Reuse Contracts: Managing Evolution in Adaptable Systems.

Kim Mens, Patrick Steyaert, Carine Lucas
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
http://progwww.vub.ac.be/

Abstract

Adaptable systems are often described as being composed of a persistent part that remains more or less stable throughout the evolution of the system and of more transient parts which are derived from the persistent part. Current research on adaptability seems to be biased towards adaptable systems of which the persistent part does not evolve at all. We argue that the issue of managing the evolution of the persistent part itself is not only a central issue in the practical application of adaptability, but is a key question in the understanding of adaptability: although changes to the persistent part occur only occasionally, most often these changes have a very large impact on the rest of the system; only very controlled changes guarantee that the rest of the system will remain more or less consistent. We propose to manage these changes by means of reuse contracts *and* reuse operators*. Rather than having a template rigidly enforced by the builder of the adaptable system, with reuse contracts a reuser declares on which parts of the adaptable system he relies, what parts of the design he respects and what parts he violates. This enables evolution of the adaptable system, reuse of adaptations to different adaptable systems, and a more layered approach to adaptation.*

1 Introduction

Adaptable systems are systems that can easily be adapted to a steady change of various requirements. Summarising [5], they do so by capturing the persistent part that remains stable throughout the evolution of a system and from which more transient versions of the system can be derived. The persistent part is important; without it we would not be able to speak of a “system” but rather of nothing more than a collection of program fragments that can only be re-used in an ad-hoc fashion. The persistent part can be expressed by different means: in object-oriented frameworks by means of abstract classes and class collaborations [3], in adaptive software by means of propagation patterns and succinct sub-graph specifications [4], in the field of software architecture by means of components and connectors [1], etc. Accordingly, transient versions of an adaptable system are derived by different mechanisms: in object-oriented frameworks by means of inheritance (preferably some controlled form of inheritance) and filling in polymorphic parameters, in adaptive systems by applying a propagation pattern to some class hierarchy, in software architectures by means of refinement, etc. In general, the persistent part expresses an overall architecture or architectural design that is used as a "template" for more transient versions of the system. The degree to which this template enforces a set of design or architectural constraints that must be respected by the different incarnations of an adaptable system differs.

Current research on adaptability seems to be biased towards adaptable systems of which the persistent part does not evolve (hence, obviously the label "persistent"). Nevertheless, as complex software systems have to cope with a steady change of various requirements from the moment they are conceived, many iterations are needed before the core of an (adaptable) software system becomes more or less stable and the system becomes truly adaptable. Therefore, it is important to examine the uninvestigated question of how an adaptable system itself evolves over time, how this affects the already derived applications, and how this evolution can be managed.

While we do agree that issues such as describing and enforcing architectures must eventually be answered, it is our conjecture that the issue of manageable evolution of adaptable systems themselves is not only a central issue in the practical application of adaptability, but is a key question in the understanding of adaptability.

2 Characteristics of Evolving Adaptable Systems

Changes to the persistent part of an adaptable system might be required when errors in the initial requirements need to be fixed, or when the requirements change because users want to extend the system to incorporate functionality it was not originally designed for. More importantly, iterations over the persistent part are inherent to the development of adaptable systems. While some of the basic identities of an adaptable system may be discovered early in the development of the system, these identities will change and improve as a system evolves. The persistent part of a truly adaptable system emerges as a result of such an evolutionary process. Moreover, the usefulness and thus the true degree of adaptability can only become clear by actually adapting the system, in other words by using it to build applications. As it is not feasible to build this kind of applications solely for test purposes, the issue of managing the impact of changes to the evolving persistent part to those already existing applications is crucial.

So, even the "persistent" part of an adaptable system is not completely stable but will need to be changed from time to time. Instead of being composed of one persistent core[1] and a transient shell, we view an adaptable system as a continuum ranging from well-designed and fairly persistent part to very unstable and less

[1]. Actually, this is an oversimplification. In practice, an adaptable system will not have just one core, but several ones which each have more transient parts based on them.

meticulous designed parts that are undergoing changes all the time. This is depicted in figure 1.

