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.
Summary

This thesis presents a number of experimental techniques for understanding legacy software systems. Software systems that are used need to be maintained. Maintaining a complex software system is a daunting task. The maintenance activity consists of modifying the source code of the system. A major part of that activity is finding the exact location of the artifact to be changed in the source code. The changes to be made in the source code can vary from performing minor changes, e.g., the fixing of a small error, to structural changes, where the design of the system is changed, and the code accordingly. Different types of information are needed to perform either change.

A way of gathering such information is through rapid system understanding, where lexical analysis is used to extract a number of facts from Cobol legacy systems. These facts include a system inventory, which presents some basic metrics of the system. Furthermore, detailed information of database usage per module is derived and information about the usage throughout the system of data fields. The relation between Cobol sections is analyzed and presented through both a graphical representation, and as sorted lists of which sections are performed most often. Finally, a number of conclusions are drawn with respect to these analysis results. Using a very lightweight, and (thus) not very precise analysis technique, we can give initial answers to questions such as: “Does the system contain reusable code?”, “What statements describe business rules?”, and “What fraction of the code is platform specific?”. These answers are by no means detailed enough to perform a structural modification of the system, but they give more than enough information to decide whether or not to perform a detailed investigation, and give this information in a rapid and cost-effective way.

The results of the analyses described above can be consolidated by building a documentation generator. This thesis presents the notion of a documentation generator as a system that performs any number of more or less detailed analyses on a legacy software system, and presents the results in an interlinked way, with different levels of abstraction. For some of the more detailed analyses that are performed as part of a documentation generator, lexical analysis as used during rapid system understanding is not powerful enough, but full syntactic analysis may
be too restrictive, or too expensive. This thesis introduces the notion of island grammars, from which an analysis can be generated that is liberal when analyzing the larger part of the source code (the “water”), but very strict when analyzing the relevant part (the “island”).

A documentation generator integrates manual documentation with the automatically generated documentation. It utilizes graphical, hyperlinked representations of dependencies between different types of modules, data files, databases, and so on.

When looking to alter a system structurally, structure must be imposed on a largely unstructured, or not-well-enough structured system. One way of imposing structure is to migrate a procedural system to an object-oriented system. This thesis compares two techniques for relating data and procedures from a procedural system, so to act as a starting point for an object-oriented (re-)design of that system. These techniques are cluster analysis and concept analysis. Cluster analysis works by calculating a distance between various objects, and group the objects that are less than a certain distance from each other in a single cluster. Here, the distance is calculated between different data fields in a system, where distance relates to whether or not they are used in the same modules. Concept analysis groups related items with their features into “concepts”. Concepts are maximal subsets of items sharing exactly the same features. Here, the items are data fields, and the properties they have are the modules they are used in.

Concept analysis appears to be better suited for object identification than cluster analysis. A case study shows that when deriving certain data fields from a Cobol software system, and ignoring others, the concept analysis of data fields and the modules they are used in results in a starting point for an object-oriented remodeling of the original software system.

Legacy software systems are largely untyped. This makes systems harder to maintain, in part because it is hard to find the “relevant” data in a system. Type inference can alleviate this problem in part. All the data fields in a system are grouped together in types, based on the way they interact with each other. Using these types as items in a concept analysis, instead of plain data fields, greatly improves the result of the concept analysis as a starting point for an object-oriented remodeling of the system.

In order to perform analysis on software systems such as those needed for type inference, the system must be parsed, and the parse trees must be analyzed. This thesis presents a system that lets a software engineer access parse trees in an object-oriented fashion. The system also presents the engineer with tree traversals, allowing him to focus on specific analyses while selecting one of the traversal strategies presented by the system. Analyses written using this system can be easily linked to all sorts of applications.

Extreme programming is a software engineering methodology that bundles a number of programming practices and tries to make their interdependency explicit.
Extreme programming focuses on changeability of systems. A system developed using the extreme programming methodology should be ready for any change its owner needs from the system, be it minor or structural. The last chapter of this thesis examines whether the techniques used in extreme programming can be retrofitted to legacy system maintenance in order to improve the understanding of, and ultimately the changeability of such a system.