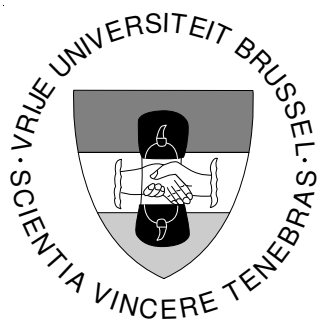
Vrije Universiteit Brussel
Faculteit Wetenschappen



Managing Software Evolution through Reuse Contracts

Carine Lucas, Patrick Steyaert, Kim Mens

Techreport vub-prog-tr-97-01

Programming Technology Lab
PROG(WE)
VUB
Pleinlaan 2
1050 Brussel
BELGIUM

Fax: (+32) 2-629-3525
Tel: (+32) 2-629-3308
Anon. FTP: progftp.vub.ac.be
WWW: progwww.vub.ac.be

Managing Software Evolution through Reuse Contracts

Carine Lucas, Patrick Steyaert, Kim Mens
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
http://progwww.vub.ac.be/
clucas@vnet3.vub.ac.be, prsteyae@vnet3.vub.ac.be, kimmens@is1.vub.ac.be

Abstract

Assessing the impact of changes in one part of a software system on other parts remains one of the most compelling problems in software maintenance. This problem can be relieved by making implicit dependencies between different system parts explicit. We propose to explicitly document the interaction between the different system parts by means of reuse contracts, that can only be manipulated by formal reuse operators. Reuse contracts and their operators facilitate managing the evolution of a software system by indicating how much work is needed to update the system, by pointing out when and which problems might occur and where and how to test and adjust the system.

1. Introduction

Minimisation of dependencies between different parts of a software system is by far the most successful software engineering principle to cope with change and evolution. This principle is the foundation of, amongst others, encapsulation, modularity, high cohesion and loose coupling. It enables reasoning about different system parts separately as well as making changes to certain parts of a system without interfering with the other parts. Details unimportant to other system parts are hidden behind interfaces. As these other parts only rely on the information they get from these interfaces, they are not affected when the structures and implementations behind the interfaces are changed.

While the continuous elaboration on this principle accounts for much of the progress that has been made in software engineering, it can only take us so far. At a certain point in the evolution of a software system, changes occur that cannot be kept local to one system part and thus interfaces do have to be changed as well.

Assessing the impact of such non-local changes remains one of the most compelling problems in the development of software. This can only be dealt with by a careful documentation of dependencies between different system parts. Such a documentation must not only include *which* parts depend on what other parts, but more importantly *how* they depend on each other. The former gives an indication on where problems might arise upon change; the latter provides us information on what the problem is (and thus on how it can be solved). The lack of this kind of documentation is a major impediment to the management of evolving software systems with current methodologies.

We have studied how changes to reusable assets propagate in the systems built on them. We propose to document the interaction between designers and users of reusable assets by means of *reuse contracts*. These provide the adequate interface for possible reusers of an asset. Moreover, reuse contracts can only be adapted through a fixed set of formal *reuse operators*: extension, refinement and concretisation and their inverse operators: cancellation, coarsening and abstraction. Reuse contracts thus not only document how an asset *can* be reused, but also how and why the asset *is* actually reused by others. When a system evolves, this information allows forecasting where and how the system should be tested, which problems might occur and how the system should be adjusted. Reuse contracts thus facilitate managing the evolution of a software system.

This paper briefly discusses the most important features of reuse contracts on class hierarchies (section 2) and multi-class components (section 3). Currently, reuse contracts for state transition diagrams, black-box components and analysis models are also under development and early results endorse our claim that this approach is applicable to other and more general reuse mechanisms. In general, the development of reuse contracts for a certain domain involves a number of steps. First, it is decided what kind of documentation should be included in the reuse contracts and an intuitive notation is defined for it. Generally, we annotate the notations from known methods (e.g., OMT, UML, Booch) with extra information. Second, the reuse operators for these reuse

contracts are defined, and third rules — based on reuse contracts and their operators — are developed that can detect conflicts when software evolves.

