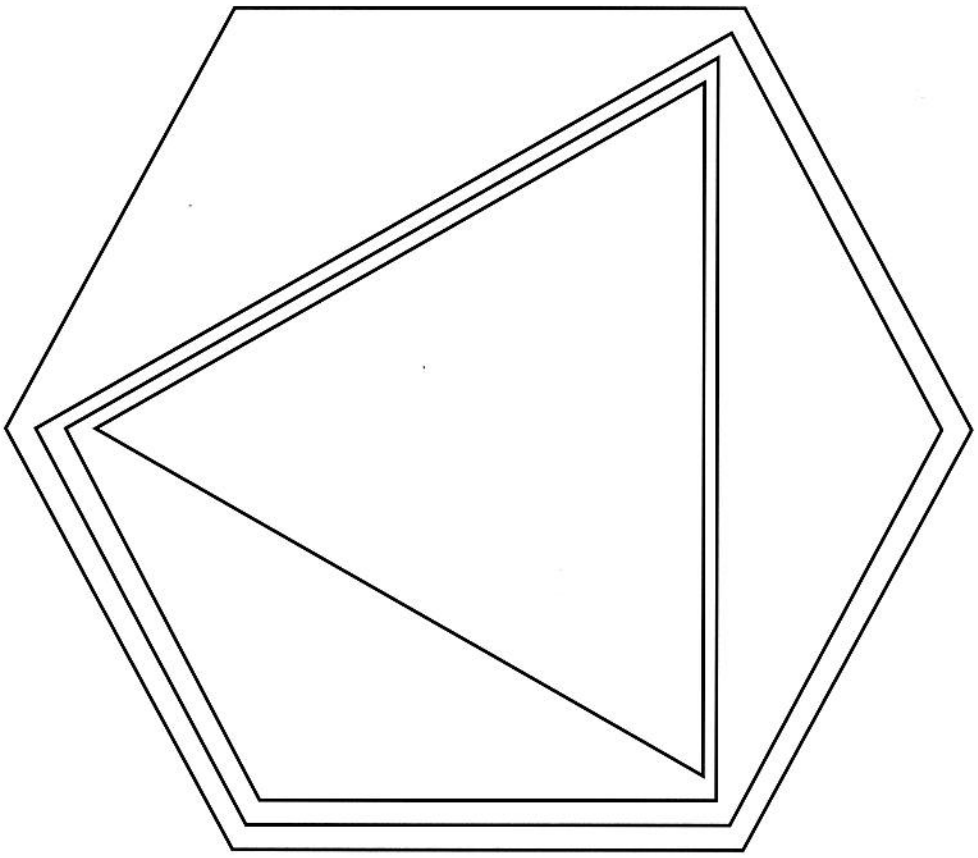
Exploring Software Systems



Leon Moonen

Exploring Software Systems

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof. mr. P.F. van der Heijden
ten overstaan van een door het
college voor promoties ingestelde commissie,
in het openbaar te verdedigen
in de Aula der Universiteit
op donderdag 5 december 2002, te 12:00 uur

door

Leonardus Martinus Franciscus Moonen

geboren te Weert

Promotor: prof. dr. P. Klint
Co-promotor: dr. A. van Deursen
Faculteit: Natuurwetenschappen, Wiskunde en Informatica

Cover: *Hexagon and Triangle in Transition*, Ad Dekkers, 1967.

CIP Data:

Moonen, Leon.
Exploring Software Systems / Leon Moonen
(PhD Thesis, University of Amsterdam, Faculty of Natural Sciences, Mathematics, and Computer Science, 2002)
Includes bibliographical references and summary in Dutch.
ISBN 90 5776 094 0
CR Subject Classification (1998): D.2.5, D.2.6, D.2.7, D.2.9, D.3.3, D.3.4, F.3.3, K.6.3.
Keywords: software evolution, software legibility, program comprehension, reverse engineering, program analysis, source model extraction & manipulation.

The work reported in this thesis has been carried out at the Center for Mathematics and Computer Science (CWI) in Amsterdam under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The publication of this thesis was made possible in part through support of the Software Improvement Group.

Copyright © 2002 by Leon Moonen.

To my parents

Preface

Here, in the first part of my thesis for all of you to read and the last part for me to write, I would like to thank all the people that have helped me getting to this point.

First of all, I am very grateful to my promotor, Paul Klint, for offering me the opportunity to do the research that is described in this thesis, and for offering me a deadline that enabled me to actually finish the thesis. I started working for Paul eight years ago while doing the practical work for my engineering degree and surely there must have been times that he thought to never get rid of me. Working for Paul has been a very pleasurable experience with lots of freedom for pursuing my own interests.

A special word of thanks goes out to my co-promotor, Arie van Deursen. His interest, encouragement and enthusiasm were always stimulating and he has taught me a lot about writing scientific papers and doing research in general. Besides being my supervisor and mentor, Arie has become a good friend with whom I could talk about life, Julia, books, Julia, the opera, and just now and then about his daughter Julia... I have truly enjoyed working together with Arie, and our cooperation turned out to be a productive one, which is shown by the fact that he was a co-author for half of the chapters of this thesis (Chapters 4, 5, 7, 9 & 10).

Tobias Kuipers has been there since I started studying at the UvA. His endless supply of interesting ideas and outspoken opinions were always good for long and fruitful discussions. We had great fun and hanging around with Tobias has taught me to express myself (although I will probably never become as good as him ;-). Tobias' research at CWI was closely related to mine: We both worked on software renovation issues and he is the co-author of Chapter 6 in which we connect my work on type inferencing to his work on concept analysis. Another important connection is our involvement in the Software Improvement Group (SIG), a spin-off company that we started together with Paul, Arie and others to transfer our research ideas into practice. The start of this company introduced some delays in the finishing of this thesis, not only because real work had to be done, but also because dealing with problems from practice provoked some additional research questions that just had to be pursued.

It was our experience with real-life software development in this company that inspired the work described in Chapter 9, which was co-authored by two of SIG's hardcore developers: Alex "*you're too technical for me*" van den Bergh and Gerard Kok.

Eva van Emden is the co-author of Chapter 8 on quality assurance using code smells. Eva visited us at CWI at a point where it was thought that I could finish my thesis by writing the introduction. However, our ideas on exploring smelly code fitted in too nicely to just ignore them and gave me another excuse for a short delay. Our common interests in electronic music, gothic literature, rock climbing and whitewater kayaking resulted in lots of highly enjoyable, off-topic, discussions and coffee breaks with demonstrations of rescue techniques. If I ever go paddle (or more likely, swim) a class V rapid, I hope Eva will be there to watch out for me.

I thank the members of my reading committee prof.dr. Hausi Müller, prof.dr.ir. Loe Feijs, prof.dr. Mike Papazoglou, prof.dr. Jan Bergstra, prof.dr. Peter van Emde Boas, and prof.dr. Martin Kersten, for their careful review of this thesis.

Merijn de Jonge and Joost Visser shared an office with me at CWI. They have both contributed to the work described in this thesis, by listening while my half-baked ideas took form and by developing elaborate tools that I could build upon for my experiments. Although we have never gotten around to writing a paper together, we have discussed lots of interesting issues that ought to be investigated further and written down (so let's *really* start project M335).

