

Third, sometimes one wants to make other adjustments that were not foreseen by the framework developers. Again a good understanding is crucial, because the specifications no longer state which methods to override and how this must be done after making a certain change. Reuse contracts are able to provide some assistance in this case as well, as will be discussed in section 7.7.

One difficult issue is the way in which different adaptations are connected. As a framework will always be described by a set of related contracts, it is possible to discover which contracts might need to be adapted after certain changes. This can be automated, when all these adaptations are expressed through modifiers.

After making one or more adaptations a consistency check can be performed which might reveal inconsistent methods or method captures caused by the former adaptations. These might hint at the places where further attention is needed. The use of the rules in this way was already demonstrated in chapter 6.

7.4.3 Layering of Design

Another crucial feature with respect to enabling reuse is the possibility to express layered designs. In the design of reusable systems there is a classic tension between generality and flexibility. The more general a design is expressed, the more specialisations can be made from it, but the more work needs to be done to create a concrete instance. The more specific a design is, the fewer possible specialisations there are, but the easier it becomes to reuse the design. Consider the example of a library including buttons[3], as depicted in figure 7.3.

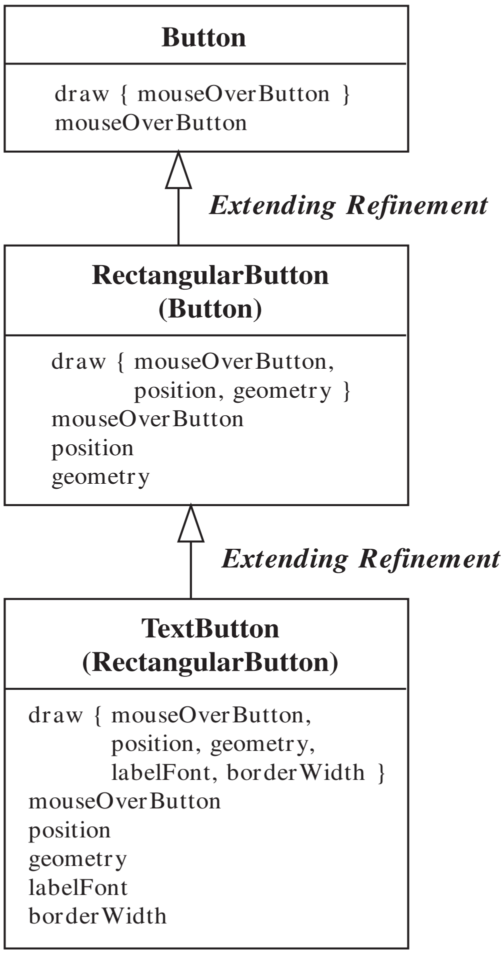


Figure 7.3: A Layered Design of Buttons

On the top layer one can have a protocol for buttons that only states that a button has a method `draw` in order to display it on the screen, that in turn calls `mouseOverButton` to check whether the button should be highlighted. On the lowest level, there can be a protocol for text buttons specifying, for example, that text buttons must be drawn with a certain font, how the border must be drawn, etc. This last protocol can be very useful, because one might often need to create text buttons and does not always want to re-implement all the details. Other layers between the top protocol and the text-button protocol might be useful to express, for example, all commonalities for rectangular buttons. The top layer continues to

---

[3] Example based on [KL92], where the problem of layered protocols is clearly discussed.

be useful because one can never foresee all ways in which future developers might want to reuse the library. Therefore only providing, for example, rectangular buttons might be too strong a restriction.

The possibility of providing these different layers enables library specifiers to decrease the tension between generality and flexibility. Reuse contracts document that part of the design of a component that is 'useful' for specialisers, while leaving out implementation-specific details. It is very difficult to decide which part of the design to document. However, by providing different reuse contracts for one component providing different degrees of detail, this decision may be less crucial. Depending on which button is needed, a developer can choose to start out from either one of these contracts for his customisation. Furthermore, by declaring which reuse contracts he complies with a developer can indicate what assumptions he is relying on. It is also conceivable that these different levels are used by different kinds of developers. Customisers are then only allowed to make changes to lower, more detailed levels, while component developers are allowed to make changes to higher levels. This is also related to the following issue.

7.4.4 Core Methods versus Peripheral Methods

Another way of looking at these layered protocols is to state that some methods make up the core of a component, while others are more peripheral. Going back to the button example, the `draw` method on a text button will probably rely on methods as `font` and `geometry`. One can then say that `font` and `geometry` are core methods, while `draw` is peripheral[4]. Adapting core methods will result in changes to the peripheral methods, while adapting peripheral methods does not affect the core methods. In a similar vein, information about the clustering of methods, i.e., knowing which groups of methods rely on each other and which are independent is useful in order to assess the impact of changes.