2. Managing Evolving Class Hierarchies

The use of abstract classes with inheritance as reuse mechanism is undoubtedly the best-known technique available today for structuring and adapting object-oriented software. Therefore, we first focus on the problem of evolving class hierarchies as a tangible case to investigate the concept of reuse contracts. In that context, reuse contracts and their operators describe the protocol between managers and users of (abstract) classes. Reuse contracts of abstract classes provide an explicit representation of the important design decisions behind an abstract class, including information such as: which methods can be sent to the class, which methods should be invoked by what other methods, which methods are abstract or concrete, … Only information relevant to the design is included. For example, auxiliary or implementation-specific methods are purposefully omitted from the reuse contract.

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 on which methods depend on what other methods, in order to decide which methods need to be overridden. For example, if we know that `addAll` depends on `add` in its implementation, it is sufficient to override the method `add` to perform the counting. Reuse contracts for classes document exactly such dependencies. In such reuse contracts each method has a specialisation clause (between curly braces in the example below) that documents on which other methods from this reuse contract a method depends (as in Lampings specialisation interfaces [2]). The reuse contract is an interface description to which the implementation must comply. It provides information that is typically not included in other methodologies. Figure 1 shows the reuse contract for `Set`.

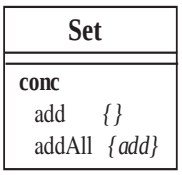


Figure 1. The Set reuse contract

Reuse contracts can only be manipulated by means of reuse operators. *Concretisation* makes abstract methods concrete, *extension* adds new methods to a reuse contract and *refinement* refines the design of some methods by adding extra methods to their specialisation clause. These reuse operators not only allow documenting the changes made to a class, but a careful investigation of their interactions also allows to predict and manage the effect of these changes. Suppose 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. This leads to inconsistent behaviour in `CountableSet` when `Set` is replaced by `OptimisedSet`. Not all additions will be counted, because the implicit assumption made by `CountableSet`, that `addAll` invokes `add`, is broken in `OptimisedSet`. Using the terminology of [1] we say that `addAll` and `add` have become inconsistent. 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. Reuse contracts and their operators provide the necessary information by making the *assumptions* (`CountableSet` assumes that in `Set` `addAll` relies on `Add`) and *intentions* (`CountableSet` intends to make a refinement of `Set`) made by adapters *explicit*. In the example below, the reuse contracts of `CountableSet` and `OptimisedSet` document how they were derived from `Set`, and thus what assumptions about `Set` they rely on.

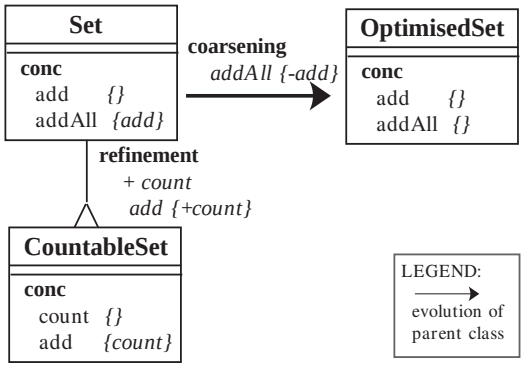


Figure 2. Inconsistent Methods

`OptimisedSet` is a coarsening (the inverse of a refinement) of `Set`, which means that it partially breaches `Set`'s design. This is done by removing a method from one of its specialisation clauses. `CountableSet` is a refinement of `Set`, as it adds a new method and refines one of the existing methods' design, by extending its specialisation clause with this new method.

We have made an extensive study of possible conflicts when making changes to parent classes and created a set of rules that allow automatic detection of conflicts based on

the interaction of reuse operators (see also section 4). For a complete discussion we refer to [3]. As an example, one rule says that inconsistent methods occur when a method, that was modified by the inheritor, is removed from the specialisation clause of the exchanged parent by a coarsening.