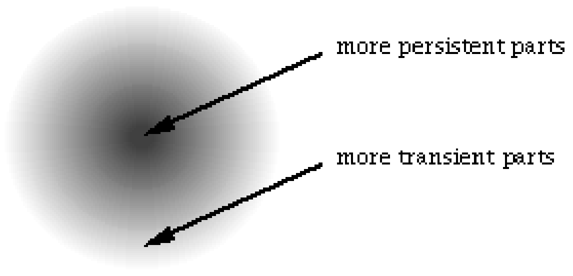

Figure 1. An Adaptable System

Changes to the more persistent parts only occur occasionally, but most often have a very large impact on the rest of the system; only very controlled changes guarantee that the rest of the system will remain more or less consistent. Changes to the outer parts of the system occur much more frequently but do not have such a large impact. This is illustrated in figure 2.

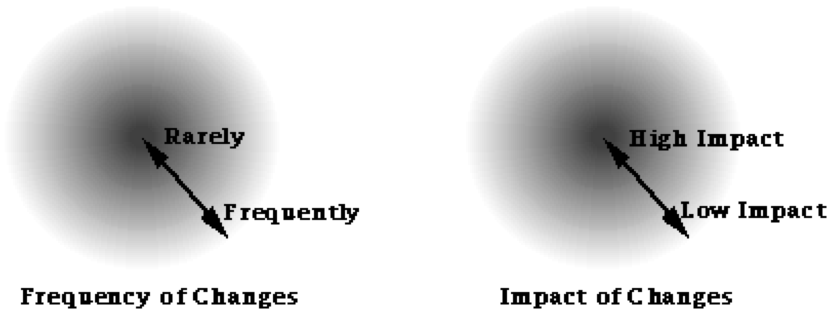

Figure 2. Characteristics of an Adaptable System

3 Reuse Contracts

In [7] we propose *reuse contracts* as a means to codify the management of change in an (adaptable) software system. Reuse contracts record the protocol between builders and customisers of an adaptable system and offer guidelines for deriving more transient versions of the system in some problem domain. Similar to real world contracts that can be amended, extended, or customised, reuse contracts are subject to typical *reuse operations* such as refinement, extension and concretisation. The inverse operations: coarsening, cancellation and abstraction intuitively correspond to the (partial) breaching of a contract.

Reuse contracts and their operations are used to "document" how a transient version has been derived from the more persistent parts of the system. This "documentation" can be used by tools to asses the impact of changes made to the system, to forecast when and which problems might occur and to give directions on where and how to test the derived transient version of a system. For example, when extending the adaptable system with new functionality, collisions with already made extensions in more transient versions must be checked; or, when cancelling functionality in the system, it needs to be checked whether no transient versions rely on such functionality. To be able to check this, more information is needed on how a transient version relies on the design decisions made in the adaptable system. Reuse contracts provide exactly this information.

Because the best-known technique available today for structuring and adapting object-oriented software is undoubtedly the use of abstract classes with inheritance as the reuse mechanism, in [7] we focused on the problem of adaptation of class-hierarchies as a more tangible case to express the ideas behind reuse contracts. In that context, reuse contracts and their operations 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 operations not only allow documenting the adaptations made to a class, but a careful investigation of their interactions also allows to predict and manage the effect of these adaptations.

