**Chapter 1**

**Introduction**

explorer /ɪksplɔrər/. An explorer is someone who travels to places about which very little is known, in order to discover what is there.

*Collins COBUILD English Language Dictionary*

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).

In this chapter, we motivate why software exploration is needed by describing how software evolution causes degradation of (knowledge about) a system after it is built. We investigate the analogy between software exploration and urban exploration which results in the concept of legibility of a software system and a collection of principal elements responsible for this legibility. Next we describe how these ideas are related to previous work in the areas of program comprehension and reverse engineering. We conclude by posing a number of research questions that are investigated in this thesis.

### 1.1 Software Evolution

One might wonder why software exploration is needed, and how these unknown areas appear in a software system. After all, a software system only exists because it was designed and created by people who clearly must know what is there, or the system could never have been built in the first place. In the remainder of this section, we will investigate the forces that operate on a software system and cause the appearance of white spots.
A typical software system is modified and extended a number of times during its lifetime to keep it 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 [Jon98]. This process of keeping a software system in sync with the ever-changing needs after it was put in production is called software evolution or software maintenance.

In the 1970s, Belady and Lehman studied the evolution of several large software systems (such as IBM’s OS/360). Based on these studies, they formulated their Laws of Program Evolution Dynamics that model the dynamic behavior of a software system during the evolution of that system [BL76, Leh80a, Leh80b]. Recent case studies report on further evidence for the validity of these laws [Leh97, LPR98]. The first two of their laws consider the software system itself and are the most relevant for our work since they describe the inevitability that parts of a system become less known and need exploring:

1. **Continuing Change**: any software system that is actually used will undergo continuous modification or it becomes useless.

   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.¹

2. **Increasing Complexity**: as a result of these modifications, the complexity of a system will increase unless specific actions are undertaken to prevent this.

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.

This kind of resistance to change is not unique to software systems. For example, it was also observed in architecture [Bra95]. In his book “How Buildings Learn: What Happens After They’re Built”, Brand states: “Almost no buildings adapt well. They’re designed not to adapt; also budgeted and financed not to, constructed not to, administered not to, maintained not to, regulated and taxed not to, even remodeled not to. But all buildings (except monuments) adapt anyway, however poorly, because the usages in and around them are changing constantly” [Bra95]. Although software systems are designed to be flexible, in practice they often turn out to resist change just as strong as buildings do, especially in the case of legacy software systems. In fact, Brodie and Stonebreaker define a legacy system as: “any information system that resists change” [BS95]. To overcome this resistance, software engineers need techniques that help them manage the increasing complexity that results from evolution. An example of such a technique is a software exploration tool that assists engineers in collecting up-to-date information about what is going on in the system.

---

¹ Extensions that result from changing user requirements have later been distinguished as a separate law of Continuous Growth which states that the functional content of a program must be continually increased over its lifetime to maintain user satisfaction [Leh97].
1.1.1 Software Immigration

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 actually still remember how and why a particular piece of code was written). Such 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 [SH98].

We can identify two main sources for software immigration: the first turnover happens after development when a system is transitioned to a different (part of the) organization that does the maintenance. Arguments for such a transition are that software maintenance requires specific skills that not necessarily correlate with the skills of good software developers. Moreover, "fresh" maintainers are more apt to make significant changes since they will be less attached to the program than its original developers [Pig97]. A second type of turnover happens whenever new employees are added to an existing software project (either in the development or in the maintenance stage) to make up for staff turnover, replace personnel, or to disengage senior team members. In both cases the new maintainers are confronted with an existing software system that they need to familiarize themselves with and investigate all its unknown aspects to find out what is there.

The result of all these complications is that software maintenance is an expensive part of the software life-cycle. Several studies report that the bulk of today's software budgets are being spent on software maintenance. Estimates range from approximately 70% [Ben90] up to 90% [Pig97] of the total software costs. Consequently, research that improves the maintenance process can make a tremendous contribution to decreasing the total costs of software.