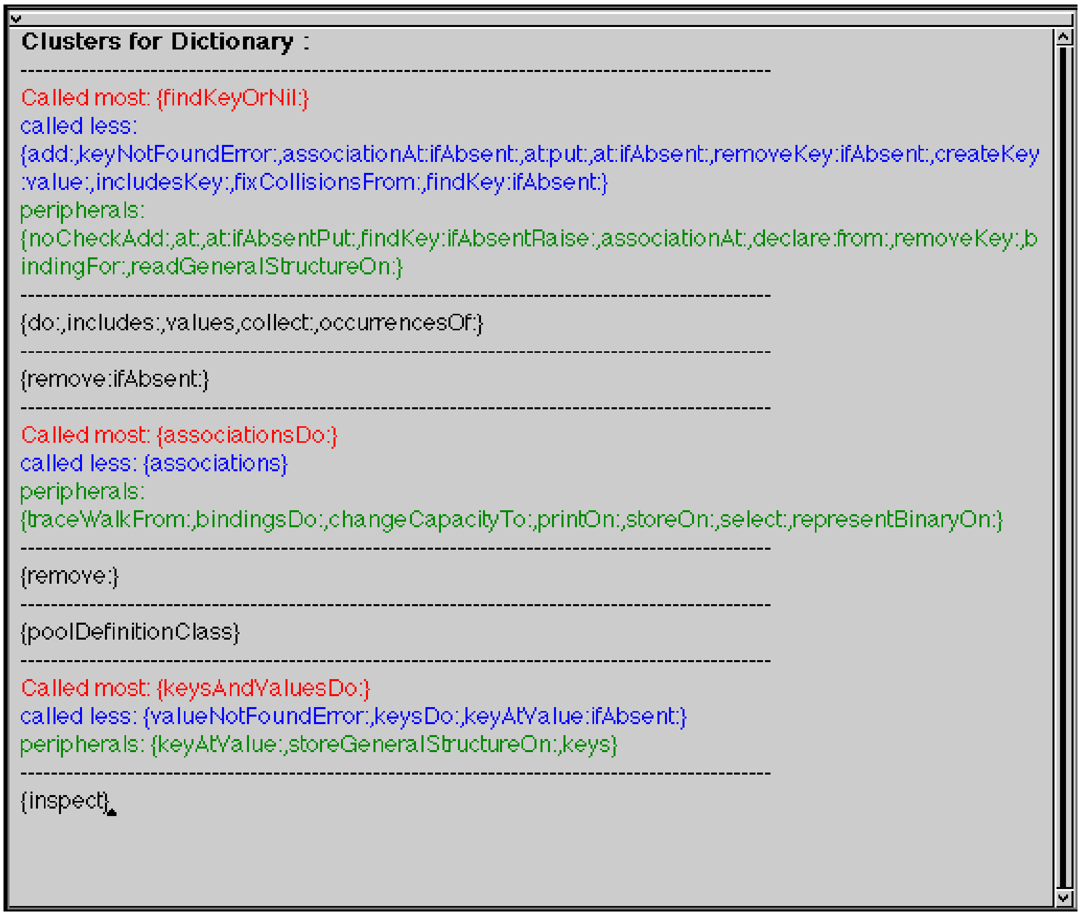
This is akin to the use of template methods in object-oriented systems in order to express the basic design of a system, as discussed in chapters 1 and 5. Template methods rely on abstract methods and default methods that need to be tuned to specific situations. Concretising abstract methods or overriding default methods is a fast way to adapt a general behaviour to particular circumstances. Overriding template methods requires far more work and can have much graver consequences.

Simple experiments have shown that, based on reuse contracts, information about clustering of methods can be detected. Also the distinction between core and peripheral methods can be made automatically to a certain extent.

Figure 7.4 demonstrates the result of a small experiment carried out on top of the reuse contract extractor in Smalltalk. A ClusterViewer was built that extracts information about the clustering of methods in a class. The figure demonstrates

---

[4] Terminology due to [Lam93].



```
Clusters for Dictionary :
------------------------------------------------------------------------------------------
Called most: {findKeyOrNil:}
called less:
{add:,keyNotFoundError:,associationAt:ifAbsent:,at:put:,at:ifAbsent:,removeKey:ifAbsent:,createKey
:value:,includesKey:,fixCollisionsFrom:,findKey:ifAbsent:}
peripherals:
{noCheckAdd:,at:,at:ifAbsentPut:,findKey:ifAbsentRaise:,associationAt:,declare:from:,removeKey:,b
indingFor:,readGeneralStructureOn:}
------------------------------------------------------------------------------------------
{do:,includes:,values,collect:,occurrencesOf:}
------------------------------------------------------------------------------------------
{remove:ifAbsent:}
------------------------------------------------------------------------------------------
Called most: {associationsDo:}
called less: {associations}
peripherals:
{traceWalkFrom:,bindingsDo:,changeCapacityTo:,printOn:,storeOn:,select:,representBinaryOn:}
------------------------------------------------------------------------------------------
{remove:}
------------------------------------------------------------------------------------------
{poolDefinitionClass}
------------------------------------------------------------------------------------------
Called most: {keysAndValuesDo:}
called less: {valueNotFoundError:,keysDo:,keyAtValue:ifAbsent:}
peripherals: {keyAtValue:,storeGeneralStructureOn:,keys}
------------------------------------------------------------------------------------------
{inspect}
```

Figure 7.4: Clustering of the Class Dictionary

the results for class `Dictionary`. `Dictionary` contains eight clusters, depicted by dotted lines. These clusters are computed by simply taking the transitive closures of the specialisation clauses of all methods and grouping together the methods that are in each other's closure. This results in a set of clusters, for example, `Dictionary` has a cluster of methods concerning keys and values (one but last from the bottom).