In addition to the people mentioned above, I would like to thank Jan Heering for his constructive comments on various chapters of this thesis, for his inquisitive remarks ("Maar wat *leer* ik nu van zo'n metafoor?") and for his willingness to discuss random issues, whether they concerned computer science or analog electrical circuitry.

The work presented in this thesis started in the "Programming Research Group" at the University of Amsterdam (UvA) and was finished in the "Interactive Software Development and Renovation" group at the Center for Mathematics and Computer Science (CWI). I would like to thank all colleagues that work or have worked in these groups for creating a friendly, open and stimulating environment to work in.

Outside the office, Miriam Egas and Daniel Dekkers showed the meaning of true friendship. They supported me in stressful times, made sure that I enjoyed the occasional rollercoaster and created something which feels like a second home in Eindhoven. Let's go water-skiing soon. I am especially grateful to Daniel for his approval of the cover picture. It really means a lot to me.

I am very lucky to share my life with Ivonne. None of this would have been possible without her confidence, encouragement, patience and unconditional love, even at times when I stayed up way too late to fix some tiny detail. Far too often she has heard me mumble the dreaded words "nog heel even dit afmaken" (just gotta fix this).

Finally, I would like to thank my parents, Ton and Anny Moonen, for always being there when I needed them and for supporting me in my choices and my study. I guess they never expected that this would be the outcome of the maths challenge they gave me 18 years ago. It is to them that I dedicate this thesis.

Leon Moonen

Amsterdam, the Netherlands
October 2002

Contents

Adventure Starts Here

Chapter 1
Introduction

explorer /ɪksplɔ̱ːrəʳ/. 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.[2] 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?

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 Nielson *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-

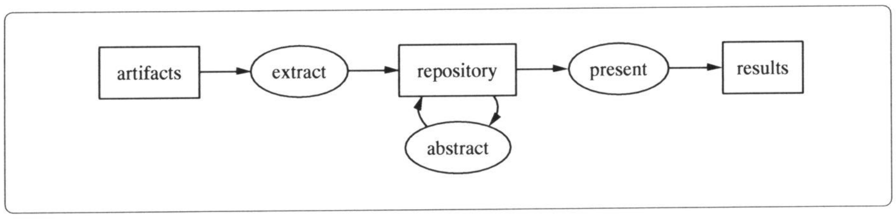


Figure 1.1: General architecture of a reverse engineering toolset.

tion using dynamic analysis of a system during its execution (e.g. using *instrumentation* to trace execution steps in a log). The resulting source models are stored in some kind of repository (either in memory, in files, or in a database management system).

2. *Abstraction:* In the next phase, new knowledge about the system is inferred by querying and manipulation of the data that is available in the repository. Generally, the results of this step are also stored in the repository to allow for an iterative abstraction process that can combine raw facts with previously inferred information to create new knowledge.

3. *Presentation:* In the final phase, the information in the repository is presented to the user in a suitable form. For a software redocumentation tool, this form may resemble printed technical documentation. In a development environment, one could think of decorating an editor with extra warnings which indicate that methods or classes possess certain bad characteristics.

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.

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

Chapter 3	L. Moonen. Lightweight Impact Analysis Using Island Grammars. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*. IEEE Computer Society Press, June 2002.
Chapter 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 Press, October 1998.
Chapter 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.
Chapter 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 Press, June 2000.
Chapter 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 Press, October 2000.
Chapter 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 Press, October 2002.
Chapter 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. This chapter will also appear in the book *eXtreme Programming Perspectives*, edited by M. Marchesi, G. Succi, D. Wells, and L. Williams. Addison-Wesley. Scheduled for release in August 2002.
Chapter 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.

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 Epictoid 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.

Part I

Island Grammars

CHAPTER 2

Generating Robust Parsers using Island Grammars

Source model extraction—the automated extraction of information from system artifacts—is a common phase in reverse engineering tools. One of the major challenges of this phase is creating extractors that can deal with irregularities in the artifacts that are typical for the reverse engineering domain (for example, syntactic errors, incomplete source code, language dialects and embedded languages).

This chapter proposes a solution in the form of **island grammars**, *a special kind of grammars that combine the detailed specification possibilities of grammars with the liberal behavior of lexical approaches. We show how island grammars 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. We conclude with a discussion of the development of* MANGROVE, *a generator for source model extractors based on island grammars and describe its application to a number of case studies. The work presented in this chapter was published earlier as [Moo01a].*

2.1 Introduction

Software engineers spend a large amount of their time on understanding the system that is being maintained (estimates of up to 50% are not uncommon). Consequently, much research is being invested in the development of tools that assist with such program understanding and program maintenance activities. The majority of these tools consist of three phases: (1) extraction of information (often referred to as *source models*) from the system's artifacts, (2) manipulation, querying and abstraction of source models, and (3) presentation of the results. This chapter focuses on the first phase: *extracting source models from system artifacts.*

One of the challenges reverse engineering tools have 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. Our goal is to obtain *robust* parsers that can handle artifacts with such irregularities. Examples of the kind of irregularities we want to deal with include:

Syntax errors: In a program maintenance environment, we want to be able to deal with systems containing syntax errors (e.g., browse or query code to fix those errors). Most parser based techniques will fail when encountering syntactic errors.

Completeness: The source code of a system may be incomplete. A typical situation is that some of the header files (or copybooks) of a system are lost or mutilated over the years, making a full reconstruction impossible.

Dialects: A legacy language like COBOL (but also a language like C) has a large number of, slightly different, vendor-specific dialects. Ideally, we can support them all. However, a parser for one dialect may not accept code written in another.

Embedded languages: Several programming languages have been upgraded with embedded languages for database access, transaction handling, screen definition, etc. COBOL examples include SQL, CICS, and IDMS. Whether we choose to analyze or to ignore such extensions, the extraction should not be hampered by them. However, a standard parser will.

Grammar availability: When supporting legacy systems, we will come across languages for which there is no grammar available. These can be proprietary languages, for which a grammar was never disclosed, or languages for which there never was a grammar since the parser (or processor) was hand-written. Reviving such grammars from scratch is expensive, and may not pay back at all.

Customer-specific idioms: Systems can use specific idioms (e.g., assigning values to "special" variables) in combination with libraries to interface with other systems, or to bypass limitations in a compiler or runtime system. Standard parsers will not recognize such customer-specific idioms and are generally not flexible enough to be made aware of them. An example regarding COBOL CALL analysis is shown in Section 2.2.1.

Preprocessing: Conceptual problems can arise with analysis of code that uses a preprocessor: Parsers usually read preprocessed code so the resulting models are based on preprocessed code. However, a maintainer's mental model is based on unpreprocessed code. It can be very hard to map these models onto another, especially when conditional compilation is used.

People have tried to bypass these problems by reusing an existing parser via a common exchange format (e.g., GXL [HWS00]), or via interface generation (for example, GENII [Dev99]). Although these are good solutions from an engineering perspective (you may not have to write a parser yourself) they do not *solve* the problems described above.