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 [BA96]. It is our objective to lower the cost of these activities by improving the support for exploring software systems by software engineers.

1.2 Exploring Software Systems

In this thesis, we investigate various possibilities of providing software engineers with tools that help them explore the software system at hand and survey the uncharted terrain that results from software evolution and immigration. We introduce the issues surrounding software exploration by drawing the analogy with traditional exploration.

Historically, exploration is associated with people that go on a voyage of discovery and examine uncharted territory. This rarely happens these days since most of earth's geographical areas have been visited by man. However, there is a related type

---

2 One could argue that geographic exploration is replaced by space exploration.
of exploration that is very common in our everyday life and has been researched extensively: whenever we visit a new city or building, we use exploratory techniques to orient ourselves and get to the places we want to visit.

Below, we will first look at the traditional exploring metaphor and then investigate how urban planning techniques can help software exploration.

1.2.1 A Voyage of Discovery

When explorers went on a voyage of discovery, they traveled to areas for which they did not have a map or at best only had a rough map that was based on hearsay information or focused on different aspects of the region. A similar situation occurs in software engineering when the engineer has to deal with a software system for which no up-to-date or relevant documentation is available.

We can ask ourselves how this problem was solved by traditional explorers? When examining a given terrain, the explorer typically starts at a known point and investigates possible routes that leave from that point. These routes can be existing trails or courses determined by taking the bearings of features that are visible from the current position. Generally, the selection of routes of interest is based on the goal of the expedition, for example the mountain to climb or the desert to cross, and the terrain survey is a by-product of that expedition.

If we translate this approach to the software domain, we get the following description of the exploration process: a software explorer starts at a known point and investigates possible routes that leave from that point. However, because software is not tangible, it is much harder to identify what suitable starting points are, what routes the explorer can follow, and which features can be used to set out a new course. These concepts need to be made manifest in a software system before it can be examined by a software explorer using the approach described above.

To illustrate the general idea of software exploration, we will give a few examples of routes and features here: When we start exploring a software system at a given program, features of interest might be all other programs that are affected by this program. In that case, potential routes to explore are the calls from this program to other programs. A different set of interesting routes originate from data flow relations that result from database entries that are written by one and read by the other program. Another potential starting point could be a certain variable type (e.g. date or currency) with routes that lead the explorer to all program locations in which that type is used.

1.2.2 Urban Exploration

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 process that people apply 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?
Section 1.2 Exploring Software Systems

3. Navigation: How do I get there?

This process of spatial exploration is referred to as wayfinding.

Wayfinding and spatial cognition are studied intensively in architecture and city planning. The goal is to collect principles and guidelines that can be applied in the design of cities and public buildings to allow its users to better orient themselves and improve how they navigate through the space.

Legibility of the city

The foundations for wayfinding research were laid out by city planner Kevin Lynch in his book “The Image of the City” [Lyn60]. In this book, he 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”.

Lynch studied how people organize spatial 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 and 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 spatial knowledge about that environment. People generally start their orientation in a new environment using landmarks and gradually extend their knowledge using the other elements until a mental map is constructed.

Lynch discovered that in cities in which these elements are not manifest, people have much more trouble with creating mental overviews of their surroundings and relating their position to the total system. Using that knowledge, he proposes a design methodology that helps to design or improve cities so people can easier find their way. Basically this is done by ensuring that all these elements are used and can be easily recognized.
Wayfinding in Architecture

A followup study was done by architect Romedi Passini, who investigated how people navigate in large public buildings and malls [Pas84]. He describes wayfinding as an iterative three-staged process consisting of mental mapping to create cognitive maps of the environment, decision making to formulate action plans and decision execution to execute those plans. Passini identified many environmental factors that influence the process, such as building symmetry, user expectations, behavior of other people in the building, and old memories of being in that environment. Based on these results, he proposes a number of design guidelines that help to improve legibility of buildings.