Within each cluster a subdivision is made between methods that are called most and less and peripheral methods. The methods that are marked "called most" are methods that are called by more than 30% of the methods in the cluster[5]. The peripheral methods are methods that are only in the cluster because they call some of the other methods, but that do not get called themselves. The methods marked "called less" are the remaining methods. This immediately gives an insight in which methods will cause large repercussions when changed and which methods can be changed more easily.

More experiments are necessary to get optimal information. For example, some of the clusters are still rather large. Trying to find subclusters by, for example, leaving out the most called methods when computing transitive closures provides more insight. Another very interesting experiment is to compare the clusters of a class and its subclasses. In the figure, we only computed the clusters for the modification between `Dictionary` and its direct superclass. It is also possible to compute the complete reuse contract for each class and derive clusters on these. That provides yet another view on the class hierarchy. It could also be very interesting to see how the reuse operators relate to these clusters.

7.4.5 Layering of Class Hierarchies

The concept of layering also appears on another level in object-oriented systems, namely the layering of class hierarchies. Because of this layering, the interface and 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. Because of the different layers of superclasses and subclasses that make up the definition of a class, it can become difficult to understand which operations are invoked as an instance executes an operation request. This is perceived by many users of object-oriented class libraries and frameworks as one of the major obstacles to reusing them. There are some environments that already give some support to tackle this problem by providing information on all methods of a subclass, not just on the ones (re)defined at that level. Take, for example, the Visualworks Full Browser.

Reuse contracts address this problem in three ways. First, the documentation of the difference between a class and its subclasses by means of the reuse operators helps in understanding the structure of a hierarchy. Distinguishing extension,

---

[5] We just took a random treshold here. More experiments are needed to point out what values give optimal results. Another option is to take all methods that are called more than average.

refinement and concretisation helps in understanding how a subclass adapts the design of its superclass, while the inverse operators indicate how the design is breached.

Second, because the modifiers that document each subclassing step keep track of all method calls that are added and removed, the complete reuse contract at every level can be computed, thus providing inheritors with the desired information without having to browse up and down the inheritance hierarchy.

Third, if inheritors document the subclassing actions they perform by means of reuse operators, this will reveal potential misunderstandings. If, for example, an inheritor declares that it is 'refining method x with z' and method x already invokes z this will be signalled because the applicability test will fail. A good example of how such problems can occur in class hierarchies can be found in the standard Visualworks class library. The reason why the names of primitive methods (e.g., `basicAt:`, `basicAt:put:`, ...) in 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.

7.4.6 Views

Different reuse contracts can provide different views on a component. Often the modification between a class and its subclass contains 30 or more methods. Leaf classes can contain up to 100 methods. This might indicate that the structure of the inheritance tree is too coarse, but maybe these are just classes that have a lot of different aspects. Some of the methods might concern the domain that the class represents, while others might concern their visual representation or persistency issues, ... By attaching different reuse contracts to one class, where one includes the domain methods, another one the visualisation methods, etc. the documentation will be much clearer. Furthermore, by making this distinction a developer only concerned with, for example, the persistency layer can ignore visualisation aspects and vice-versa.

As an example let us return to the documentation of the Java AWT. We already mentioned above that the reuse contract there represented only a small fraction of the entire class. A large number of methods on the class should not be presented in a reuse contract because they represent implementation-specific methods. Another reason for the large number of methods in the class is the heavy use of overloading. Investigating the clustering of the methods reveals however that the class `Component` contains two completely separate parts: a part for visualisation as partially described above and a part for event handling, that has absolutely no dependencies on the visualisation part or vice versa. Therefore, it is clearer to introduce two separate reuse contracts and declare that the class `Component` implements both.

```
public interface EventHandlingRC {
  public void deliverEvent (Event e)
    { postEvent (Event) };
  public void postEvent (Event e)
    { handleEvent () };
  public boolean handleEvent(Event e);
  . . .
}

public class Component implements ComponentRC, EventHandlingRC
{
  // code for the class Component
}
```

This is possible, as the event handling methods present a well-formed reuse contract on their own. Sometimes there might be problems to neatly divide classes in fragments, because some methods might take part in different views. More experiments are needed to assess how important this problem is and to decide how it can be solved.

7.5 Design Guidelines and Quality Assessment

The information given by the extractor tool or obtained when manually documenting a system with reuse contracts as was done with the AWT example can also give indications about the general quality of a design. Of course, it is never possible to judge a design purely on statistical grounds, but some features of a good or bad design are widely recognised. We discuss some of these and how their absence can be discovered through the extractor tool.

