Exploring Software Systems

— Ph.D. Dissertation Synopsis —

Leon Moonen[1,2,*]

[1] *Delft University of Technology, Software Evolution Research Lab, Faculty Information Technology and Systems, Mekelweg 4, 2628 CD, Delft, The Netherlands*
[2] *CWI (Centrum voor Wiskunde en Informatica), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands*

`http://www.cwi.nl/~leon/` `Leon.Moonen@acm.org`

Abstract

Software evolution is required to keep a software system in sync with the ever-changing needs of the system's users and environment. An unfortunate side-effect of evolution is that it often causes the knowledge about a system to degrade, which in turn impedes further evolution.

In the dissertation, we investigate techniques and tools that help remedy this situation by supporting the exploration of a software system and improving its legibility [1]. We examine the analogy with urban exploration and present innovative techniques for the extraction, abstraction, and presentation of information needed for understanding software.

1. Introduction

Just like traditional exploring is about traveling to unknown places for discovery, *software exploration* is about investigating the unknown aspects of a software system to find out what is there. The objectives of these investigations can range from obtaining a birds eye view of the system (cf. reconnaissance flights) to a detailed examination of a system's "white spots" (cf. surveying previously uncharted territory).

One might wonder *why* software exploration is needed, and *how* these unknown areas appear in a software system. The answer to both these questions is *software evolution*: every software system that is used for an extended period of time needs to be modified and extended a number of times during that lifetime to keep the system operational. In fact, the majority of software engineers today are not involved with the production of new systems but are busy with changing and extending existing software systems [17]. Common reasons for these modifications include removal of program defects, improvement of the system's performance, adaptation to a new hardware or software environment and extensions or changes to the functionality of the system.

[*] The work described in the dissertation has been carried out at CWI.

Recurring changes and extensions to a system deteriorate its structure and pollute originally "clean" designs. Gradually, the relation between the system and its design documentation diminishes and the system becomes less and less maintainable. When less information is available, subsequent changes will have an even more damaging impact on structure and maintainability. It is a well-known fact that as a result of this evolution process, the complexity of a system increases and the knowledge about the system degrades, unless specific actions are undertaken to prevent this [11, 18].

Although software systems are designed to be flexible, in practice they often show a strong *resistance to change*, especially in the case of legacy software systems. In fact, Brodie and Stonebreaker define a legacy system as: "*any information system that resists change*" [14]. To overcome this resistance, software engineers need techniques that help them manage the increasing complexity that results from evolution.

Another complicating factor in software maintenance is the fact that these maintenance tasks are often performed by others than the original developers of the software (who might still remember how and why a piece of code was written). These newcomers to the system have been called *software immigrants* since they are faced with the difficult task of finding their way in an existing software system, an experience similar to that of people who arrive in a new country and need to learn a new language and understand a new culture [21].

Several studies report that the bulk of today's software budgets are being spent on software maintenance. Estimates range from approximately 70% [12] up to 90% [20] of the total software costs. Bohner and Arnold report that the two most expensive activities in software maintenance are *understanding* the software system that has to be maintained and determining the *impact* of proposed change requests [13].

It is our objective to lower these costs by improving the support for the *exploration of software systems*. We investigate various possibilities of providing software engineers with tools that assists them in surveying the uncharted terrain that results from software evolution and collect up-to-date information about what is going on in the system [1].

2. Exploring Software Systems

Whenever we visit a new city or building, we use exploratory techniques to learn about the space and get to the places we want to visit. The cognitive process that is applied during such visits can be thought of as continually trying to answer the following three questions:

1. *Orientation*: Where am I?
2. *Discovery*: What else is out there?
3. *Navigation*: How do I get there?

This *wayfinding* process is studied intensively in architecture and city planning. The goal is to collect principles and guidelines for the design of cities and buildings that allow their users to better orient themselves and improve how they navigate through the space.

In his book "The Image of the City", city planner Kevin Lynch uses the concept of *legibility of a city* to develop a theory of city planning and urban design where he defines legibility as "*the ease with which its parts may be recognized and can be organized into a coherent pattern*" [19]. Lynch studied how people organize information about their environment by asking them to draw simple maps of their hometowns. Based on these surveys, he identified five principal elements that are used to build a mental model of a city:

Landmarks: The outstanding (static) features in a city. Examples include prominent buildings, monuments, and shop-fronts. Landmarks are used as reference points by the observer: they give a sense of location and bearing.

Paths: Streets or footpaths that allow the observer to travel through the city.

Nodes: The important points of interest along paths, for example, street intersections, bridges or town squares.

Districts: The areas in a city that have a common property allowing them to be viewed as a single entity. Examples of districts are shopping areas, residential areas, but also the historic center or the business district.

Edges: The boundaries to areas. They form a physical barrier to travelers. Examples include rivers or major roads for pedestrians.

These structural elements can be used to divide a complex environment into smaller, connected and more manageable pieces that can be used directly to create a mental map detailing knowledge about that environment.