Others have proposed to use *lexical analysis* techniques to remedy these problems [MN96, CC00]. Lexical analysis provides a flexible and robust solution that can handle incomplete and syntactically incorrect code at the cost of losing some accuracy and completeness.

An additional advantage of lexical analysis is that it often takes less time to develop a solution based on lexical analysis than on syntactical analysis. It is tedious and expensive to write a parser for a language or to write a grammar that can be used to generate such a parser. For example, van den Brand *et al.* report a period of four months for the development of a fairly complete COBOL grammar [BSV97b].

This chapter proposes another solution to remedy these problems: we describe the use of *island grammars* to generate robust parsers that are used to build source model extractors. Island grammars are grammars that contain detailed productions (rules) describing the language constructs of interest, and generic productions that capture the remainder. Island grammars have been briefly sketched before in [DK99a, DKM00]. In this chapter, we present a more detailed account.

By generating parsers from island grammars, we combine the accuracy of syntactical analysis with the speed, flexibility and robustness of lexical analysis. The remainder of this chapter presents island grammars and their use in MANGROVE, a generator for source model extractors based on island grammars. We propose a reusable framework for defining island grammars and describe how the mapping from parse results to source models can be specified using patterns in a term rewriting language and in JAVA. We conclude with the application of MANGROVE in a number of case studies and a discussion of related work.

2.2 Island Grammars

Parsers for reverse engineering tools have a number of requirements. Most importantly, the parser should recognize certain *constructs of interest* in a given language. Additionally, the parser should be *robust*: it should not be obstructed by irregularities in the input. In this chapter, we study how such parsers can be generated from (context-free) grammar definitions.

Recall from compiler class that, given a language L_0, we can give a description of L_0 by defining a context-free grammar G such that the language $L(G)$ generated by G satisfies $L(G) = L_0$.[1] In order to satisfy the requirements stated above, we need to describe L_0 using a grammar that on the one hand generates more sentences than available in the actual language L_0 (namely also sentences with irregularities) but on the other hand should give an exact specification of the interesting parts of that language. This

[1] In short: if $G = (V, \Sigma, P, S)$ is a context-free grammar with sets of *non-terminals* V, *terminals* Σ and *productions* $P \subseteq (V \cup \Sigma)^* \times V$, a *start symbol* $S \in V$, and $V \cap \Sigma = \emptyset$, then a string $s \in \Sigma^*$ is a *sentence* of G, iff $S \xrightarrow{*} s$ (s can be derived from S by repeatedly applying productions from P). The *language* generated by G contains all sentences $L(G) = \{s \mid s \in \Sigma^* \wedge S \xrightarrow{*} s\}$. We refer to [Sud88, pp. 43–64] for more information.

is exactly what an island grammar amounts to, as follows from the following definition:

Definition 2.2.1 *An* island grammar *is a grammar that consists of two parts: (i) detailed productions describing certain constructs of interest (the* **islands***), and (ii) liberal productions that catch the remainder (the* **water***).*

or expressed in terms of language properties:

Definition 2.2.2 *Given a language L_0, a context free grammar $G = (V, \Sigma, P, S)$ such that $L(G) = L_0$ and a set of constructs of interest $I \subset \Sigma^*$ such that $\forall i \in I : \exists s_1, s_2 \in \Sigma^* : s_1\ i\ s_2 \in L(G)$. An island grammar $G_I = (V_I, \Sigma_I, P_I, S_I)$ for L_0 has the following properties:*

1. $L(G) \subset L(G_I)$ *G_I generates an extension of $L(G)$.*
2. $\forall i \in I : \exists v \in V_I : v \xrightarrow{*} i$
   $\exists s_3, s_4 \in \Sigma^* : s_3\ i\ s_4 \notin L(G) \wedge s_3\ i\ s_4 \in L(G_I)$
   G_I can recognize constructs of interest from I in at least one sentence that is not recognized by G.
3. $K(G) > K(G_I)$ *G has higher complexity than G_I.*[2]

Note that island grammars do not require the use of a particular grammar specification formalism or parsing technique. However, the limitations of the chosen formalism and technique may influence the island grammar. In this chapter, we express island grammars in SDF, a syntax definition formalism that is supported by *generalized LR* parsing [HHKR89, Vis97]. We benefit from the expressive power of this combo which makes development of island grammars easier. Other formalisms and parsing techniques can, and have been used. For example, JAVACC (the Java parser generator by MetaMata/Sun Microsystems) has been used for an island grammar developed together with our industrial partner, the Software Improvement Group, as part of their documentation generator DOCGEN [DK99a, DKM00]. The requirements originating from the LL parsing technique used in JAVACC made development and extension of this grammar unwieldy. The tooling described in the next section enables us to reimplement this grammar based on SDF and generalized LR parsing.

2.2.1 Island Grammar Example

Figures 2.1 and 2.2 show an example island grammar that describes COBOL CALL statements. The specification uses the modular syntax definition formalism SDF. Note that productions in SDF are reversed with respect to BNF: on the right-hand side of the arrow is the non-terminal that can be produced by the symbols on the left-hand side. Section 2.3.1 gives a short introduction to SDF.

The grammar contains three modules: The module Layout specifies the lexical non-terminal symbol LAYOUT containing whitespace characters. This symbol has

[2] The complexity of a context free language $K(G)$ can be computed by analyzing the productions of G. See [Gru73] for a detailed discussion.

```
module Layout                                         (1)
lexical syntax                                        (2)
  [\␣\t\n]            → LAYOUT                        (3)
module Water                                          (4)
imports Layout                                        (5)
context free syntax                                   (6)
  Chunk*              → Input                         (7)
  Water               → Chunk                         (8)
lexical syntax                                        (9)
  ~[\␣\t\n]+          → Water        {avoid}          (10)
```

Figure 2.1: Base for island grammars.

special meaning in our parsers since it can be recognized between any two symbols in a context-free production.

The module Water uses the definitions from module Layout (line 5) and adds two context-free non-terminals: the symbol Input that can be produced from a list of zero or more Chunks (line 7) and the symbol Chunk that can be produced from Water (line 8). Later, we will add more productions for Chunk, thus providing alternatives that can be recognized instead of Water. The lexical non-terminal Water consists of a list of one or more characters that are not whitespace (line 10). The attribute "{avoid}" prevents the parser from using this production if others are applicable. This allows us to specify *default behavior* that can be overridden by other productions (without generating ambiguities).

The grammar specified by module Water is extremely robust: it describes almost all programming languages. It is, however, not very useful by itself since the terminal symbols in a parsed sentence are indistinguishable. We can turn this into a useful grammar by adding *islands* that specify constructs of interest: The module Call adds such an island by specifying that a Chunk can also be produced by the literal `CALL` followed by an identifier (line 4). Identifiers are characters followed by zero or more characters or digits (line 7).

```
module Call                                           (1)
imports Water                                         (2)
context free syntax                                   (3)
  "CALL" Id           → Chunk        {cons(Call)}     (4)
lexical syntax                                        (6)
  [A-Z][A-Z0-9]*      → Id                            (7)
```

Figure 2.2: Cobol program calls.