As an example, let us explain how reuse contracts can help in detecting the problem of *method capture*. Suppose we want to adapt a class library by substituting some parent class with another class[1] with a seemingly similar behaviour. Methods from inheritors that were not invoked by methods in the former parent could be invoked now (and vice versa). This situation is called *method capture* (the opposite case leads to what is called *inconsistent methods* in [3]) and often causes problems on a behavioural level. Reuse contract can help in detecting such problem situations because they explicitly provide information on which methods invoke which other methods in a class. Without this documentation, programmers of subclasses are forced to inspect all implementation details of the parent classes in order to assess the effect of overriding a method. However, the major obstacle for locating problems

[1]. or by editing an existing parent c lass.

such as method capture (and inconsistent methods) is that the different conceptual ways to reuse an abstract class are all performed by the same operation, i.e. inheritance. More information about the intentions of inheritors are needed. Consider exchanging a parent class with a new parent class that introduces a new method `m`. When looking at plain inheritors it is not always clear whether a method with the name `m` in an inheritor was intended to *override* a method `m` of the parent class or whether it was intended as a *new* method. The second case would imply that after the parent class exchange this method is unintendedly captured, and thus would probably cause some behavioural conflict. If the inheritor would explicitly declare that he intended to perform an extension (i.e. introducing a new method), it would be easier to detect possible problems. This kind of information is precisely provided by the reuse operations on reuse contracts.

For more a detailed discussion and for more examples on reuse contracts, reuse operators, and on how they can detect and solve problems such as the above, we refer to [7].

4 Evaluation and Conclusion

The primary goal of reuse contracts is a very practical one. They are useful in managing changes to an adaptable system from which different applications have already been derived. It has already been stated in different sources that the absence of mechanisms to manage such changes is an important inhibitor to successful reuse [2,6,8].

Moreover, we believe that evolving adaptable systems and reuse contracts shed a new light on adaptability. Adaptable systems that define an unchangeable template that is enforced in all transient versions are too rigid as a model. Not only because this makes adaptable systems closed systems as they can only be adapted in a limited way, but also because in many cases adapters need to make exceptions to the rigidly defined architecture (most often for tuning the implementation [3]). Current research on reuse and adaptability is also biased towards systems that make a clear-cut distinction between the system builder and the system reusers. This view is limited, especially when talking about object-oriented systems. In a class library for example, classes go through different layers of adaptation. An initial abstract class is transformed through a chain of inheritors, each adding its own layer of adaptations. An essential problem here is that the persistent template that was clear at the top (i.e. at the abstract class) gets modified and becomes unclear at the bottom where classes are actually reused. This is perceived by many users of object-oriented class libraries and frameworks as one of the major obstacles to reuse them.

A model of adaptable systems based on reuse contracts can provide a more smooth transition between the persistent template and more transient versions. Reuse contracts complement information on how an adaptable system is designed (and can be reused) with information on how such an adaptable system *is* actually reused.

Rather than having a template rigidly enforced by the builder of the adaptable system, with reuse contracts a reuser declares on which parts of the adaptable system he relies, what parts of the design he respects and what parts he violates. This enables evolution of the adaptable system, reuse of adaptations in different adaptable systems, and a more layered approach to adaptation.

References

1.Garlan, D. and Shaw, M.: *An Introduction to the Field of Software Architecture* , In Ambriola, V. and Tortora, G., eds., *Advances in Software Engineering and Knowledge Engineering* , volume I. World Scientific Publishing Company, 1995.

2.Goldberg, A. and Rubin, K.S.: *Succeeding with Objects Decision Frameworks for Project Management*, ISBN 0-201-62878-3, Addison-Wesley Publishing Company, Inc., 1995.

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

4.Lieberherr, K.J.: *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, ISBN 0-534-94602-X, PWS Publishing Company, Boston, 1996.

5.Lieberherr, K.J.: *From Transience to Persistence in Object Oriented Programming: Architecture and Patterns*, Position Statement for the MIT Working Group on Object Oriented Programming within the ACM Workshop on Strategic Directions in Computing Research (June 1996) and for the ECOOP Workshop on Adaptability in Object Oriented Software (July 1996).

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

7.Steyaert, P., Lucas, C., Mens, K. and D’Hondt, T.: *Reuse Contracts: Managing the Evolution of Reusable Assets,* Proceedings of OOPSLA‘96 Conference on Object Oriented Programming, Systems, Languages and Applications, pp. 268-285, ACM Press 1996.

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