Vrije Universiteit Brussel
Faculteit Wetenschappen

Documenting Evolving Software Systems through Reuse Contracts

Kim Mens, Patrick Steyaert, Carine Lucas

Techreport vub-prog-tr-96-12

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

Documenting Evolving Software Systems through Reuse Contracts

**Submitted to the OOPSLA'96 Workshop on
"Object-Oriented Software Evolution and Re-Engineering"**

Kim Mens, Patrick Steyaert, Carine Lucas

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

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

Introduction

Minimisation of dependencies between the 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 that are of no importance to other parts of the system 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 occur 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 building reusable software with current methodologies.

We propose to document the protocol between designers of different parts of a system by means of *reuse contracts*. Reuse contracts not only document how a system part *can* be reused, but also how and why the part *is* actually reused by other parts. Just as real world contracts can be extended, amended and customised, reuse contracts are subject to parallel changes encoded by formal *reuse operators*: *extension*, *refinement* and *concretisation*. The inverse operators: *coarsening*, *cancellation* and *abstraction* intuitively correspond to the (partial) breaching of a contract.

Reuse contracts together with their operators facilitate managing the evolution of a software system by indicating how much work is needed to update the system, by forecasting when and which problems might occur, and by providing information on where and how to test and adjust the system.

Managing Parent Class Exchange

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, in [Steyaert&al96] we focused on the problem of evolution of class-hierarchies as a more tangible case to explain the ideas behind reuse contracts. In that context, reuse contracts and their operators describe the protocol between managers and users of (abstract) class libraries. Reuse contracts of abstract classes provide an explicit representation of the design decisions behind an abstract class, including information such as: which methods can be sent to the class, which methods are invoked by what other methods, which methods are abstract or concrete, relationships with other classes, ... Only information relevant to the design is included. For example, auxiliary or implementation-specific methods are not mentioned in a reuse contract.

Reuse contracts can be manipulated by means of reuse operators. Refinement refines the design of some methods, extension adds new methods, concretisation makes abstract methods concrete. These reuse operators not only allow documenting the changes (and the intentions of these changes) made to a class, but a careful investigation of their interactions also allows to predict and manage the effect of these changes.

Consider the example of a Collection hierarchy. A class `Set` defines a method `add` and a method `addAll` to add a collection of elements simultaneously.

```
Class Set
   method   add(Element) = 0
   method   addAll(aSet:Set) =
     begin
      for e in aSet do
         self.add(e)
     end
end
```

In order to decide which methods need to be overridden 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. For example, if we know that `addAll` depends in its implementation on `add`, it is sufficient to override the method `add` to take counting into account. Reuse contracts for classes document these dependencies. In a reuse contract each method has a specialisation clause (in italics in the example) that documents how it depends on the other methods from this reuse contract (as in Lamping's specialisation interfaces [Lamping93]). The reuse contract is an interface description to which the implementation must comply. It provides information that is typically not included in other methodologies.

```
reuse contract Set
   abstract
      add(Element)
   concrete
      addAll(Set) {add(Element)}
end
```

Reuse contracts in their current form only document the internal dependencies among a class's methods. Part of our future work is studying how reuse contracts can be extended to include interclass dependencies as well. Other experiments are being conducted to include information on state and state transformations in reuse contracts and operators.

Reuse contracts can be used to assess the impact of changes or updates to system parts. 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 it decides to upgrade `OptimisedSet`; not all additions will be counted. This is because the assumption made by `CountableSet` that `addAll` invokes `add` is broken in `OptimisedSet`. Using the terminology of [Kiczales&Lamping92] we say that `addAll` and `add` have become *inconsistent methods*. Although in this simple example the conflict can easily be derived from the code, in practice it should be possible to detect such conflicts without code inspection. The major obstacle for locating problems such as inconsistent methods is that the different conceptual ways to reuse an (abstract) class are all performed by the same operator, i.e. inheritance. More information about the intentions of inheritor is needed. This kind of information is precisely provided by the reuse operators on reuse contracts. In the example the reuse contract of `CountableSet` and `OptimisedSet` document how they were derived from `Set`.

```
reuse contract CountableSet concretises Set
   concrete
      add(Element)
end

reuse contract OptimisedSet coarsens Set
   concrete
      addAll(Element) {-add(Element)}
end
```

The fact that `add` and `addAll` have become inconsistent can be derived directly from the reuse contracts: `CountableSet` concretises a method that has been removed from the specialisation clause while changing from the old parent class to the new parent class (in italics above).

For a more complete set of possible conflicts on parent class exchange we refer to [Steyaert&al96]. Furthermore, it gives a complete set of reuse operators together with a set of rules that allow automatic detection of conflicts based on the interaction of reuse operators.

Environment and Tool Support for Reuse Contracts

An environment for managing software evolution based on the concept of reuse contracts should include tool support for assessing the impact of making changes to a system by signalling possible problems that (might) occur. A prototype version of such a tool has been implemented in PROLOG.

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

Finally, for software systems that have not been documented by means of reuse contracts, tools can be constructed that semi-automatically extract this documentation from the code, based on the calling structure. 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 corresponding to the different parts of the system are related to one another by means of reuse operators. A prototype implementation of such a tool for Smalltalk classes has been implemented.

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 these 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. Many tools to support the use of reuse contracts for managing software evolution can be conceived. When adopted, reuse contracts may significantly enhance the way in which software is being built and managed.

References

[Kiczales&Lamping92] G. Kiczales, J. Lamping: *Issues in the Design and Specification of Class Libraries*, Proceedings of OOPSLA '92, Conference on Object-Oriented Programming, Systems, Languages and Applications, pp. 435-451, ACM Press, 1992.

[Lamping93] J. Lamping: *Typing the Specialisation Interface*, Proceedings of OOPSLA '93, Conference on Object-Oriented Programming, Systems, Languages and Applications, pp. 201-215, ACM Press, 1993.

[Steyaert&al.96] P.Steyaert, C.Lucas, K.Mens, T.D'Hondt: *Reuse Contracts: Managing the Evolution of Reusable Assets,* To appear in Proceedings of OOPSLA‘96 Conference on Object Oriented Programming, Systems, Languages and Applications, ACM Press 1996.