This very simple grammar allows us to generate a parser that searches for program calls in COBOL code. Although this may not be a spectacular example (something similar could be done, for example, using a tool like `grep`), we will show below how easy it is to extend this grammar to do a much more complicated analysis. Furthermore, the modularity of SDF allows us to reuse the base grammar developed here for other island grammars.

Remember the customer specific idioms described in Section 2.1? We found a good example of that situation when analyzing a COBOL system where program calls were not made using the `CALL` statement but by setting a global variable and then calling a generic *call-handler*. This call-handler enabled the run-time system to dynamically load and execute the desired program (instead of static linking supported by the compiler). A standard call-graph extractor will not be able to generate useful graphs for such a system.

We can add support for that situation using the grammar module in Figure 2.3. Suppose the name of the call-handler is `HANDLER` and the name of the global variable is `CALLEE`. We specify an assignment to `CALLEE` as if it is a program call (line 4). Furthermore, we prevent the parser from recognizing calls to `HANDLER` using the "{reject}" attribute (line 5).

The "{cons(Call)}" attributes in Figures 2.2 and 2.3 are used to explicitly specify the constructor function that has to be used to create an abstract syntax tree. Using this attribute we can map different concrete syntax productions to the same abstract syntax. This will make processing easier.

Note the source for potential errors here: (1) when there are two subsequent assignments to `CALLEE` before the call-handler is called, both will be recognized as calls; (2) when the value in `CALLEE` is computed instead of assigned, it will not be recognized. These problems can be remediated in a back-end that does a more detailed (data flow) analysis. In practice, however, we found that such call-handlers were used in a disciplined manner following strict coding conventions, so these situations did not occur.

2.2.2 Island Grammar Applications

The employment of island grammars is especially suitable for reverse engineering (as opposed to, for example, compiler construction) since it takes maximum advantage of the fact that such applications generally do not need the complete parse tree. Particu-

```
module CallHandler                                              (1)
imports Call                                                    (2)
context free syntax                                             (3)
  "MOVE" Id "TO" "CALLEE"     → Chunk     {cons(Call)}          (4)
  "CALL" "HANDLER"            → Chunk     {reject}              (5)
```

Figure 2.3: Dealing with a call-handler.

larly analyzers that try to arrive at higher levels of abstraction (for example, architecture extraction) can profit from this early elimination of detail in the parsing phase.

By varying the amount and details in productions for the construct of interest, we can trade off accuracy, completeness and development speed. For example, it is possible to approach island grammars from a completely different side by starting with a complete grammar for a given language and extending that grammar with a number of liberal (*water*) productions. We will call such a grammar a *lake grammar*. This approach is typically useful to allow for arbitrary embedded code in the programs that can be processed by given tool. Furthermore, we can mix productions for water and islands to allow variations such as: *islands with lakes* to specify "nested" constructs such as conditional or iteration statements, and *lakes with islands* to combine extraction for a language with extraction for an embedded extension.

In our opinion, the main application area for island grammars is robust parser generation for source model extraction and simple analysis. Island grammars can be used for both local and non-local analysis. Obviously, grammars that only allow local analysis (for example, the CALL statements of Figure 2.2) will be simpler than those that allow non-local analysis. Additional work has to be done in the back end of a non-local analyzer to find and combine islands that "belong together".

The main advantage that island grammars have over lexical approaches is that it is much easier to use structure while specifying patterns (which requires state manipulation in a lexical approach). Moreover, solutions can easily be combined and are completely declarative making them easier to understand.

In theory, island grammars can be used for program transformations. Since the use is evidently restricted to the parts that are contained by the islands, applications are probably limited to local transformations. Examples one can think of include simple structure modifications, normalization of conditions, enforcement of some coding standards. In general, however, we believe that program transformations require more in depth knowledge of the source language than what is usually expressed in an island grammar.

2.2.3 Processing

There are a number of ways to process the parse trees obtained after parsing an input sentence. Initial observations indicate that in most island grammars, the Water symbols always occur in a sequence of symbols. Consequently, removing those subtrees from a parse tree does not invalidate the tree. Based on this observation, we have created a simple filter that removes all subtrees that have been parsed as *Water* from a parse tree. After applying this filter, processing the resulting term becomes both easier and faster (less input to consider). Simple analysis of the term can even be done using lexical techniques. Note that it is always possible to create grammars for which Water does not occur in a list context. Use of the filter will invalidate parse tree with respect to such grammars. This may or may not be a real problem depending on the processing that remains to be done on the tree.

Another way is to process the parse trees using hand-written C code. Currently, such processing is cumbersome but this might improve when supportive tooling becomes available that generates access functions on an AST level

In order to be able to create more involved source model extractors that are not hand-written in C, we have created MANGROVE, a generator for source model extractors. MANGROVE is described in the next section.

2.3 MANGROVE

MANGROVE is a generator for source model extractors based on island grammars. The design requirements were similar to those described by Murphy and Notkin for their lexical source model extractor [MN96]. The approach has to be:

- *Lightweight:* specification of new extractor should be small and relatively easy to write.
- *Flexible:* few constraints on structure of the artifact that is analyzed (possible to create analyzers for both source and structured data).
- *Tolerant:* few constrains on the condition of the artifact that is analyzed (possible to analyze code that cannot compile).

An overview of the MANGROVE architecture is given in Figure 2.4. Tools are drawn as ellipses, artifacts as boxes. The generation of a extractor is based on two types of input (the grey boxes in Figure 2.4): The first defines an island grammar describing the syntax of constructs that need to be recognized. It is used to generate an island parser; The second specifies the mapping of those constructs to the desired source model. It is used with the grammar to generate an extractor that reads the output of the island parser and converts it to the source model.

In contrast to most lexical approaches, our approach separates parsing and analysis instead of attaching semantic actions to the constructs to be recognized. This has the advantage that the resulting analyzers are easier to adapt and that it is easier to combine two existing analyzers into a new one. Most lexical analyzers are hard to adapt since the analysis logic is entangled with the constructs that have to be recognized. Combining two of these analyzers into a single new one is even more tricky.

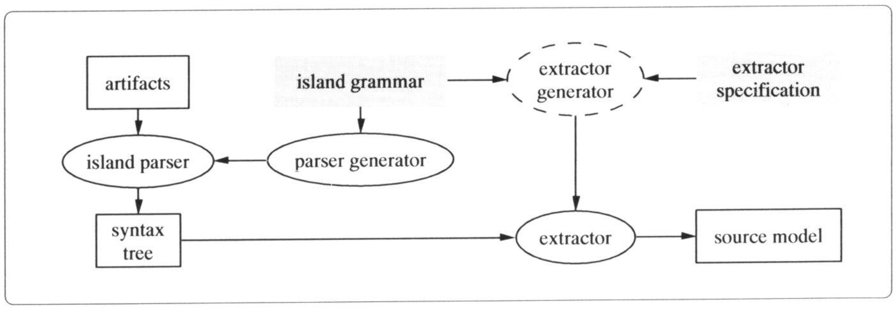


Figure 2.4: MANGROVE architecture.

