Assessing the Architectural Quality of Evolvable Systems

Submitted to the ECOOP '98 Workshop on Techniques, Tools and Formalisms for capturing and assessing Architectural Quality in Object-Oriented Software

Tom Mens tommens@vub.ac.be

Kim Mens kimmens@vub.ac.be

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, BELGIUM
http://progwww.vub.ac.be/pools/rcs/

Abstract. *Object-oriented software systems have a natural tendency to evolve. This can be due to many different reasons such as changing requirements, software maintenance, etcetera. Current trends even seem to indicate that continuously evolving systems will become the norm for most organisations. To cope with this fact, architectures and designs should be made as adaptable and reusable as possible. This is the only way to facilitate the evolution and maintenance of software systems, and to avoid them turning into legacy systems. We propose the reuse contract formalism as a methodology to make architectures and designs more reusable.*

Introduction

It is clear that the quality of an object-oriented architecture or design can be described by a set of important characteristics such as modularity, extensibility, flexibility, adaptability, understandability and reusability. Each of these characteristics is recognised to facilitate the evolution and maintenance of software systems. However, rather than measuring (either qualitatively or quantitatively) and improving these characteristics, and thus indirectly improving the evolutionary aspects of software systems, we propose to take a more direct approach by trying to address the evolutional aspects of software directly.

Object-oriented software systems have a natural tendency to evolve. This can be due to many reasons:

- changing requirements
- adoption of new technology
- software maintenance and bug fixing
- to increase the performance of the software
- use of software beyond its original goals
- new insights in the problem domain
- new design insights

Current trends seem to indicate that continuously evolving systems will become the norm for most organisations [Booch98]. To cope with this fact, object-oriented software systems, and more specifically their architecture and design, should be made as adaptable and reusable as possible. This is the only way to facilitate the evolution and maintenance of software systems, and to avoid them turning into legacy systems. Indeed, the main difference between legacy systems and reusable systems is their ability to change!

In order to make systems more reusable, we need to come up with a methodology and notation that supports the process of change in software systems. To this extent, the Programming Technology Lab of the Vrije Universiteit Brussel has invented the *reuse contract* formalism ([Lucas97], [Mens&al98], [Mens&al98b], [Steyaert&al96]). This formalism allows us to deal with reuse and evolution in a disciplined way, which makes it easier to detect anomalies in reusable designs, and facilitates assessing the impact of changes in software.

Problems with evolving systems

The main problem with evolving systems in software engineering can best be compared with the law of entropy in physics: given enough time, any system will eventually drift into chaos. This problem is often referred to as the problem of *architectural drift*. Another term that is frequently coined in this context is *software ageing*. As stated nicely by Parnas:

> "Programs, like people, get old. We can't prevent ageing, but we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable." [Parnas94]

Again, there are several reasons why software programs are ageing:

- lack of movement
- ignorant surgery and architectural erosion
- inflexibility from the start
- inadequate documentation
- deadline pressure
- "not invented here" syndrome

In order to stop or even reverse the effects of ageing, and to address the problem of architectural drift, software architectures need to take evolution into account. The main problem with current systems is that their architecture is defined in a manner that is much too static:

- A *system* is usually defined as a set of communicating parts or components that fit together to realise the behaviour requested by the functional requirements.
- A *part* or *component* is understood to be a systems building block that is defined by its external interface (like an ADT).
- A *software architecture* is a specification of a set of components and a communication pattern or protocol among them to achieve the desired behaviour.

None of these definitions deal with the fact that any software architecture is inevitably subject to change! Because of this, there are many problems when the architecture does change. Without going into details, some of these problems are enumerated below:

- With *improper reuse techniques* like e.g. direct code editing, the original version of the software gets lost, and there is no way to make the changes undone.
- When doing so-called *copy-and-paste reuse*, each time a change is made to a component, a new version is created. Ultimately this leads to a *proliferation of different versions*, and it becomes very difficult to find out which version is more appropriate for a specific situation.
- *Architectural drift* occurs when the software is modified without paying attention to the architectural requirements. This can be due to many reasons such as time pressure, ignorance, etcetera. As a result, the software tends to drift away from its original architecture, ultimately leading to legacy code.

