Assessing the Evolvability of Software Architectures

Tom Mens, Kim Mens

Programming Technology Lab, Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
{ tommens, kimmens }@vub.ac.be

The quality of object-oriented architectures is measured by characteristics such as modularity, extensibility, flexibility, adaptability and reusability. It is recognised that software systems featuring these characteristics are much easier to evolve and maintain. 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 address the problems of architectural evolution *directly*.

Software systems have a natural tendency to evolve, due to changing requirements, adoption of new technology, software maintenance, performance issues, new insights in the domain, etc. To cope with this fact, object-oriented software systems, and more specifically their architecture, should be made as evolvable as possible. This is the only way to avoid them turning into legacy systems.

Unfortunately, software architectures are usually defined in a much too static manner, and do not adequately deal with change, let alone *unanticipated* changes. Because of this, there are many problems when the architecture does change: version proliferation, architectural drift and the ripple effect are only some of these problems. In order to solve them, evolution should be dealt with in a less restrictive (but still disciplined) way, for example by applying the *reuse contract* formalism [1, 2, 3] to the domain of software architectures.

The reuse contracts approach allows to detect anomalies in object-oriented architectures in a semi-automatic way. More specifically, the formalism enables detection of conflicts that show up during evolution or composition. Reuse contracts can also help in assessing the impact of changes in software.

The essential idea behind reuse contracts is that (architectural) components are modified on the basis of an explicit contract between the *provider* of a component and a *reuser* that modifies this component. The purpose of the 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 declare (in a *provider clause*) how the component can be modified. The reuser needs to declare (in a *reuser clause*) how the component is reused or how it 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. The *contract type* expresses *how* the provided component is modified. Contract types and the obligations, permissions and prohibitions they impose are fundamental to disciplined evolution, as they are the basis for detecting conflicts when provided components evolve. Note that reuse contracts are flexible enough to allow unanticipated evolution of components too. Evolved components can break assumptions made by a provider, as long as this is explicitly declared in the reuse contract.

To conclude, we briefly discuss how the reuse contracts approach helps in keeping the model of the provider consistent with the model of the reuser, and how it addresses several questions posed during the workshop.

Detecting anomalies in object-oriented design: Often conflicts show up during evolution or composition because properties of the provided component that were relied on by reusers have become invalid. These conflicts may result in a model that is inconsistent (for example, referencing elements that do not exist anymore), or in a model that does not have the meaning intended by the different reusers. The reuse contract formalism allows to detect many of these conflicts 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 of these conflicts, formal rules can be set up to detect them.

Assessing the impact of changes in software: Because reuse contracts maintain an explicit link between the evolved and the provided component, it becomes easier to trace on which other components changes will have an effect. Reuse contracts can also provide help with *effort estimation*, where the software developer needs to assess the cost of customising or redesigning a certain software component.

Tool support: Some tools have already been implemented for reuse contracts. The most important one is a reverse engineering tool in Smalltalk that uses reuse contracts to extract architectural information directly from the code [4]. A basic graphical reuse contract editor has also been implemented in Java. It allows to write down class collaborations, and express their evolution by means of reuse contracts. Finally, the construction of a reuse contract repository is currently under development.

References

[1] C. Lucas: "Documenting Reuse and Evolution with Reuse Contracts", PhD Dissertation, Vrije Universiteit Brussel, September 1997.
[2] T. Mens, C. Lucas and P. Steyaert: "Supporting Reuse and Evolution of UML Models", Proceedings of UML ’98 International Workshop, Mulhouse, France, June 1998.
[3] P. Steyaert, C. Lucas, K. Mens and T. 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.
[4] K. De Hondt: "A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems", PhD Dissertation, Vrije Universiteit Brussel, October 1998.