The two inputs are generally small and easy to write down; therefore, we feel that our approach satisfies the lightweight requirement. The flexibility and robustness requirements are satisfied by using island grammars to generate the parser.

The *extractor generator* in Figure 2.4 is drawn with a dotted line to indicate that there are several possible instantiations. These allow the user to choose the language in which he describes the mapping of constructs on the source model. We have made two instantiations of this tool that are described below. One allows the user to write the mapping using traversals over the AST in Java, the other using concrete syntax patterns in a simple functional specification.

2.3.1 Syntax Definition in SDF

MANGROVE reads island grammars that are written in the syntax definition formalism SDF [HHKR89, Vis97]. These definitions combine the definition of lexical and context-free syntax in the same formalism. The definitions are purely declarative (as opposed to, for example, definitions in YACC that can use semantic actions to influence parsing) and describe both concrete and abstract syntax.

SDF definitions can be modular: productions for the same non-terminal can be distributed over different modules and a given module can reuse productions by *importing* the modules that define them. This allows for the definition of a base or kernel grammar that is extended by definitions in other modules. An example of this is module Water defined in Figure 2.1 that is extended by module Call in Figure 2.2.

SDF provides a number of operators to define optional symbols ($S?$), alternatives ($S_1|S_2$), iteration of symbols ($S+$ and $S*$), and more. These operators can be arbitrarily nested to describe more complex symbols. Furthermore, SDF provides a number of disambiguation constructs such as relative priorities between productions, preference attributes to indicate that a production should be preferred of avoided when alternatives exist, and associativity attributes for binary productions (for example, the left associative operator *op* can be defined as: $S\ op\ S \rightarrow S$ {left}).

SDF is supported by a parser generator that generates *generalized LR* (GLR) parsers. Generalized parsing allows definition of the complete class of context-free grammars instead of restricting it to a non-ambiguous subclass of the context-free grammars, such as the LL(k), LR(k) or LALR(1) class restrictions common to most other parser generators [Tom86, Rek92]. This allows for a more natural definition of the intended syntax because a grammar developer no longer needs to encode it in a restricted subclass. Moreover, since the full class of context-free grammars is closed under composition (the combination of two CF grammars is again a CF grammar), generalized parsing allows for better modularity and syntax reuse. For more information on SDF, we refer to [HHKR89, Vis97].

2.3.2 MANGROVE/JAVA

MANGROVE/JAVA allows the extractor builder to process the results of the island parser using the object-oriented programming language JAVA. An overview of the tool is given in Figure 2.5. Apart from the obvious advantage of being able to process using a mainstream object-oriented programming language, this also allows the tool builder to reuse the large amount of tools, libraries and interoperability techniques that are available for JAVA.

From an island grammar in SDF, we generate JAVA code for the construction, representation, and manipulation of syntax trees in an object-oriented style. The generated classes relate to the abstract syntax of the grammar using the following scheme: (i) for every non-terminal, an abstract class is generated and (ii) for every production, a concrete class is generated that refines the abstract class corresponding to the result of the production. Factory methods are generated to convert a parsed input string into an abstract syntax tree (object structure). Furthermore, several variants on the Visitor pattern are generated that provide tree traversals over these ASTs. We have reused JJFORESTER for the generation of this JAVA code [KV01].

The generated code can be extended by a tool builder to perform the actual mapping between the AST and the desired source model. This is done by refining the generated visitors and feeding them to the generated accept method of a given AST node. These accept methods perform the actual traversal over the AST and call visit methods defined in the visitor. This approach has the advantage that the user does not have to reconstruct the traversal behavior when refining visitors. Consequently, it is easier and less error-prone to write extensions and refinements of the generated code.

User extensions are compiled together with the generated code using a standard JAVA compiler to create an extractor (i.e., byte code that can be executed using the JAVA virtual machine). This extractor interfaces with the generated island parser using a utility that implodes the parse tree into an abstract syntax tree.

Example: Figure 2.6 presents an UML class diagram showing the classes that are generated for the island grammar presented in the COBOL program call example (Section 2.2.1). The grey class (CallCollect) was not generated but is an example of an analysis that can be added by a user. This class refines the standard visitor so that it collects the identifiers of all called programs. The JAVA code that implements this class is shown in Figure 2.7.

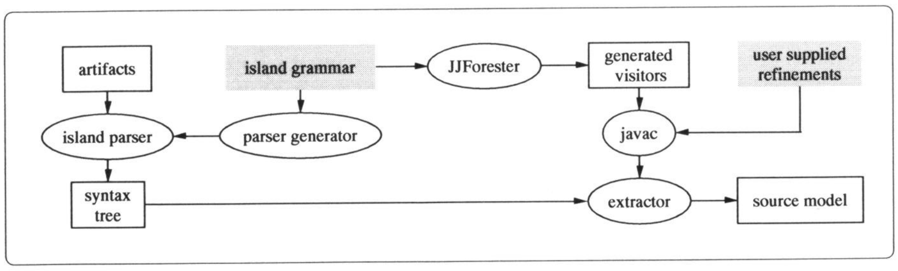


Figure 2.5: MANGROVE instantiation that allows processing in JAVA.

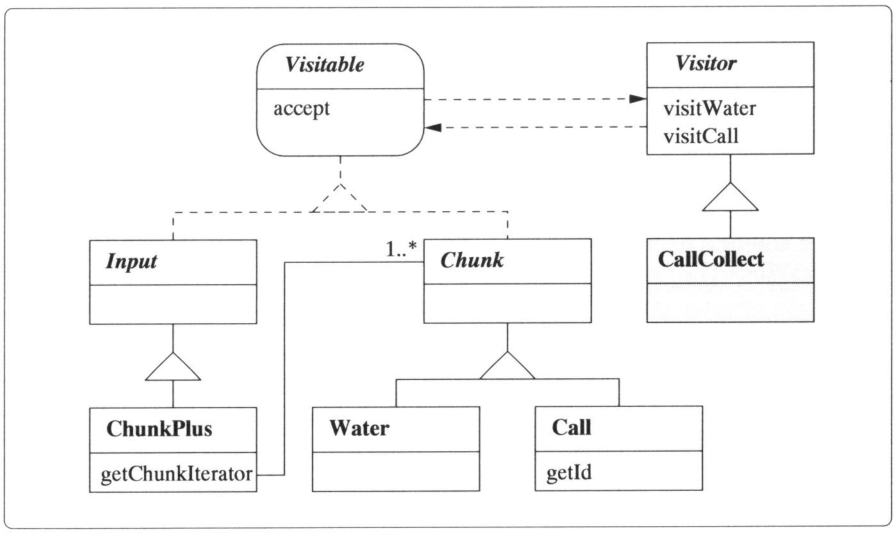


Figure 2.6: UML class diagram for Call collector.

2.3.3 MANGROVE/ASF

MANGROVE/ASF allows the extractor writer to process parse results in a functional fashion using the term rewriting language ASF [BHK89].

Programming in ASF is done by creating specifications that consist of a number of rewrite rules. These rules are defined using pattern matching on concrete syntax defined in an SDF grammar. The use of concrete syntax has the advantage that the extractor writer does not have to learn a new language for processing terms. The use of term rewriting allows for a natural expression of the translation of one language into another.