3. Managing Evolution of Collaborating Classes

A first elaboration on reuse contracts for class hierarchies is the extension of reuse contracts to include information on interclass interactions. To be able to reuse larger components composed of more than one class, information is also needed on which methods rely on which other methods from the acquainted classes.

Consider the example of a software simulation of a local area network (LAN). The LAN exists out of a series of connected nodes, that pass packets to each other. The general behaviour of passing the packets and handling them is described by the PacketHandling reuse contract as depicted in figure 3.

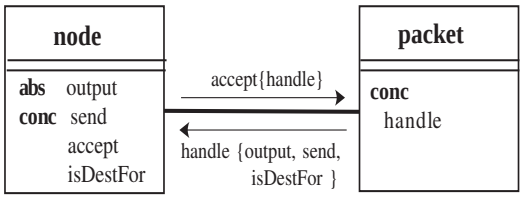


Figure 3. The Packet Handling contract

This contract describes that when a node receives a new packet through the message `accept` it sends the message `handle` to the packet. This is depicted by the arrow along the binding between node and packet. These arrows represent an extension of the specialisation clauses from above, that not only show which methods are sent to self, but also which are sent to other participants. The reaction of the packet is to send the message `isDestinationFor` back to the node, to check whether this node is the destination (or one of the destinations) of this particular packet. If it is, it sends the message `output` to the node to perform some action with the packet, if not it sends the message `send` that will pass the packet on to the next node. This reuse contract describes the general behaviour of the LAN. Different kinds of nodes and packets can then be created that comply to this contract and perform a particular behaviour by giving a particular implementation to `output` or `handle`. For example, special kinds of nodes could be printservers, where `output` prints the contents of the packet, and fileservers, where `output` saves the contents of the packet. Special kinds of packets could be broadcastpackets, that are to be handled by every node. In other words, these reuse contracts describe how different parts of a system, here nodes and packets, can work together. They thus document the design of the PacketHandling mini-framework. Moreover, they can be used to check that this design is adhered to.

Consider introducing gateways, as a new kind of node. When a gateway receives the message `isDestinationFor` from a packet, it will check the domain name. When this name is correct it will return yes and receive the message output, which sends the packet through to the particular domain. `Gateway` thus complies to the PacketHandling reuse contract. This implies that Gateway can be used in combination with all kinds of packets as long as these comply in their turn to the contract. Consider, for example, introducing a `VisitorPacket`. Such a packet has to be passed to all the nodes in the LAN to perform a particular behaviour. It could, for example, count the number of printers in the network. To achieve this `VisitorPacket` re-implements `handle`, so that it always sends the message `send` to node to be passed through. As `handle` on `VisitorPacket` no longer has `output` in its specialisation clause it does not comply to `PacketHandling`. This is exactly what we would want to detect, as using `VisitorPacket` and `Gateway` together would cause a problem. `Gateway` uses the method `output` to send a packet through to the right subnetwork. As it will never receive this message from `VisitorPacket`, a `VisitorPacket` would never get passed the `Gateway`.

So, on the one hand, a better documentation already assists in managing evolution, by enabling to detect whether newly added comoponents will work together correctly with the framework. On the other hand, again the reuse operators concretisation, extension and refinement and their inverses can be defined to describe how the different kinds of reuse contracts are derived from each other. Instead of just describing what happens to methods and their (now extended) specialisation clauses, the operators will now also consider the addition of new acquaintances etc. These operators can then again be used to discover conflicts upon change.

4. Features of the Formal Model

In the previous sections reuse contracts, with their operators and associated rules for conflict detection were informally discussed. Although space limitations prohibit a complete technical discussion of the methodology and formal model of reuse contracts, some important features of the general model are summarised and motivated here.