7.5.1 Well-formed Reuse Contracts

Some good design principles are already built into the definition of reuse contracts. One of these is that one can only perform a super call of a method with the same name. If a method performs a super call to another method, the extractor will come across this when checking well-formedness and signal this problem.

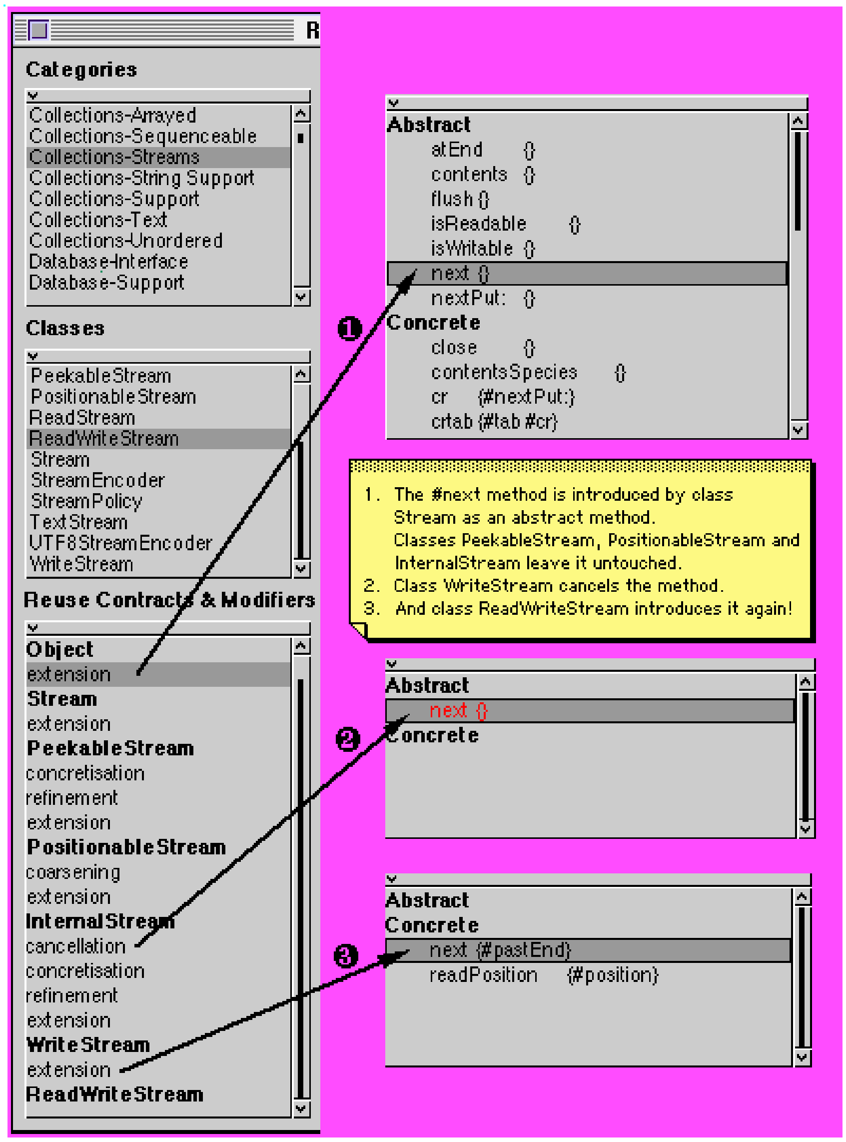
An even worse design flaw we came across when using the extractor is the invocation of methods that are not present at that level or higher up the class hierarchy. This is of course only possible in dynamically typed languages such as Smalltalk. We have found examples where a method on a superclass invokes a method that only exists on one of the subclasses. Probably, this did not cause problems, because the method on the superclass only performs this invocation on instances of the subclass, but still this is an error-prone design, so this must be signalled. Again, this problem will be detected when checking well-formedness, because one of the requirements in the definition is that all messages sent in a contract must correspond to methods present in this contract.

7.5.2 Assessments Based on the Contracts and Operators

The information given by reuse contracts also unveils other problems. For example, after extracting information from one application we found a lot of leaf classes in an application still having a large number of abstract methods. In other words, the leaf classes were no complete concretisations of their superclasses. When examining this closer it became clear that a number of abstract methods were defined too high in the hierarchy. As a consequence some branches of the hierarchy, which these methods were irrelevant for, had abstract methods at the bottom. This can be seen either by looking at the reuse contracts of the bottom classes, or when distinguishing between complete and partial concretisation, by checking whether a complete concretisation is performed along every branch under an abstract class.

Another indication concerning the structure of a hierarchy can be given by looking at the operators modelling subsequent subclassing. When a certain operator is applied and further on in the chain its inverse is applied, this might indicate a flaw in the hierarchy. Sometimes it gets even worse. Consider the next example from the VisualWorks `Stream` hierarchy.