The combination of syntax definition formalism SDF and term rewriting language ASF is supported by the ASF+SDF Meta-Environment [Kli93, BDH$^+$01]. This environment generates parsers and syntax directed editors from SDF definitions and provides an interpreter and compiler for ASF specifications.

```
public class CallCollect extends Visitor {
    public Set set = new HashSet();
    public void visitCall(Call c) {
        set.add(c.getId());
    }
}
```

Figure 2.7: JAVA visitor for collecting program calls.

In Mangrove/Asf, we instantiate the extractor generator using the Asf+Sdf Meta-Environment. For an architectural overview, we refer to the Mangrove overview in Figure 2.4.

The Asf+Sdf Meta-Environment contains special support for the generation of term traversal functions [BKV01]. When a user attaches a "{traverse}" attribute to a production in Sdf, additional functionality is inferred that can perform a traversal of the first argument of the production. Conceptually, adding such an attribute is shorthand for adding a set of productions and rewrite rules (which can be calculated from the grammar). The default behavior of the generated rewrite rules is to do nothing. A user can override that behavior by adding a concrete rewrite rule for a particular (sub)term.

Example: Figures 2.8 and 2.9 show an example of the use of generated traversals for the program call example described in Section 2.2.1. Again, we will build a tool to collect the identifiers of all called programs. The grammar (Figure 2.8) defines two functions: one that we will use to start the traversal (line 4) and the actual traversal function in line 5. This traversal function has two arguments, the first contains the term to traverse, the second is the accumulator in which traversal results are gathered. The Asf equations in Figure 2.9 define the rewrite rules. We see that rule $[c_1]$ starts the traversal using a copy of the input and an empty accumulator. The other two rules contain patterns for which we want specific behavior: Rule $[c_2]$ specifies that whenever a CALL statement is matched with arbitrary identifier, we add that identifier to the accumulated set. Call-handlers are supported using rule $[c_3]$ that collects all

```
module CallCollect                                        (1)
imports CallHandler Set                                   (2)
context free syntax                                       (3)
  collect( Input )          → Set                         (4)
  collect( Input , Set )    → Set        {traverse}       (5)
variables                                                 (6)
  "in"                      → Input                       (7)
  "set"                     → Set                         (8)
```

Figure 2.8: Grammar for collecting program calls.

```
equations
[c1] collect( in )                       = collect( in , {} )
[c2] collect( CALL id , set )            = {id} ∪ set
[c3] collect( MOVE id TO CALLEE , set )  = {id} ∪ set
```

Figure 2.9: Equations for collecting program calls.

identifiers that are assigned to the CALLEE variable.

2.4 Case Studies

We have done a number of case studies to validate our hypothesis that island grammars can be used to create robust parsers that allow for construction of lightweight, flexible and tolerant source model extractors.

The first case uses island grammars to build an analyzer that computes the cyclomatic complexity of Cobol programs. The second case was done in cooperation with the Software Improvement Group and involves the creation of a source model extractor for Uniface systems.

2.4.1 Cobol Cyclomatic Complexity

McCabe's *cyclomatic complexity* measure [McC76] is one of the better known software metrics that can be computed from source code. In this case study we build a simple analyzer that computes this complexity measure for Cobol programs using island grammars.

The cyclomatic complexity metric is based on the control graph of the program. It computes the number of linearly independent control flow graphs using the number of nodes (n) and edges (e) in a control flow graph. For a graph with n nodes and e edges, McCabe defines the cyclomatic complexity as $G(v) = e - n + 2$.

However, there is a simpler definition that does not require us to construct a control flow graph in advance. In the NIST report on structured testing, McCabe defines the cyclomatic complexity by counting the number of decision predicates in the code [WM96]. We will use this latter approach in this case. Our analyzer basically traverses a parse tree and counts occurrences of decision predicates. We show how we use Mangrove/Java to build the analyzer in four steps.

First, we create an island grammar for Cobol that describes the constructs that can influence the cyclomatic complexity. In the case of Cobol, these are standard constructs like IF-THEN, REPEAT-UNTIL, and EVALUATE-WHEN (Cobol's case statement) but also constructs like GO-DEPENDING that jumps to one of a list of locations based on the value of a variable. Other constructs of interest are predicates that surround code that has to be executed in case of errors, such as ON-ERROR and ON-OVERFLOW for computational statements, and INVALID-KEY and AT-END for access to flat-file databases.

Note that we have to take special precautions to prevent occurrences of these constructs in strings or comments from being recognized as real occurrences (so called *false positives*). This can be done by adding specific productions to the island grammar that specify that strings should be recognized as water and that comments should be considered LAYOUT. An example of such productions can be found in Figure 2.10.

Second, a parser and Java classes are generated from this island grammar as described in Section 2.3.

Third, we refine the generated visitor so that computes the cyclomatic complexity during traversal of the parse tree. This is done by incrementing a counter every time the abstract syntax tree contains one of the complexity increasing constructs that were specified in the island grammar.

Finally, we compile the code to build an executable analyzer. The parts that we had to write to create such an analyzer are small and easy to write: construction, testing and refinement took 4–5 hours. The grammar consists of 17 productions, 10 for describing constructs of interest, 4 we reused from the base grammar of Figure 2.1, and 3 were added to prevent false positives. The Java code that refines the generated visitor contains one integer field (the complexity counter) and seven methods that each perform exactly one statement: increment the complexity.

We have applied our analyzer to a number of Cobol systems (each around 100,000 lines) that were written in different dialects and contained various extensions (SQL, CICS, IMS). These irregularities posed no problems for the analysis. Initial results show that the performance is good but should be measured in more detail. For example, the implosion prototype that converts parse trees to ASTs is slow for very large inputs. A reimplementation will solve these issues.

2.4.2 Uniface Component Coupling

In a case study performed in cooperation with the Software Improvement Group (SIG) we developed an island grammar and source model extractor to parse Uniface components and collect facts about the coupling between them.

Uniface is a 4GL application development environment that is marketed by Compuware [Com00]. It allows for the development of both conventional and web-based applications. The application development is model-driven and component-based. Developers create models of business processes. These models are used to generate components that inherit properties from the model. Whenever the model is changed, components are updated accordingly. To eliminate the need to build systems from scratch, developers can reuse components from other systems and standard libraries. Components contain operations that specify behavior. Components can interoperate with each other by activating operations in other components (similar to objects and methods in an object-oriented setting).

To get insight in Uniface systems, a SIG customer would like to get information about the components in a system and the coupling between them. To collect this information, we have built a source model extractor that analyses Uniface components and gathers facts about the activation of other components and of the activation parameters.

```
module StringsAsWater                                (1)
lexical syntax                                       (2)
  [\"] ~[\"]* [\"]              → Water              (3)
```

Figure 2.10: Strings as water.

The extractor was generated using an island grammar that describes module activation and parameter passing in UNIFACE. This grammar extends the base grammar from Figure 2.1 and was developed without prior knowledge of UNIFACE (but with help of `activate` documentation). It took approximately one day to develop, test and refine the island grammar and about the same amount of time to develop the source model mapping in JAVA.