In his guidelines, Passini uses the two structural elements of Lynch that he considers to be the most important: paths and landmarks, and introduces the notion of enclosures (or containers) to replace nodes and districts. These containers are used as a more general "node" element that itself can consist of a collection of organized elements. Since the user's ability to understand and orient in the environment is affected by the (apparent) logic of how the elements are arranged, Passini argues that it should follow a known organizational scheme. For example, the streets and canals in the center of Amsterdam are organized in a circular pattern, whereas the streets in Manhattan are organized as a grid. When we know the organizational scheme, it becomes easier to navigate, determine our location and memorize a route.

Together with graphic designer Paul Arthur, Passini studied how wayfinding in existing buildings can be improved by adding signage and if such signage can be incorporated in the architecture of a building (so called "environmental communication") [AP92]. An example of such incorporation is using the burbling sound of a fountain to help people find and recognize the lobby of a building as public space. Their conclusion is that the addition of signs can be an efficient way to improve wayfinding in an existing space but since there are other factors that limit people's wayfinding capabilities (discussed above), the addition of signs alone does not suffice.

Application to Software

We can 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 [Bro83]. 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, our focus will
be 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, and currencies. Code characteristics such as code smells and points at which a certain refactoring can be applied.

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

**Paths:** Relations that can exist between nodes such as call relations, inheritance, links between variables of the same type, etc.

**Districts:** Separation of the so-called business logic or business rules that describe how the system contributes to an organization's bottom line from the technical aspects such as database access, communication with the environment, user interfacing, etc.

The modules in a software architecture, for example, the Linux operating system kernel can be thought of as consisting of separate districts for process scheduling, memory management, file system access, network interfacing, and interprocess communication [BHB99].

**Edges:** The boundary 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.

Before we can investigate how these ideas can be applied in concrete software exploration tools, we need to take a more detailed look at the cognitive and technical issues of program comprehension.

### 1.3 Program Comprehension

The overall goal of software exploration is to gain a better understanding of a software system. It is a widely accepted fact that software engineers spend a large amount of their time on understanding the system that they are working on. Corbi reports that at least 50% of a software engineers' time is spent on trying to figure out what is actually going on in the system [Cor89]. Effective understanding is needed before one can find and fix defects, add new functionality, improve the implementation, etc. Because program comprehension (or program understanding) is such an important aspect of the software engineering process, numerous studies have been performed to come to a theory of program comprehension and identify techniques that can assist engineers with this task.

Comprehension is characterized as the construction of mental models that represent the objects in a text and the relationships between them [DK83]. In program
comprehension, these mental models represent the examined software system at various levels of abstraction. They can range from models of the code itself (e.g. the main components of the system and their relation to each other) to models of the underlying application domain (e.g. tasks performed by the system). Software engineers need these models during the maintenance, evolution, and re-engineering of the system.

Comprehension is an incremental process: software engineers gradually build up their knowledge by studying various aspects of the system, possibly at different times, and possibly by revisiting previously examined parts. They make use of comprehension strategies which help them manage information and reach a particular goal.

A number of people have studied the cognitive processes that are involved with program comprehension and the strategies that are used to build the mental models. Detailed surveys of these processes are presented by von Mayrhauser and Vans [MV95] and Storey [Sto98]. Here, we will give a short overview of the three main approaches that can be distinguished:

1. **Top-down**: this approach tries to reconstruct the mappings from the problem domain into the programming domain that were made when programming the system. This reconstruction is an expectation driven process: understanding starts with some pre-existing hypotheses about the functionality of the system and the engineer investigates whether they hold, should be rejected or refined in a hierarchical way (Brooks [Bro83], Soloway and Ehrlich [SE84]).

2. **Bottom-up**: this approach starts understanding from the source code, constructing higher level abstractions using chunking and concept assignment (Shneiderman and Mayer [SM79], Pennington [Pen87]). Chunking creates new higher level abstractions from lower level structures. When higher level structures are recognized, they replace the more detailed lower level ones. This helps to overcome the limitations of the human memory when confronted with too many pieces of information [Mil56]. The term concept assignment was introduced by Biggerstaff et al. for the process of describing the intent of certain parts of the system using terms at a higher level of abstraction than the source code [BMW93].

