Architectural Extraction in Reverse Engineering by Prototyping: An Experiment*

Sander Tichelaar Stéphane Ducasse
Theo Dirk Meijler
Software Composition Group, Universität Bern
(ducasse,tichel)@iam.unibe.ch,tdmeijler@research.baan.nl
http://iamwww.unibe.ch/~(ducasse,tichel)/

1 Introduction

In this workshop proposal we present a prototype approach to help the extraction of architectural information in the re-engineering process.

Commonly, the re-engineering life-cycle has been defined as a succession of the following tasks: analysis of requirements, model capture (understanding the system), problem detection, problem analysis, reorganisation and change propagation [1]. We have evaluated the benefit of a prototyping approach with a focus on model capture. Although prototyping is a known approach to evaluate the application feasibility, costs, comparison and validation of choices, we focus in this paper on the aspects of prototyping that are helpful for re-engineering.

In the following sections we first present the problem, and afterwards we present our proposal to solve this problem: a pattern describing how to use prototyping to extract architectural information from a legacy system.

2 Problems with extraction of architecture

During re-engineering a complex system, understanding its architecture is of primary importance for the quality of the process itself [1]. Indeed, without understanding the architecture the re-engineering can miss some aspects of old systems that are of key importance.

Problems arise when there is no accurate documentation and when the original designers and programmers are difficult to access.

- Good documentation may be not present due to factors like time-to-market pressure and inconsistency between code and documentation due to versioning. Sometimes there is simply no documentation at all. Moreover, documentation often does not present the *why* and only focusses on the *how* of the design elements: although there is documentation it is not clear what problem is actually solved, and how the system can be adapted and extended.
- Access to the original designers and programmers gives the possibility to discuss about the architecture. But access can be difficult, because developers are working on other projects or even worse, left the company at all.

*This research is supported by the Swiss National Science Foundation, grant 2000-46947.96 and the FAMOOS European Esprit Project grant 21975.

3 A Pattern for Architectural Extraction

In this section we introduce a pattern for the extraction of architectural knowledge from old systems. The approach uses a prototype to model the extracted information.

3.1 The pattern

Name: Architectural extraction using a prototype

Problem: The knowledge of the application architecture is hard to obtain because existing documentation is missing or inconsistent, and/or the original developers are gone or difficult to access.

Symptoms:

- Inconsistent documentation: documentation may appear not to match the actual implementation. For instance, documented classes are not found in the code, or classes appear in the code and not in the documentation.
- Lack of useful documentation: often, documentation explains only *how* the system is designed and not *why*. To re-engineer a system you have to understand why decisions have been taken, otherwise the re-engineered system can have the same problems as the old one, or solutions to problems in the old system (i.e. knowledge) may be overlooked.
- Even if the original developers are accessible, knowledge may be hard to extract from them, because they are normally working on the implementation level.

Solution: Building a prototype that represents the architecture of the part of the system that you want to understand.

Recipe: The process is iterative and looping :

1. start to identify the aspects, i.e. the main focus points of the system, like for instance, communication and user interfaces.
2. make an initial design based on the available documentation and discussions with the developers, if possible.
3. start a prototype according to the initial design.
4. have a running prototype at all stages. This forces you to be *complete*: a system has to solve all the problems up to a certain point to be able to run. In this way you are automatically confronted with known and unknown problems the old system solves and the prototype should solve: knowledge is extracted.
5. compare the design with the old code and discuss with the developers about the design of the prototype. In this way you can find out design decisions, features of the system that are unknown (for instance, because the original developers left) or undocumented.
6. alter the prototype and the inconsistent documentation according to the discussions.
7. goto point 4....

Example: In the context of the FAMOOS[1] project we currently test this approach for a user interface system for the (remote) control of mail sorting machines for AEG [4]. This user interface system consists of 350.000 lines of C++-code and it should be re-engineered to provide for a more flexible and scalable communication layer. The system is highly complex due to the flexibility of user interfaces and the intrinsic communication structure. The communication structure now runs into performance problems while being scaled up to cover more communication nodes. It should therefore be extensively adapted. This change is hard to achieve, because of the complete lack of documentation and technical comments in the code.

The prototype clearly helped to extract the architecture of the AEG system due to discussions with the developers. Assumptions of the prototype developers proved partly false, but it triggered the developers of the old system to explain the architecture and its features.

Advantages: Besides the extraction of documentation the prototype approach offers the following advantages:

Extracting knowledge from engineers. The process of prototyping will be an indirect form of extracting the knowledge of the engineers of the existing software[2]. This is also related to the role rapid prototyping plays in certain "standard" (i.e., forward engineering based) software engineering processes [3] to find out what the users really want. It will give understanding of what certain parts of the existing software does.

Problem detection. Knowlegde about the existing software can be extracted by looking for solutions in the existing software for problems that occur in the prototype. Indeed, when a problem occurs in the prototype it has to have a solution in the existing software.

Separation of aspects. We can prototype different aspects of software separately. We may for example, separately prototype a solution to communication and one for implementing the user interface. Thus, the prototypes may be simpler to build, allow us to focus on the different aspects. However the necessity for integration in a final solution should be kept in mind.

Awareness. Using a running prototype demands awareness of all the aspects that you are prototyping, i.e. with a running prototype you know always if the structure you have in your prototype, has the same behaviour as the old system.

Limitations: In this pattern we focused on using prototyping in re-engineering. But due to unclearness of goals, within and without the re-engineering life-cycle, there may be some problems:

The prototype approach does not fit well in the re-engineering life cycle as presented in the introduction. The model capture and problem detection and analysis are mixed in an *ad hoc* process. This is done by comparing with the old code and trying to understand it, and the fact that problems are detected and that reorganisations by means of new design are done. (Therefore, the prototype can probably benefit from many metrics and other analysis methods as well).

It is also easy to mix re-engineering and forward engineering in the prototype approach: the use of a prototype always implies a tension between the extraction of

[1] FAMOOS that stands for Framework-based Approach for Mastering Object-Oriented Software Evolution is an European Esprit project.

[2] Out of our experience, we saw that a prototype can trigger developers to make their knowlegde explicit ("He, in our system we already solved that problem in *this* way").

the original architecture and the definition of a new one (cleaning up the old one or clearly changing it). When the prototype is used in the latter goal in mind it may not reproduce the old architecture. The work involved in building the prototype may be too much for the sake of extracting the architecture alone. It may be more attractive to build a prototype, if it is also used for testing possible evolution schemes of the architecture or new technology, which is the normal use of rapid prototyping.

Related:

- While building the prototype other analysis methods (e.g. metrics) may help.
- A prototype approach can have other goals like testing target architectures.
- Confrontation with developers in form on question-response discussion based on class diagrams can give real insights about the architecture of the system [2]. In comparison the prototype approach is more behavioral oriented in the sense that this is not the structure of the classes that is the starting point of the discussion but the behavior of the prototype in terms of its functionality and properties like flexibility.

References

[1] E. Casais. State of the art in OO re-engineering methods, Oct. 1996. FAMOOS Achievement Report A1.3.1.

[2] G. Florijn. Class diagrams as support for re-engineering. Personal Communication.

[3] I. Sommerville. *Software Engineering*. Addison-Wesley, fifth edition, 1996.

[4] S. Tichelaar. AEC bedienkonzept prototype: A first basic design experiment. Technical report, UniBern, 1997.