In general, reuse contracts provide an explicit description of the important design aspects of some reusable asset. A remark must be made about the kind of information that reuse contracts provide. The specialisation clauses

discussed above describe method dependencies purely by name. This could be extended by including type information or by including semantic information, that specifies, for example, the order in which methods should be invoked. The art is in finding the right balance between descriptions that are easily understood and expressed, and descriptions that capture enough of the semantics of the system part it describes. Understandability and simplicity of reuse contracts is important for asset reusers, and makes the development of automated support tools for reuse contracts feasible. On the other hand, reuse contracts should be detailed enough to be able to detect the most important conflicts when assets evolve. For example, the same (and more) conflicts could be detected using pre- and post-conditions, but the required conditions often tend to be so low-level and technical, that their intuitive meaning is either lost or hidden behind the details. With reuse contracts we try to explore the boundaries of how far we can get in detecting behavioural problems by considering only structural information. For example, although reuse contracts for abstract classes essentially provide information only on the calling structure, a number of important behavioural conflicts when changing a parent class can be detected

Reuse contracts can only be manipulated through a small number of elementary reuse operators. Each reuse operator corresponds to changing a particular aspect of the contract. A detailed study of the interactions between the elementary reuse operators provides a categorisation of possible conflicts that can occur when a reuse contract is changed. Based on this categorisation, formal rules can be defined to automatically detect conflicts when a reuse contract is adapted. An informal example of such a rule was given at the end of section 2. A prototype version of a detection tool based on such rules has been implemented in PROLOG.

For assets that have not been documented by means of reuse contracts, tools can be constructed that semi-automatically extract this information from the assets. The programmer only has to delete the implementation-specific parts of the extracted documentation, as reuse contracts should include only information relevant to the design. Once the different reuse contracts have been extracted, the tool can easily compute how the reuse contracts are related to one another by means of elementary reuse operators. A prototype implementation of such a tool for Smalltalk classes has been implemented. Similar to the extractor, tools can be conceived that assist in the synchronisation of reuse contracts and their corresponding implementations

Because the elementary operators are too low level, more intuitive reuse operators — which are a combination of more elementary operators — are needed. An operation on abstract classes that is often needed is making a method abstraction: i.e. when a method m makes a call to a method n, the call to m can be replaced by a call to a newly introduced auxiliary method, which in turn calls n. This intuitive operator can be defined as a combination of the elementary operators extension (introducing the auxiliary method), refinement (adding a call to the auxiliary method) and coarsening (removing the original call from m to n).

Conflict detection of the intuitive reuse operators — or in general, on more complex adaptations, described by multiple operators — can then be defined in terms of conflict detection of elementary reuse operators. However, not every conflict at the lower level is actually a conflict at the more intuitive level. It is possible that some conflicts cancel each other. To see this, the intuition of the considered operator needs to be taken into account. For example, a method abstraction is defined in terms of a coarsening, and a coarsening might introduce an inconsistent method conflict (as explained at the end of section 2). However, we know that this is not really a conflict, because the coarsened method is still called indirectly by the newly introduced auxiliary method. It is possible to define extra rules such that this kind of information is also taken into account.

5. Conclusion

Current methodological and tool support for managing the evolution of large, long-lived software systems, focuses mainly on minimising dependencies between system parts. However, the question what happens when existing dependencies are changed at some point during the evolution process is largely neglected. Documenting these dependencies by means of reuse contracts and reuse operators allows us to signal such changes and to assess their impact. Moreover, reuse contracts have a broader scope than managing change in evolving systems: they shed light on the architecture of a system, can be used as structured documentation and can generally assist software engineers in adapting systems to particular requirements. When adopted, reuse contracts may significantly enhance the way in which software is being built and managed.

References

[1] G. Kiczales and J. Lamping. Issues in the design and specification of class libraries. In *Proceedings of OOPSLA '92, Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 435–451. ACM Press, 1992.

[2] J. Lamping. Typing the specialisation interface. In *Proceedings of OOPSLA '93, Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 201–215. ACM Press, 1993.

[3] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of OOPSLA '96, Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 268–285. ACM Press, 1996.