The complete island grammar contains 38 productions, including the base grammar and productions to prevent false positives. This relatively high number is influenced by the act that UNIFACE is case insensitive, thus our grammar contains a number of productions whose sole purpose is to specify case insensitive variants of keywords that have to be recognized.

The resulting source model extractor can process both UNIFACE source listings and XML dumps of modules. The extractor emits a source model that describes component coupling in textual or in GXL format [HWS00].

2.5 Discussion

2.5.1 Expressive Power

Island grammars do not depend on a particular grammar specification formalism or parsing technique. However, the expressive power of an island grammar is limited by the chosen syntax definition formalism and more important by the chosen parsing technique. In MANGROVE, we have chosen to express island grammars in SDF, a syntax definition formalism that is supported by *generalized LR* parsing techniques. Since we inherit the expressive power, we can express the complete class of context free languages using our island grammars.

The different MANGROVE instantiations allow an extractor writer to choose a processing language that fits his needs. The JAVA instantiation enables processing in a mainstream object-oriented programming language and allows reuse of the large amount of tools, libraries and interoperability techniques that are available for JAVA. The ASF instantiation allows processing using term rewriting with patterns over concrete syntax. This has the advantage that the extractor writer does not have to learn a new language and term rewriting allows for natural expression of translation between languages.

2.5.2 Accuracy

Island grammars do not give a restrictive description of the language that is analyzed. On the one hand, we consider this an advantage since this is, after all, the property that allows for irregularities, releases structural requirements on the artifacts and increases development speed. On the other hand, however, this lack of detail may result in erroneous results.

We distinguish two kinds of extraction errors: (i) *false positives* occur when the grammar allows constructs to be recognized in places where they should not have been recognized. (ii) *false negatives* occur when the grammar is too restrictive and does not allow constructs to be recognized in places where they should be recognized.

False positives can be solved by extending the part of the grammar that specifies Water. For example, false recognition of constructs inside of strings can be prevented by adding a production that specifies string syntax as Water. Figure 2.10 gives a simple example of such a specification. It specifies strings as starting with a double quote, a number of characters and ending with a double quote.

False negatives are not that straightforward to solve. One needs to reconsider the grammar and look for productions that are too restrictive. A common source of false negatives are "nested" constructs, for example statements such as `if-then` and `while-do` that contain statements themselves.

2.6 Related Work

Related work can be divided into methods that perform lexical analysis and syntactical analysis. Another division comes from application domain with research focus-sing on computer language processing or on natural language processing.

2.6.1 Lexical Analysis

Several tools are available that perform lexical analysis of textual files. The most well-known tool is probably `grep` and its variants (`fgrep`, `egrep`, `agrep`, etc.) that allows one to search text for strings matching a regular expression. These tools generally give little to no support to process the matched strings, they just print matching lines.

This kind of support is available in more advanced text processing languages such as AWK [AKW88] and PERL [WS91], and in the LEX scanner generator [LS75] that allow a user to execute certain actions when a specific expression is matched. TLEX provides a pattern matching and parsing library for C++ that generates parse trees for the strings that match a regular expression [Kea91].

2.6.2 Hierarchical Lexical Analysis

Murphy and Notkin describe the Lexical Source Model Extractor (LSME) [MN96]. Their approach uses a set of hierarchically related regular expressions to describe language constructs that have to be mapped to the source model. By using hierarchical patterns they avoid some of the pitfalls of plain lexical patterns but maintain the flexibility and robustness of that approach.

The MultiLex system of Cox and Clarke [CC00] uses a similar hierarchical approach. The main difference with LSME is that it focuses at extracting information at the abstract syntax tree level whereas LSME extracts higher level source models.

This hierarchical technique is related to work in computational linguistics that divides natural language into chunks that can be recognized using a finite-state cascade parser [Abn96].

2.6.3 Syntactic Matching

Parser based approaches are used to increase the accuracy and level of detail that can be expressed. Syntactic matchers create a syntax tree of the input and allow the user to traverse, query or match the tree to look for certain patterns. This relieves them from having to handle all aspects of a language and focus on interesting parts.

Systems in this category are A* [LR95] that provide traversals over parse trees with AWK-like pattern matching and processing, TAWK [GAM96] that provides similar operations on abstract syntax trees with processing in C.

Other tools support querying of the abstract syntax trees such as GENOA [Dev99] that uses its own traversal language, REFINE [MNB$^+$94] that allows queries in first order logic and SCRUPLE [PP94] that allows queries using concrete syntax.

The disadvantage of these systems is that they are all based on a full parse of the complete language making it hard/impossible to deal with incomplete sources, dialects or syntax errors. However, with the proper amount of interfacing, it should be possible to connect them to the island parsers we generate which would remove such problems.

2.6.4 Fuzzy Parsing

The notion of *fuzzy parsing* comes in two flavors. The first flavor are parsers that recognize a sentence as belonging to a language with a certain degree of correctness (thus allowing for grammatical errors) [LZ69]. This type of fuzzy parsers is mainly used in computational linguistics for natural language processing. Productions in a *fuzzy grammar* are annotated with correctness degrees that are used to assess the quality of the input sentence. This can be used to model grammatical errors by adding special productions with a correctness degree less than 1 to an ordinary grammar. For more information, we refer to [Asv96].

The second flavor of fuzzy parsers are parsers that are able to discard tokens and recognize only certain parts of a programming language [Kop97]. The SNIFF programming environment was the first to use this kind of fuzzy parsing [Bis92]. Since then, it has been used in a number of other programming environments and program browsers such as: CSCOPE,[3] SOURCE NAVIGATOR,[4] SOURCE EXPLORER,[5] and the CRTAGS tool.[6] These fuzzy parsers are hand crafted to perform a specific task. They focus mainly on fuzzy parsing C and C++ to support program browsing. Typically this involves extracting information regarding references to a symbol, global definitions, functions calls, file includes, etc.

2.6.5 Parser Reuse

Some approaches address the problems associated with parser or grammar development by reusing existing parsers (for example, in GENOA/GENII [Dev99]). Others reuse

[3] http://cscope.sourceforge.net/
[4] http://sources.redhat.com/sourcenav/
[5] http://www.intland.com/
[6] http://www.vital.com/crtags.html

or retrieve grammars that are used in existing tools [SV99]. However, both approaches ignore the fact that the structure of a grammar used in a tool is often tightly coupled to the design of that tool. Another tool may need a completely different grammar. Such parser reuse problems were also signaled by Reubenstein *et al.* [RPR93]. Furthermore, this does not solve the robustness issues (dealing with missing code, embedded extensions or syntactical errors).

2.6.6 Island Parsing

The term island parsing is also used in computational linguistics (for example [Car83, SFI88]). However, this is different notion referring to island parsers that start at some point in a sentence (by recognizing an island) and parse the complete sentence by extending that island to the left and right (in contrast to left-to-right scanning done by LL and LR parsers). This technique is used for example for speech recognition. A similar approach has been applied by Rekers and Koorn for computer languages to provide error recovery and completion in syntax directed editors [RK91].