3. **Opportunistic combinations of top-down and bottom-up**: according to this theory, programmers frequently change between top-down and bottom-up approaches (Letovsky [Let86]), or even combine them at the same time (von Mayrhauser and Vans [MV95, MV96, MV97]), to create mental representations of a software system.

All these approaches have in common that they are based on the recognition of certain features in the code\(^3\) that are used for both abstraction and orientation. Brooks introduces the notion of *beacons* which are (sets of) easily recognizable features that appear in the code and are used for the generation and validation of hypotheses [Bro83].

---

\(^3\) We use the word "features" in its English meaning which refers to the traits, characteristics, elements, aspects, and properties of a system in contrast to software jargon where the meaning of features is limited to the functional aspects of a software system.
Soloway and Ehrlich describe the use of programming plans to capture the intent of the code: plans are *patterns* of features that indicate that a piece of code has a specific task. Biggerstaff *et al.* introduce the notion of *signatures* to describe sets of features that together signal the occurrence of a specific concept [BMW93].

These notions described above are all examples of software elements that can be used as landmark elements for the application of the wayfinding theories of Lynch and Passini. One of the goals of our work is to enable automatic detection and manifestation of such landmarks in a software system. Such automatic detection can be performed using reverse engineering.

### 1.4 Reverse Engineering

Reverse engineering techniques are often used to support program comprehension. Reverse engineering is defined as *the process of analyzing a subject system to identify the system's components and their interrelationships and, create representations of the system in another form or at a higher level of abstraction* [CC90]. The goal of reverse engineering is identification and recovery of the design artifacts of a system, such as its requirements, specifications, and architecture. In most cases, the process starts with analyzing the system's source code. From there, several higher-level abstractions can be derived such as its major building blocks (components), their relations and dependencies, architectural views of the system structure, etc. This information can be used to support comprehension of the system since such higher-level views help the maintainer to manage the complexity of the lower (source) levels.

A typical application of this technique deals with the *(automatic) redocumentation* of software systems. There, reverse engineering techniques are used to generate (technical) documentation from the sources of a software system to support maintenance activities [DK99a]. The obvious advantage of automatic redocumentation is that the documentation can be regenerated whenever the source is changed so it will never be out-of-date. Furthermore, the quality of the functional part of the documentation (which cannot be generated) will generally improve since maintainers don't have to spend time on the (boring) technical part of the documentation.

Many reverse engineering tools make use of compiler technologies such as lexical, syntactic, and semantic (static) analysis [BKV97]. Static analysis is a technique for computing approximate information about the dynamic behavior of computer programs. Static analysis of computer programs can for example be used to infer types, identify unreachable code, detect variable aliasing and find uninitialized variables. An overview of the major approaches to static program analysis is given by Nielsen *et al.* in [NNH99].

Figure 1.1 presents a general architecture that can be found in the majority of reverse engineering tools. It consists of three phases:

1. **Extraction:** Each reverse engineering effort starts with extracting facts (also referred to as *source models*) from a software system's artifacts such as its source code, build scripts and configuration files. It is also possible to collect informa-
Introduction

Chapter 1

Figure 1.1: General architecture of a reverse engineering toolset.

Figure 1.1: General architecture of a reverse engineering toolset.

1.5 Research Questions

The work described in this thesis concerns the creation of tools that support exploration of software systems using reverse engineering techniques and the application of such tools to perform particular maintenance tasks. The research is structured around four central questions discussed below.

1.5.1 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 that a software exploration tool has to cope with is parsing the artifacts during the extraction phase. 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, missing parts, etc. Furthermore, since the information needed to improve
legibility is task dependent, one can not a priori determine what type of source model should be extracted. Consequently, we should investigate techniques for robust parsing of artifacts that allow flexible specification of the extracted models.