- *Ripple effect* (butterfly effect): small changes in one place can sometimes lead to enormous changes in totally different places of the software. Complicated impact analysis techniques are needed to investigate when this will be the case.

In order to solve these problems, evolution should be dealt with in a more disciplined way. For this reason, we have developed the *reuse contract* formalism at the Programming Technology Lab.

Reuse contracts

The essential idea behind reuse contracts is that a component is reused on the basis of an explicit contract between the *provider* of the component and a *reuser* that modifies this component. The purpose of a contract is to make reuse and evolution more disciplined. For this purpose, both the provider and the reuser have *contractual obligations*. The primary obligation of the provider is to document how the component can be reused. The reuser needs to document how the component is reused or how the component evolves. Both the provider's and reuser's documentation must be in a form that allows to detect what the impact of changes is, and what actions the reuser must undertake to "upgrade" if a certain component has evolved. To summarise we can say that reuse contracts help in keeping the model of the provider consistent with the model of the reuser.

Before the process of evolution can be documented, the provider needs to document what properties of the component can be relied on at a particular point in time. The *provider clause* states certain properties of the operations in the provided component. The reuser documents the changes made to the component in the *reuser clause*. The *contract type* expresses *how* the provided component is reused. Possible contract types include extension, cancellation, refinement and coarsening. The contract type imposes obligations, permissions and prohibitions onto the reuser. For example, the extension contract type obliges reusers to add new operations, but prohibits overriding of existing operations. It permits adding multiple operations at once. Contract types and the obligations, permissions and prohibitions they impose are fundamental to disciplined reuse, as they are the basis for detecting conflicts when provided components evolve.

In this position paper, we will not go into details about the reuse contract formalism. For more information we refer to the literature on reuse contracts. Originally, reuse contracts were used at implementation level to express reuse in evolvable class inheritance hierarchies [Steyaert&al96]. In her PhD thesis, Carine Lucas extended this idea to deal with reuse and evolution of collaborating classes [Lucas97]. Practical experiences with this work were reported in [Codenie&al97] and [Mens&al97] in the context of object-oriented application frameworks. More recent work involved expressing reuse and evolution at analysis and design level, and incorporating the reuse contract formalism into the Unified Modelling Language ([Mens&al98], [Mens&al98b], [Rational97]). Some research has also been done in trying to find a general underlying foundation independent of the domain to which reuse contracts are applied [Mens98]. This will facilitate the work when trying to apply the reuse contract formalism in new domains, like e.g. evolution of object-oriented software architectures.

Workshop Topics

Although the benefits of reuse contracts should already be intuitively clear, in this section we explain into more detail how the reuse contracts methodology fits into the topics addressed by the workshop. More specifically we show some useful results regarding the issue of capturing and assessing architectural quality.

Detecting anomalies in object-oriented design

Although it is necessary to address the topic of evolution of reusable components, component evolution also involves a certain cost: all reusers must consider upgrading to the new version and eventually must actually upgrade. Evolution, also, may cause unexpected behaviour in reusers. A reuser that upgrades to a

new version of a component can experience different problems: the behaviour of the evolved component has changed, properties of the component that were valid before do not hold anymore, and so on. This kind of conflicts is referred to as *evolution conflicts*.

Furthermore, a component that is reused improperly may cause unexpected behaviour, both in the reuser and in the component itself. Or, even worse, two components that exhibit correct behaviour when reused separately may cause errors when reused both together in the same system. These kinds of conflicts are called *composition conflicts*.

Conflicts show up during evolution or composition because properties that were relied on by reusers have become invalid. At the programming level composition and evolution conflicts result in erroneous or unexpected behaviour. From a modelling perspective, composition and evolution conflicts may result in a model that is inconsistent (for example, referencing model elements that do not exist anymore), or in a model that does not have the meaning intended by the different reusers.