Figure 7.5 shows parts of the decomposition deduced by the reuse contract extractor on the VisualWorks `Stream` hierarchy. On the right hand side of the figure three parts of subsequent reuse operators are modelled. The first shows how next is added as an abstract method to `Stream`. The second shows how the method `next` is cancelled in `WriteStream`, while the third shows it is added again in `ReadWriteStream`. Note that when `next` is cancelled it is still abstract. None of the intermediary classes have performed a concretisation of `next`. Two facts draw our attention to this situation. First, the cancellation of abstract methods might point at abstract methods being defined too high up in a hierarchy. Second, the subsequent extension, cancellation and extension with the same method might indicate a bad structuring. Note that in this case the designers of the hierarchy are probably aware of this situation as this is a classical case of trying to capture a multiple inheritance structure in a single inheritance hierarchy. However, the example demonstrates how the extraction of reuse contracts and operators and the occurrence of certain combinations of reuse operators might indicate design flaws. This information can thus form the basis for restructuring the hierarchy, or for refactorings. The relationship between reuse contracts and refactoring will be discussed in section 7.8.



1. The #next method is introduced by class Stream as an abstract method. Classes PeekableStream, PositionableStream and InternalStream leave it untouched.
2. Class WriteStream cancels the method.
3. And class ReadWriteStream introduces it again!

Figure 7.5: Spotting Possible Design Flaws

7.5.3 Existing Design Guidelines

Law of Demeter

The Law of Demeter is a design principle for object-oriented software systems stating that 'An operation O of a class C should only call operations of the classes (called preferred supplier classes) of the following objects: immediate subparts of the current object, argument objects of O (including self), objects created by O'.

This principle was inherently present in the original definition of reuse contracts because it is only allowed to send messages to direct acquaintances. We needed, however, to include the notion of results to which messages can be sent to be able to model existing systems. It became apparent very soon that not that many systems obey this law.

When systems are documented with reuse contracts, the question where in the system the law is severely broken, or in other words, where a lot of messages are sent to acquaintances that are the result of other method invocations is easy to answer. This information is useful in detecting where possible enhancements to the design can be made.

Narrow Inheritance Interface Principle

Another well-known design principle is the *narrow inheritance interface principle* [WGM89] that states 'Behaviour that is spread over several methods in a class should be based on a minimal set of methods which have to be overridden.'[6] This principle closely relates to the use of template methods, which encode crucial behaviour of a class while relying on a number of abstract and/or default methods that can be overridden to adapt the general behaviour.

As discussed before, 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 implies 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 impossible to get template methods in a framework right from the start and a number of iterations is required to improve their usefulness. The optimum balance between flexibility and ease of use will depend on domain-specific adaptations of the framework. The narrow inheritance interface principle expresses how to reconcile these conflicting goals.

The documentation provided by reuse contracts can help in assessing the quality of template methods. As discussed before, through the documentation on the calling structure information on which methods are core methods and which are peripheral can be derived. Template methods will typically be peripheral methods relying on a

---

[6] Lalonde [Lal89] uses the term "design by primitives" to describe the same principle.

number of abstract or default core methods. We can now combine this information with the reuse operators and see, for example, which methods are most often refined, and whether they are refined through participant refinement or specialisation. Participant refinement of peripheral methods might point at template methods that are not flexible enough. A number of layered concretisations of core methods with the same or partially similar specialisation interfaces indicates that different specialisers need to make the same specialisations. This might point at template methods that are not specific enough or at a missing layer of templates.

As an example, consider again the AWT hierarchy. In the previous discussion none of the reuse contracts had abstract methods. The reason for this is that although `Component` and `Container` are declared to be abstract classes, they do not contain any abstract methods. They do, however, contain some methods with an empty implementation. This might be a useful approach in some cases[7]. This is however not always the case. Take for example, the method `layout` in `Component`. This method has an empty implementation as well. However, while `Component` has eight subclasses, layout is only refined in one of them, namely `Container`. This was already discussed in section 7.2. This is logical, as this is the only subclass that has a `LayoutManager` as depicted in figure 7.6. Therefore, it seems logical to remove the method `layout` from `Component`. The consequences of this removal can be easily estimated through the reuse contract of `Component`. This shows that `layout` is called by one method, namely `validate`. Therefore, if we remove `layout` from `Component`, `validate` needs to be reimplemented in `Component` (so that it no longer invokes `layout`) and on `Container` (so that it invokes `layout` there).

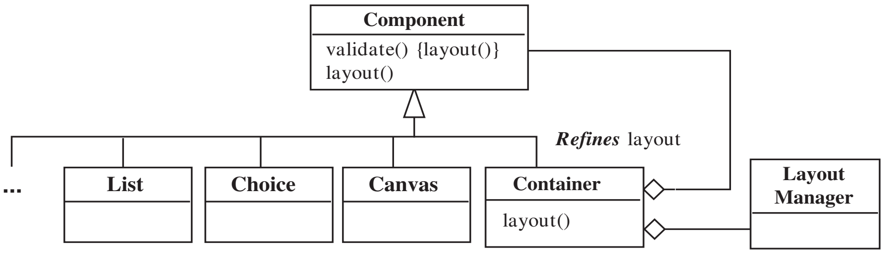


Figure 7.6: Part of the AWT Hierarchy

Note that this is actually a case of inconsistent methods. The method `layout` is refined in `Container`, while a call from `validate` to `layout` is removed on

---

[7] For example, the class `Applet`, also a subclass of `Component`, has five methods with an empty body (`init`, `getAppletInfo`, `start`, `stop` and `destroy`). To make an application developers need to subclass from Applet and these methods are invoked by the system when an applet is started or stopped. Therefore, they need to be executable. Note that here these methods will be refined in many subclasses of `Applet`.