2.1. Application to Software

We argue that the cognitive process of exploring software systems is very similar to that of urban exploration and that corresponding principles and techniques can be used to support both processes. We define *legibility of software* using the same terms as Lynch used for legibility of the city: "*the ease with which its parts may be recognized and can be organized into a coherent pattern*". Improving the legibility of software is an important aspect of supporting the exploration of software systems because legible systems are more memorable and generate stronger mental models, which makes them easier to explore, and therefore easier to maintain.

However, in urban environments legibility is defined in the context of solving the spatial exploration problem that has a rather static nature. The set of structural elements for a given space are largely fixed (although there will be some variation between people based on cultural backgrounds and mobility). In contrast, the legibility of a software system is much more dependent on the particular problem that an engineer has to solve [15]. For example, the elements of interest that are used to explore the impact of a Euro conversion on a software system will differ significantly from the elements for exploring quality aspects of that same system. Consequently, we focus on flexible techniques that allow us to improve the legibility of software in respect to a given task instead of aiming at overall legibility improvement.

Some examples of software legibility elements are:

Landmarks: Particular variable types such as dates, account numbers, or currencies. Code characteristics such as code smells or design patterns that have been applied.

Nodes: "Structural" entities in software such as programs, modules, functions, types, classes, methods, variables.

Paths: Relations between these nodes such as call relations, inheritance, variables of the same type, etc.

Districts: Separation of business logic that describes how a system contributes to an organization's bottom line from technical aspects such as database access, communication with the environment, user interfacing, etc.

The modules in a software architecture, for example, the Linux kernel can be thought of as separate districts for scheduling, memory management, file system access, networking, and interprocess communication.

Edges: Boundaries between libraries (both system libraries and third party libraries) and the application code written by the developers, boundaries between parts that were produced by different teams that have code ownership, or the boundaries between client and server code.

3. Research Questions

The research described in the dissertation concerns the creation of tools that support the exploration of software systems. The work is structured around four central questions:

3.1. 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 challenges in a software exploration tool is parsing the artifacts during *source model extraction*: the automated extraction of information from software artifacts.

We argue that, in reverse engineering domain, these artifacts typically contain *irregularities* that make it hard (or even impossible) to parse the code using common parser based approaches. Examples of such irregularities are syntax errors, programming language dialects, embedded languages, incomplete source code, etc. Furthermore, since the information needed to improve legibility is task dependent, one can not a priori determine what source model should be extracted. Consequently, we need techniques for robust parsing of artifacts that allow flexible specification of the extracted models.

To resolve these issues, we propose *island grammars*, a special kind of grammars that can be used to generate *robust parsers* that combine the detail and accuracy of syntactical analysis with the speed, flexibility and tolerance usually only found in lexical analysis [2, 3].

We have created 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 software system's legibility.

3.2. Creating New Knowledge

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

The second challenge is to find (new) abstraction levels that are not explicitly available in the code and help software engineers gain knowledge about the system. There are two ways in which abstractions can contribute to the knowledge about a system: (i) they identify new landmarks that act as beacons for comprehension, and (ii) they disclose new routes for navigation through the system. Example abstractions one can think of are: *architectural views* that show the modules in a system and how they depend on each other, *data flow* that shows how data propagates through the statements in a program and between the programs in a system (for example via program calls, but also via databases), and *types* that group the variables in a system to make them more manageable.

The types of variables are an interesting starting point for software exploration (just think of Y2K remediations or Euro currency conversions) [7]. Unfortunately, not all software systems that require exploration were written in a language with an adequate type system. Furthermore, developers often use the built-in types of a language to represent different "logical" types, rendering them unusable as abstractions since they group variables that should be in different groups.

To resolve these issues, we propose a method to infer "substitute" types for the variables in such systems that can be used like ordinary types in the exploration process [4]. 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. We analyze this problem and present an improved version of our type inference algorithm that uses subtyping [5]. In addition, we provide empirical evidence that subtyping is an effective way for dealing with pollution.

Furthermore, we combine 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 [6]. These abstractions are very similar to abstract data types and can be used as starting points to explore an object oriented re-design of the system.

3.3. Supporting Maintenance

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

Several issues have to be addressed before the information obtained in the first two questions can be used to support maintenance tasks: What are useful methods for presenting the results of our analysis to the user? How to deal with the differences between the conceptual view in the programmer's mind and the technical view used by the machine (e.g. in a compiler, but also in a reverse engineering tool like ours)? To address these issues, the thesis presents a number of case studies that were performed to investigate how software exploration techniques can be used to support particular tasks.

We start with two smaller case studies to 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 [2].

Next, we show how island grammars can be used for *goal directed parsing*, in this case lightweight impact analysis for estimating and planning software maintenance projects [3]. 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.

Subsequently, we consider the gap between conceptual and technical views on a software system that may appear when combining concept analysis with type inferencing to find abstractions in a system. To remedy this situation, 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 [6].

