Exploring software systems
Moonen, L.M.F.

Citation for published version (APA):
Moonen, L. M. F. (2002). Exploring software systems Amsterdam: University of Amsterdam, Faculty of Natural Sciences, Mathematics, and Computer Science

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.
The main contributions of this thesis include (i) an investigation of the analogy between software exploration and urban exploration which results in the concept of legibility of a software system, (ii) island grammars that can be used for robust and goal directed parsing of software artifacts, (iii) a type inferring technique to abstract from COBOL code, and (iv) the detection and use of code smells to assess and improve the quality of software.

In the introduction to this thesis, we posed a number of questions concerning the creation of tools that support exploration of software systems and the application of such tools to particular maintenance tasks. Below, we will reflect on these questions, describe how the various chapters contribute to answering each question and draw some conclusions.

11.1 Effective Extraction

Question 1: How can we effectively extract information from a software system’s artifacts that can be used in a software exploration tool?

One of the first steps in a software exploration tool is source model extraction: the automated extraction of information from software artifacts. In the first part of this thesis, we argue that this step is hindered by the typical irregularities that occur in these artifacts (such as, syntactic errors, incomplete source code, language dialects and embedded languages) which make it hard (or even impossible) to parse the code using common parser based approaches. In Chapter 2, we present a solution to these issues in the form of island grammars, a special kind of grammars that can be used to generate robust parsers that combine the accuracy of syntactical analysis with the speed, flexibility and tolerance usually only found in lexical analysis.
In addition, we describe **MANGROVE**, a generator for source model extractors based on island grammars that provides its user with generated traversals that ease the mapping from parse results to source models. The combination of island grammars with generated traversals blends two forms of attractive default behavior: (i) island grammars allow us to limit ourselves to that part of the grammar necessary to describe the problem at hand, and (ii) generated traversals allow us to treat only those cases for which we need specific behavior. Consequently, extractor specifications are small and easy to write, modify and combine. The resulting flexibility contributes to software exploration because it enables task specific improvements of a system's legibility.

### 11.2 Creating New Knowledge

**Question 2: How can we combine and abstract facts about a software system to create new knowledge?**

In the second part of this thesis, we focus on *inferred types* as an abstraction that groups the variables that occur in a software system. Types form a good starting point for software exploration but, unfortunately, not all software systems that require exploration were written in languages with an adequate type system. Furthermore, developers often use the built-in types of a language to represent variables of different “logical” types which renders them unusable as abstractions since they group variables that should be in different groups.

To resolve these issues, Chapter 4 proposes a method to infer types for the variables in a COBOL system. Our method groups variables in types by considering the way in which they are actually used in the system. We present the formal type system and inference rules for this approach, show their effect on various real life COBOL fragments, and describe the implementation of these ideas in a prototype tool.

A potential problem with this method is *type pollution*: the phenomenon that inferred types become too large and contain variables that intuitively should not belong to the same type. In Chapter 5, we analyse this problem and present an improved version of our type inference algorithm that uses subtyping. Furthermore, we provide empirical evidence that subtyping is an effective way for dealing with pollution.

Chapter 6 combines type inferencing with mathematical concept analysis to create a new level of abstractions that group the procedures in a legacy system together with the data types they operate on. These abstractions are very similar to abstract data types and can be used as starting points for software exploration and for an object oriented re-design of the system.
11.3 Supporting Maintenance

Question 3: How can we use the information obtained in the first two questions to support maintenance?

In this thesis we describe a number of case studies that investigate the use of software exploration techniques to support particular maintenance tasks. The issues that were studied focus on useful methods for presenting analysis results of the user and deal with the differences between the conceptual view in the programmer's mind and the technical view used by the machine.

Chapter 2 describes two small case studies that illustrate how island grammars can be used to compute the cyclomatic complexity of COBOL programs and to document component coupling in systems written in a 4th generation language.

In Chapter 3 we present a large study that shows how island grammars can be used for goal directed parsing, in this case lightweight impact analysis for estimating and planning software maintenance projects. We give a detailed description of the process of translating an impact analysis problem into an island grammar and discuss the advantages that this approach has over other techniques. We present a generative framework that allows a maintainer to create lightweight and problem-directed impact analyzers and demonstrate our technique using a real-world case study where island grammars are used to find account numbers in the software portfolio of a large bank.

Chapter 6 considers the gap between conceptual and technical views of a software system that may appear when we combine concept analysis with type inferencing to find abstractions in a legacy system. To address this issues, we present CONCEPTREFINERY, a tool that allows a software engineer to bridge this gap by manipulating an additional view on the calculated concepts while maintaining the relation with both the original concepts and the legacy source code.

Finally, in Chapter 7 we investigate how an invented abstraction as inferred types can be presented meaningfully to software engineers. We describe the construction of TYPEEXPLORER: a tool that supports exploration of COBOL software systems based on inferred types and illustrate its use on an industrial COBOL legacy system of 100,000 lines of code.

11.4 Software Quality Assurance

Question 4: How can we use software exploration tools to investigate and improve the quality of a software system?

In the last part of this thesis we explore the quality aspects of a software system from a refactoring and testing perspective. In Chapter 8, we present an approach for the automatic detection and visualization of code smells in JAVA code. These results were
used to support automatic code inspections where detected smells guide the inspection process. The graphical overviews immediately show the maintainers if the system contains bad smells, what parts are affected, and where the concentration of smells is the highest. Another promising application for smell detection is in refactoring tools. Currently, such tools only assist the developer with performing the actual transformation steps that are needed for a given refactoring. Combined with our smell detection, it would be possible to build more intelligent refactoring tools which actively suggest that a certain refactoring can be applied at a given point.

Chapter 9 argues that refactoring test code is different from refactoring production code. We present a set of bad smells that indicate trouble in test code and a collection of test specific refactorings to remove these smells. In Chapter 10, we explore the relation between testing and refactoring and investigate how they become intertwined when refactorings invalidate tests (e.g. by removing a method that is expected by a test). We describe the conditions under which such invalidation can occur and survey which of the refactorings from [Fow99] affect the test code. Finally, we present the notion of “test-first refactoring”: a method for improving the quality of software that uses smells in the test code as landmarks to explore where production code may be improved.