`Component`. As we discussed earlier, the solution to inconsistent methods is to re-implement the method from which the call was removed on the subclass in order to make the behaviour consistent again. This can be done here by simply adding the invocation again; in other cases similar behaviour could be added to the method manually.

7.5.4 Tools for Quality Assessment

A tool can be conceived where a set of the above discussed design flaws can be detected and design heuristics can be measured and the results analysed. A user should be able to mark which guidelines he wants to see checked or which constructs he wants information about. In a further stage, it is conceivable to build a layer on top that allows users to define their own guidelines and heuristics based on reuse contracts and reuse operators and thus measure different aspects, depending on their interest.

7.6 Enforcing Design

Next to assessing the quality of a design it is also possible, by using the compliance checker, to enforce implementors to respect a design that was set out for a system. A design can be expressed using the drawing tool or through the extended interfaces in Java or some other notation. When changes are made by the developer the checker can be called to see whether the implementation respects the original design. Whether this checking is enforced in a coercive way, or more in a guiding way, can be decided depending on the situation.

Enforcing Design Patterns

Reuse contracts can also be used on a more general level. While above we meant checking compliance with a rather specific design, we can also use reuse contracts to check compliance with general design principles. One special way of describing designs is of course by means of design patterns. One can, for example, specify that the way a certain structure in a system is built from subcomponents must follow the Decorator Pattern.

The Decorator Pattern is depicted in figure 7.7. It is used to add responsibilities to individual objects, not to entire classes. It can also be used when extension by subclassing is impractical, because a large number of independent extensions are possible and that can produce an explosion of subclasses to support every combination. Figure 7.7 gives the example from [GHJV94], where a graphical user interface toolkit allows the addition of properties like borders or behaviour like scrolling to any user interface component. To achieve this, components can be embedded in decorators that add these extra functionalities.

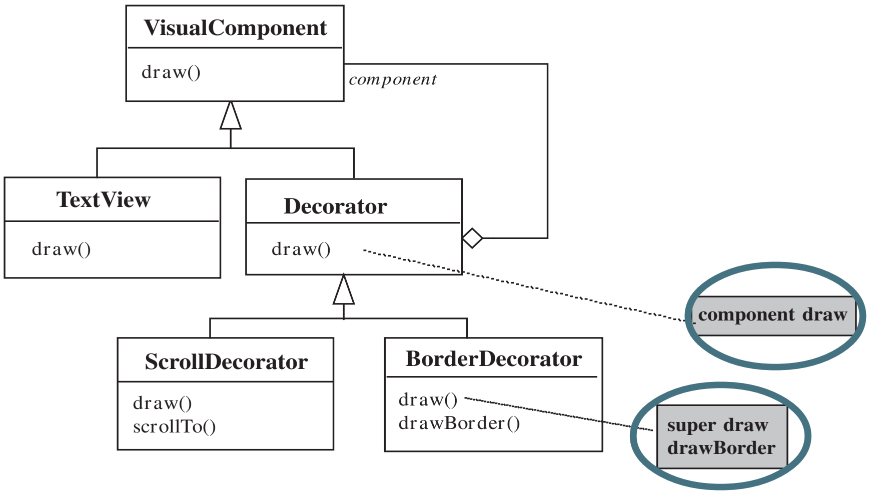


Figure 7.7: The Decorator Design Pattern

The class diagram does not suffice to describe the pattern. Additional explanation is given in textual form. First, the decorator's interface must conform to the interface of the component it decorates. A decorator must forward requests to its component. Decorator subclasses are free to add extra functionality. The decorator may perform this extra functionality before or after forwarding. While this is now explained through natural language and with code fragments, reuse contracts can clarify much of this information.

The reuse contract in figure 7.8 describes the decorator pattern. The specialisation clause of `draw` on `Decorator` denotes that decorators forward their behaviour to their component. This is also denoted by the refinement. The subclasses of Decorator show that special decorators can add extra functionality (denoted by the extensions) and the specialisation clauses of `draw` on these subclasses show that extra functionality can be added but only before or after the forwarding. This is denoted by the specialisation. Furthermore, strict obedience to the reuse contracts implies that only concretisation and refinement are allowed between a Component and its Decorator and therefore, their interface is equal.

Although the example shows that reuse contracts already help in better describing design patterns and can serve as the beginning of a more formal support for design patterns, to be able to fully describe patterns a more generic notation for reuse contracts will be needed. Figure 7.9 sketches such a description of the decorator pattern. Making reuse contracts more generic is future work.

