Techniques for understanding legacy software systems
Kuipers, T.

Citation for published version (APA):
Kuipers, T. (2002). Techniques for understanding legacy software systems

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 8

Conclusions

In the introduction to this thesis, three research questions have been posed. They were:

- How can we reduce the search space when searching for a particular artifact in a legacy software system?
- How can we obtain sufficiently detailed information about a legacy software system to perform a structural change (semi-)automatically?
- How can we alter the software engineering process such that we no longer produce legacy systems?

8.1 Reduce Search Space

In Chapter 2, it was shown how we can use a number of analysis techniques, lexical analysis being the prime one, to derive facts from a legacy system. A tool architecture for performing such analyses was described. Using these techniques, a call-dependency analysis, or a data-usage analysis can be made of a system. The results from such an analysis reduce the search space of someone looking for a particular artifact because, for instance, they are looking for a module that modifies a particular piece of data.

Chapter 3 builds on those results by integrating all analyses in a compact, browseable presentation format. This way, all analysis results are readily available, and are all linked to each other. An engineer can move easily from an overview of all modules in a system, to an overview of all modules that operate on a piece of data, to an overview of all modules that actually modify that piece of data, and so on. Building these documentation generators gets facilitated by so-called island grammars, an analysis technique which borrows from lexical analysis its flexibility, and from syntactic analysis its thoroughness.
8.2 Information for Structural Change

Since legacy systems are usually very large, the amount of facts extracted from them is usually large too. Somehow, this information needs to be filtered, preferably automatically. One way of automatically filtering is to try to derive an object-oriented design from an existing, non-object-oriented system. Once this design is derived, the actual structural change can be performed manually. Chapter 4 examines two methods that can be used for the automatic derivation of objects. These methods use exactly the facts that can be derived using the techniques developed in the earlier chapters. Although deriving an object-oriented design completely automatically from a legacy system turns out to be undesirable, if not impossible, the techniques examined in this chapter do support the human engineer in automatically relating data and procedures. Having this information available is a very good starting point for an actual object-oriented redesign.

Another way to interpret the derived facts is by automatically grouping the data in a system by the concept they are related to. That is, to group data of the same type. Chapter 5 uses a technique called type-inference to infer the type of a variable in a system. Using type inference, a variable can be determined to hold either a date, or a monetary value, or a social-security number, etcetera. Using the concept analysis from Chapter 5 on programs and the inferred type of the variables they use appears to give a better starting point for an object-oriented redesign than using just the variables.

When we are looking for detailed information, even syntactical analysis in itself will not, in general be enough. Usually, a number of syntactical analyses need to be performed to get the right fact out of the system. More often than not, these analyses have a conditional dependency, and they use each others results as input. Having a system that allows for the elegant and concise development of such combinations of analyses is the topic of Chapter 6. In it, a form of parse tree traversal is developed that can be used to perform exactly the kinds of interdependent syntactical analyses as described.

8.3 Altering the Software Engineering Process

New insights have improved the software engineering process over the last decades. These improvements are usually only applied to software development (the building of new software systems), not to software maintenance. In Chapter 7 the relation of the techniques developed earlier in this thesis to a software development process called "Extreme Programming" is examined. Furthermore, the feasibility of applying this process to the maintenance of legacy systems is discussed. Using this, or a similar methodology, and supported by the right analysis techniques, a team of engineers can maintain a legacy system in such a way that it stops being a legacy system, and becomes an understandable, modifiable, and maintainable system, once again.