Finally, we investigate how an invented abstraction as inferred types can be presented meaningfully to software engineers [7]. 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.

3.4. Software Quality Assurance

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

The fourth question addresses the use of software exploration tools for the purpose of software quality assurance. In particular, we look at the *quality aspects* of a software system from a refactoring and testing perspective.

Software exploration tools may be used to find places in the code that can be improved using refactoring. "*Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure*" [16]. The places that could benefit from refactoring are identified using so-called *code smells*. Code smells are a metaphor for patterns in code that are generally associated with bad program design and bad programming practices. As such, code smells are landmarks that can be used to assess and explore the quality of a software system: when a system possesses a lot of smells, its quality is questionable and the smells guide the way to the places that need to be improved. Some examples of code smells are: duplicated code, methods that are too long, classes that perform too much tasks, classes that violate data hiding or encapsulation rules or classes that delegate the majority of their functionality to other classes.

We present an approach for the automatic detection and visualization of code smells in JAVA code [8]. 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.

Unit tests are used to verify that refactorings do not change external behavior of the system [16]. Besides acting as safe-guard, unit tests help the maintainer understand the functionality and usage of the code that is tested. Consequently, it is not surprising that unit tests are also being refactored to improve their structure.

We discuss how refactoring test code is different from refactoring production code and present a set of bad smells that indicate trouble in test code and a collection of test specific refactorings to remove these smells [9]. Furthermore, we explore the relation between testing and refactoring and investigate how they can become intertwined when refactorings invalidate tests (e.g. by removing a method that is expected by a test) [10]. We describe the conditions under which such invalidation can occur and survey which of the refactorings from [16] 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.

4. Contributions

The main contributions of the dissertation 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 inferencing technique to abstract from COBOL code, and (iv) the detection and use of code smells to assess and improve the quality of software.

The various case studies show how these software exploration techniques and tools can be applied to solve real-life problems on industrial-size software systems.

5. Author's Publications

The following publications have appeared in the context of this dissertation:

[1] L. Moonen. *Exploring Software Systems*. PhD thesis, Faculty of Natural Sciences, Mathematics, and Computer Science, University of Amsterdam, December 2002. An electronic version is available for download from: `http://www.cwi.nl/leon/papers/phdthesis/`

[2] L. Moonen. Generating Robust Parsers using Island Grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*. IEEE Computer Society, Oct. 2001.

[3] L. Moonen. Lightweight Impact Analysis Using Island Grammars. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*. IEEE Computer Society, June 2002.

[4] A. van Deursen and L. Moonen. Type Inference for COBOL Systems. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE 1998)*, pages 220-230. IEEE Computer Society, Oct. 1998.

[5] A. van Deursen and L. Moonen. An Empirical Study into COBOL Type Inferencing. *Science of Computer Programming*, 40(2–3):189–211, July 2001.

[6] T. Kuipers and L. Moonen. Types and Concept Analysis for Legacy Systems. In *Proceedings of the International Workshop on Programming Comprehension (IWPC 2000)*. IEEE Computer Society, June 2000.

[7] A. van Deursen and L. Moonen. Exploring Legacy Systems Using Types. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE 2000)*, pages 32-41. IEEE Computer Society, Oct. 2000.

[8] E. van Emden and L. Moonen. Java Quality Assurance by Detecting Code Smells. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE 2002)*. IEEE Computer Society, Oct. 2002.

[9] A. van Deursen, L. Moonen, A. van den Bergh and G. Kok. Refactoring Test Code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP 2001)*, May 2001.

Also appears as a chapter in the book *eXtreme Programming Perspectives*, edited by M. Marchesi, G. Succi, D. Wells, and L. Williams. Addison-Wesley, Aug. 2002.

[10] A. van Deursen and L. Moonen. The Video Store Revisited — Thoughts on Refactoring and Testing. In *Proceedings of the 3nd International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2002)*, May 2002.

References

[11] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[12] K. H. Bennett. An introduction to software maintenance. *Information and Software Technology*, 12(4):257–264, 1990.

[13] S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society, 1996.

[14] M. L. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, interfaces and the incremental approach*. Morgan Kaufman Publishers, 1995.

[15] R. Brooks. Towards a theory of the comprehension of computer programs. *Int. Journal of Man-Machine Studies*, 18:543–554, 1983.

[16] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[17] C. Jones. *Estimating Software Costs*. McGraw-Hill, 1998.

[18] M. M. Lehman. On understanding laws, evolution and conservation in the large program life cycle. *Journal of Systems and Software*, 1(3):213–221, 1980.

[19] K. Lynch. *The Image of The City*. MIT Press, 1960.

[20] T. M. Pigoski. *Practical Software Maintenance – Best Practices for Managing Your Software Investment*. Wiley, 1997.

[21] S. E. Sim and R. C. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *Proc. of 20th Int. Conference on Software Engineering (ICSE-20)*, pages 361–370. ACM, 1998.