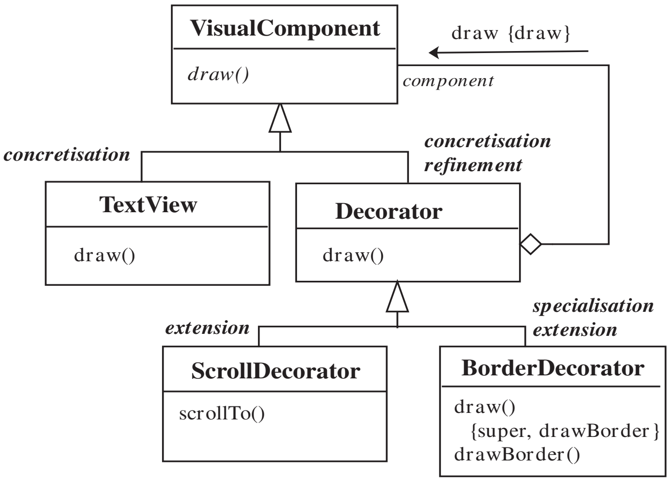


Figure 7.8: Reuse Contracts for Decorator

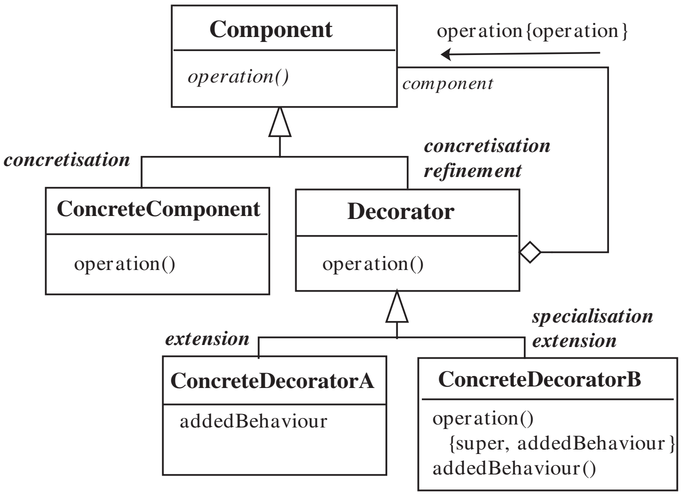


Figure 7.9: Generic Structure of Decorator

7.7 Evolution and Incremental Development

7.7.1 Consistency Checking and Conflict Detection

The previous chapter already demonstrated how reuse operators and the rules based on them can assist in systems evolution and incremental development. As was mentioned in the introduction, the same kinds of problems occur in a number of situations.

- A first situation occurs when applications are built as customisations of a framework or on top of reusable libraries or other components. When this framework or library evolves, we want to be able to upgrade these applications to the new versions. A first motivation is to take advantage of the improvements. Second, and at least as important, not upgrading leads to a proliferation of versions that have to be maintained. This results in a maintenance problem that has grown worse because of the use of reusable components, instead of having improved. This situation is all the more worrying when taking into account that the development of reusable components usually is performed in an iterative fashion. The creation of a stable set of truly reusable components usually requires a number of iterations. The example with the Set and OptimisedSet from chapter 1 was an example of such a kind of evolution.
- When considering problems caused by changes to reused components there is not really a distinction between building a system on top of a framework and the incremental development of a new system. In incremental development one usually starts out with a core of the system and then continues to build layers on top of this core. While building these additional layers deficiencies in the core become apparent. Adapting the core can lead to inconsistencies in all the layers built around it. Of course, there need not always be a certain order of importance among the different parts. The same problems can occur between different interacting layers.
- A third similar situation is consistency checking upon composition as in the local area network example from chapter 1. There one can imagine two components working together and different variations in both kinds of components. Not all variations of one component might work together with all variations of the other component. Therefore, upon composition some support in consistency checking is very useful.

While in the previous chapter the rules were checked by hand, it is clear that this soon becomes infeasible for larger systems. A tool must therefore be constructed to assist developers in assessing where and which problems can occur after changes are made or when carrying out compositions and that helps in deciding how these problems might be solved. The first case, where a new version of a framework or library is incorporated into a system is the easiest, because it is clear where changes occur. If the framework is well-documented by means of reuse contracts, the changes can then be expressed by means of the operators. If the customisers or the developers building on top of the library have clearly documented, again by means of the operators, how they have adapted the framework or library[8], the rules as described in chapter 3 can be checked automatically. This kind of conflict detection not only provides information on locations in a system changes might propagate to. Different situations will break different rules, thus indicating what kind of conflict might occur and making suggestions on how these conflicts can be solved.

In the case of incremental development this is harder because the relationships are less clear cut and the tool should be a constant aid to the developer, watching over his shoulder indicating where problems can arise. It is impossible to check every line of code that is written. Moreover, this would be annoying as during development one usually wants to be able to be in an inconsistent state at certain times. Therefore, some balance has to be found between constant guidance and consistency checks invoked by the developer. It is very important that the developer is able to tune the level of activation.