One of the virtues of the reuse contract formalism is that many evolution and composition conflicts can be detected in a semi-automatic way. When the same component is modified by means of two different reuse contracts, conflicts can be detected by comparing the two contract types and reuser clauses. For each conflict a formal rule can be set up to detect the conflict. As the conflicts that can possibly occur are dependent of the contract type, tables can be set up where both the rows and columns represent contract types and the fields specify what conflicts can possibly occur for a certain combination of types. This table can be filled in by simply comparing all contract types two by two, and determining whether they can interact in an undesired way. Using these tables it becomes possible to detect conflicts semi-automatically.

Qualitatively assessing the impact of changes in software

The reuse contract formalism can also help in *assessing the impact of changes in software*. Since an explicit link is maintained between the modified component and the original component, it becomes easier to trace on which other components the changes will have an effect.

Reuse contracts can also provide help for *effort estimation*, where the software developer needs to assess the cost of customising or redesigning a certain software component.

Tools

Thanks to the simplicity of the reuse contract formalism, it is fairly straightforward to provide tool support. Some tools have already been implemented for reuse contracts. The most important one is a *reuse contract extractor* (written in Smalltalk) that makes it possible to extract reuse contract information directly from the code, for single classes as well as for collaborating classes. Some preliminary work has also been done on implementing a graphical *reuse contract editor* (in Java), which allows us to write down a class collaboration, and express its evolution by means of reuse contracts. Finally, the construction of a *reuse contract repository* is also under development.

Conclusion

In order to improve the quality of an object-oriented architecture or design from an evolutionary point of view, reuse and evolution should be dealt with in a disciplined way. To achieve this, notion of change should be present in the definition of object-oriented architectures and designs. We have proposed the reuse contract formalism as a general mechanism to do this. It deals with component evolution in a natural way, by expressing a contract between the provider and reuser (or evolver) of a reusable component.

Thanks to reuse contracts we are able to detect anomalies in object-oriented designs and architectures in a semi-automatic way. More specifically, the reuse contract formalism allows us to detect conflicts that show up during evolution or composition of components. Moreover, reuse contracts can also help in assessing the impact of changes in software.

References

[Booch98] Grady Booch, “Master Class - Best of Booch: The Future of Software", June 1, 1998.

[Codenie&al97] Wim Codenie, Koen De Hondt, Steyaert Patrick and Vercammen Arlette: “From Custom Applictions to Domain-Specific Frameworks”, Communications of the ACM, Special issue on application frameworks, 40(10), pp. 70-77, October, 1997.

[Lucas97] Carine Lucas: "Documenting Reuse and Evolution with Reuse Contracts", PhD Dissertation, Vrije Universiteit Brussel, September 1997.

[Mens98] Tom Mens: “Nested Dependency Graphs as a Visual Formalism for Object-Oriented Reuse”, Poster, Proceedings of 20th International Conference on Software Engineering ICSE ’98, Volume II, pp. 129-134, IEEE Press, 1998.

[Mens&al97] Tom Mens and Wilfried Verachtert: “Practical Experiences with Reuse Contracts”, Proceedings of the European Reuse Workshop, Brussels, November 1997.

[Mens&al98] Tom Mens, Carine Lucas and Patrick Steyaert: “Giving Precise Semantics to Reuse in UML”, Proceedings of ICSE ’98 Workshop on Precise Semantics for Software Modeling Techniques, Kyoto, Japan. Technical Report TUM-I9803, pp. 73-89, Technische Universität München, April 1998.

[Mens&al98b] Tom Mens, Carine Lucas and Patrick Steyaert: “Supporting Reuse and Evolution of UML Models”, Proceedings of UML ’98 International Workshop, Mulhouse, France, June 1998.

[Parnas94] D. L. Parnas: “Software Aging”, Proceedings of 16th International Conference on Software Engineering ICSE ’94, IEEE Press, 1994.

[Rational97] Rational Software Corporation: "Unified Modeling Language 1.1 Document Set", http://www.rational.com, September 1997.

[Steyaert&al96] Patrick Steyaert, Carine Lucas, Kim Mens and Theo D'Hondt: "Reuse Contracts: Managing the Evolution of Reusable Assets", Proceedings of OOPSLA '96, ACM SIGPLAN Notices, 31(10), pp. 268-286, ACM Press, 1996.