1.5.2 Creating New Knowledge

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

The 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: (1) they identify new landmarks that act as beacons for comprehension, and (2) 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. Since a lot of legacy systems are written in a language without types, an interesting issue is whether we can infer "substitute" types for the variables in those systems, and if they can be used like ordinary types in the exploration process.

1.5.3 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)? In order to address these issues, we need to perform a number of case studies that investigate how software exploration techniques can be used to support particular tasks.

1.5.4 Software Quality Assurance

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

Our final question addresses the use of software exploration tooling for the purpose of software quality assurance. In particular, software exploration 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" [Fow99]. 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, it's 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.

1.6 Organization of this Thesis

The subsequent chapters of this thesis were originally written as a separate articles that investigate various issues in software exploration. As a result of this, there is a small amount of overlap between some chapters in the form of reiteration of definitions and examples. We have deliberately chosen to leave this overlap in place to make the work more accessible and to ensure that the chapters can still be read as self-contained papers.

This thesis consists of three parts: In the first part, we consider automated extraction of source models from software artifacts and the use of those models in impact analysis. One of the major challenges of source model extraction is dealing with irregularities in the artifacts that are typical for the reverse engineering domain (e.g. syntactic errors, incomplete source code, language dialects and embedded languages). Chapter 2 presents a solution in the form of island grammars that are used to generate robust parsers which combine the detail and accuracy of syntactical analysis with the flexibility and development speed of lexical approaches. In Chapter 3, we motivate that lightweight impact analysis is needed for the planning and estimation of software maintenance projects and present a technique for the generation of lightweight impact analyzers from island grammars. We demonstrate this technique using a real-world case study that concerns the impact of mass transformation in the software portfolio of a large bank.

The second part of the thesis considers inferred types as an abstraction that groups the variables that occur in a software system. Types are a natural abstraction in programming languages and form a good starting point for software exploration and re-engineering tasks. Unfortunately, the software systems that require re-engineering most desperately are often written in languages without an adequate type system (such as Cobol). Additionally, in languages that do have types (such as C), developers often only use the same built-in type (e.g. char, int or float) to represent different “logical” types (e.g. amount and age). As a result, types cannot be used as abstractions since they group variables that should be different. To solve these issues, Chapter 4 presents a method of automated type inference that considers the way in which types are actually used in a software 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 type inference tool for COBOL.

We continue our study in Chapter 5 with the analysis of type pollution, the phenomenon that inferred types become too large and contain variables that intuitively should not belong to the same type. We present an improved type inference mechanism that uses subtyping and provide empirical evidence that this is an effective way for dealing with pollution. In Chapter 6, we combine type inference and mathematical concept analysis to logically group the procedures in a legacy system together with the data types they operate on. The results are abstractions that are very similar to abstract data types. These abstractions can be used for exploration and are the starting point for an object-oriented re-design of the system. Finally, Chapter 7 investigates 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 how it can be used by examining an industrial COBOL legacy system of 100,000 lines of code.

In the third and 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 a method for the automatic detection and visualization of code smells in Java code. These results can be applied in two ways: (1) to support automatic code inspections where smells are used to guide the inspection process; and (2) the creation of intelligent refactoring tools that not only perform the transformation (as currently is state-of-the-art) but also suggest that a 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.

### 1.7 Origins of the Chapters

The chapters in this thesis have appeared as a paper in a journal or in the proceedings of an international conference. Only minor changes have been made to each published paper. The remainder of this section gives an overview of the earlier publications.


Acknowledgments

The work described in this thesis was sponsored in part by bank ABN AMRO, Pink Roccade Finance, and the Dutch Department of Commerce (Ministerie van Economische Zaken) via SENTER Project #ITU95017 SOS Resolver and in part by the Telematica Institute via the Domain-Specific Languages project.

The author would like to thank ABN AMRO, Pink Roccade Finance, the Software Improvement Group, and Epicont for providing the real-life and industrial-size software systems that have been used as software “guinea pigs” for the development and validation of the techniques and tools described in this thesis.