When considering the incremental development of systems one crucial factor is multi-user development. When several developers simultaneously work on one system, reuse contracts and their rules for consistency checking can be applied on integration. The rules can be integrated in environments for multiple developers that feature versioning. The rules can then be automatically invoked whenever a developer wants to add his code to the system. This way it can be verified both whether his work does not conflict with adaptations made by other developers and whether his work complies with the general design set up for the system.

7.7.2 Traceability

Another use of the reuse operators is to document the evolution of subsequent steps in the analysis, design and implementation of a software system. Choices are made and alternatives are excluded at each step in the development. By documenting where new concepts were added (through extension, refinement, ...) or others omitted (through cancellation, coarsening, ...) it is possible to track back to earlier stages and see how things evolved, when later on it seems that one of the decisions made might not have been so good.

---

[8] Note that this documentation could also have been extracted automatically.

7.8 Re-engineering and Reverse Engineering

7.8.1 Extraction

It is obvious that the extractor tool can be used as a basis for reverse engineering as it extracts information about the actual design of a system. Moreover, as we argued before, the discovery of reuse contracts can help in signalling possible design flaws and can thus form the basis for re-engineering efforts.

7.8.2 Refactoring

Another related field is that of so-called refactoring algorithms. Here one also starts with the observation that a number of iterations is necessary to achieve truly reusable hierarchies. A class collection must therefore evolve to eliminate defects such as inadequate inheritance structures, missing abstraction levels, or overly specialised classes, all of which seriously impair its robustness and its reusability. What's important concerning refactoring algorithms is that they are behaviour preserving. Refactoring tools for amongst others Smalltalk [DR96], C++ [Opd92], Eiffel [Cas94] and Self [Moo96] have been presented.

There are a number of points where reuse contracts and refactoring tools meet:

- First, throughout this chapter we have discussed how reuse contracts can help in detecting flaws in architectures. These diagnoses, which can be made automatically, can form the basis for the detection of possible refactorings. Take for example the AWT hierarchy, where we noticed that part of the top class `Component` handled event handling and was completely independent from the rest of the class. It probably would have been better to introduce a Strategy pattern to factor out the event handling behaviour and split the class `Component` into two classes using an aggregation relationship between them.
- When renaming functions, factorising methods, etc. help from developers is needed. Here reuse contracts can help in explaining to the users what changes are proposed and guiding them in making the decision whether it is a good change or not.
- As the spots for refactorings are detected automatically, based purely on information like names of methods, refactorings do not necessarily enhance the design. Reuse contracts can help in setting up rules to better assess whether or not they do. For example, reuse contracts can be extracted from inheritance trees before and after refactoring.

7.9 Conclusion

Throughout this chapter we have discussed different tasks in software development which reuse contracts can be of assistance in. Most of them require tool support and so we discussed the kinds of tools that can be conceived. In this conclusion, we shortly run through the different stages of the software life cycle and discuss how reuse contracts fit in.

Design of Software Systems In the design phase reuse contracts can be used as a *notation*, for specifying both the structure and behaviour of software systems. By using a drawing tool that is linked to the rest of the environment, further development is simplified.

Documentation Because of the close link between the documentation and the code, the documentation can more easily be kept consistent. This can be supported by the *extractor* and the *compliance checker*. Furthermore, the reuse contract formalism gives far more information on the *structure* of the architectures, the *layering* of protocols, etc. than other methods.

Customisation and Reuse Reuse contracts can be used to *guide* reusers and customisers in this process. Furthermore, because of the *compliance* checker it can be verified whether the customisations comply with the design that was set up before and if necessary this can even be *enforced*.

Iterative Development Code can be developed, starting from a design specified with reuse contracts. When changes are made to this code it can be *checked* and if necessary *enforced* whether the code still complies to the design. More importantly, the rules for *consistency checking* can be applied to see whether different changes will no conflict.

Maintenance and Evolution In the same vein as with iterative development, when changes are made for *maintenance* reasons or on *evolution*, the rules can assist to ensure that these will not conflict with the rest of the system.

Quality Assurance The information obtained by the *extractor* can be used to verify *criteria* about the design and apply *heuristics* to them.

Refactoring and Re-Engineering The information given by the *extractor* can be used to indicate where enhancements to the design are possible. Reuse contracts can also be useful in *guiding* these *adaptations*.

Reverse Engineering For systems that have insufficient or incorrect (or no!) documentation the extractor can help in rediscovering the design.

7.10 Acknowledgements

This chapter would not have been possible without the work of other people in the reuse contracts group. I thank Koen De Hondt, Kim Mens, Tom Mens, Patrick Steyaert and Roel Wuyts for inspiring brainstorms. Koen and Wim Codenie built the first version of the extractor. Koen and Roel carried on working on it, while Roel also built the Clusterviewer. Koen and Patrick documented design patterns with reuse contracts. The extension of Java and the documentation of the AWT were carried out by Gerrit Cornelis as part of his graduation thesis.