2.6.7 Island Grammars

The term island grammars was coined in [DK99a] which provides an informal definition and small example but does not present a detailed discussion, nor does it describe tool support. We try to fill those gaps by improving the definition, describing properties of island grammars and providing a number of detailed examples that result in a reusable framework for island grammar definitions. Furthermore, we present a generator for source model extractors based on island grammars that supports various programming languages and show how it can be used in a number of case studies. A case study for COBOL island grammars is described in [Ver00].

2.7 Conclusions

Robust parsing is a prerequisite for most reverse engineering tools. This chapter shows that island grammars can be used to generate such parsers. The generated parsers combine the accuracy of syntactical analysis with the speed, flexibility and tolerance usually only found in lexical analysis.

Contributions of this chapter are the extension of previous work on island grammars [DK99a, DKM00] with a detailed discussion and definition of island grammars. We present MANGROVE, a generator for source model extractors based on island grammars. We provide a reusable framework for the definition of island grammars in syntax definition formalism SDF and support various processing languages allowing a developer to pick the language that fits his needs. We have shown how MANGROVE supports JAVA and ASF programmers by providing generated traversals that ease the mapping from parse results to source models. We report on the application of MANGROVE to a number of case studies and provide a detailed discussion of related work.

The combination of island grammars with generated traversals combines 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 resulting in a *lightweight*, *flexible* and *tolerant* approach.

Acknowledgments

The author would like to thank Mark van den Brand, Tobias Kuipers, and Joost Visser for fruitful discussions. Arie van Deursen, Jan Heering and Paul Klint provided valuable feedback on earlier versions of this chapter.

Chapter 3

Lightweight Impact Analysis using Island Grammars

Change impact analysis is needed for the planning and estimation of software maintenance projects. Traditional impact analysis techniques tend to be too expensive for this phase, so there is need for more lightweight approaches.

In this chapter, we present a technique for the generation of lightweight impact analyzers from island grammars. We demonstrate this technique using a real-world case study in which we describe how island grammars can be used to find account numbers in the software portfolio of a large bank. We show how we have implemented our impact analyzer using generative programming. The work presented in this chapter was published earlier as [Moo02].

3.1 Introduction

Estimates indicate that 70% of software budgets are spent on software maintenance [Ben90]. 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]. Consequently, research that addresses techniques to assist maintainers in performing these tasks can make an important contribution.

A significant part of the program understanding research focuses on generic tools such as program browsers and documentation generators. These tools generally try to provide various means of querying or navigating through a software system that can be used by maintainers to answer their questions. There are obvious advantages to such a generic approach: it offers wide applicability, and it is easy to see cost-performance benefits of such tools.

However, we think that there is also a need for program understanding tools that are more tailored towards the questions to be answered. These tools should generate detailed reports or browsers that allow a maintainer to understand code with respect to such a specific question.

A typical task that would benefit from such problem-directed tooling is, for example, assessing the costs of mass change project such as Euro-conversion or database migration. This typically boils down to estimating questions like: How many systems are affected? How much code needs to be changed? Where do we need to make changes? Finding the answers to such questions is the domain of *(software change) impact analysis* [BA96].

One would expect that making such estimates is relatively cheap; It is hard to justify that the costs of estimating a mass change project are similar to performing the project. However, that is exactly what would happen if those estimates were based on a full blown impact analysis. Performing such an analysis would take almost the same amount of time and resources as the actual project. Consequently, a more *lightweight form of impact analysis* is needed.

A common approach for achieving lightweightness is based on the use of lexical analysis [DK98]. This has several advantages: lexical analysis is a flexible and robust solution that can handle incomplete and syntactically incorrect code. Additionally, it often takes little time to develop solutions based on lexical tooling. Unfortunately, there are also some serious drawbacks: Lexical analysis tends to be sensitive to the layout of the code that is being analyzed, for example, a simple newline may prevent recognition of a language feature. Furthermore, it is hard to write lexical analysers that take the structure of a language into account. Consequently, lexical analysis results typically have lower accuracy and completeness than those of syntactical analysis.

We think that an approach based on island grammars is better suited for creating lightweight impact analyzers. In this chapter, we investigate this hypothesis. We will do this using a case study in which we revisit a project that was performed earlier by our spin-off, the Software Improvement Group, for one of their customers.

This project involved estimating in which parts of the software portfolio of a large Dutch bank changes have to be made when converting their 9-digit account numbers into 10-digit account numbers. The customer was interested in a quick-scan of their complete software portfolio for planning and estimation purposes. This portfolio consists of 200 systems containing a total of 50,000,000 lines of COBOL code.

The chapter is organized as follows: Section 3.2 gives an overview of the problem and Section 3.3 sketches the impact analysis that is needed to solve this problem. Island grammars are described in Section 3.4, followed by a discussion of how the impact analysis is translated into island grammars in Section 3.5. The implementation of our analyzer is described in Section 3.6. Section 3.7 generalizes our approach to other applications . Finally, Sections 3.8 and 3.9 summarize related work, discuss future extensions and draw conclusions.

3.2 Problem Description

Currently, most Dutch banks use client account numbers that consist of 9 digits. Collective agreements ensure that each number is used uniquely and each bank typically uses certain sub-ranges of the spectrum. The pool of unassigned account numbers is managed by a subsidiary where banks can apply for free numbers. Since this supply of unused numbers is running out, the banks have decided that they will convert their systems to account numbers that consist of 10 digits (by prefixing existing numbers with '0' and using the prefixes '1...9' for fresh numbers).

This poses several questions for managers that are responsible for the software portfolio of a bank: How much of our portfolio is affected by this decision? For a given system, where are the parts that need to be changed? How many of these changes can be done automatically?

In our case study, we investigate if it is possible to answer such questions using an impact analysis technique based on island grammars. It is important to keep in mind that we are looking for a *lightweight* technique that can be used to analyze the impact on the complete software portfolio. The goal is to enable correct *estimation and planning* of the next steps in this mass change project. Consequently, our focus is more on short development time and enabling quick feedback for the complete portfolio, rather than on the detailed and complete impact analysis that would be needed to actually remedy the situation.

Furthermore, since the need for this conversion has been known for some time, some of the newer (or updated) parts of the software portfolio are already prepared for 10-digit account numbers. An important aspect of this study is that we need to handle code that contains a mixture of "good" and "bad" account numbers, and that we need to distinguish between them in order to provide correct estimates.

3.3 Impact Analysis Approach

This section describes how we want to perform the impact analysis that was described in the previous section. In the next section, we will describe how we have implemented it.

3.3.1 Patterns

The case study started with talking to (representatives of) the maintainers of the software to see how they would normally perform this kind of impact analysis. From these discussions, it turned out that it was possible to search the system's artifacts for variables that might represent bank account numbers. This search is partially based on pattern matching on the names of the variables, so together with the maintainers, a list of patterns was compiled that would signal bank account numbers in the software. The starting point for the compilation of such a list is the organization's data dictionary. Typical examples of patterns in this list are `ACCOUNTNR`, `ACCNR`, `ACC-NO` and `GNR` (that last one is used for giro number).[1]

[1] The examples in this chapter were taken from a Dutch software system. Although we have translated variable names into English, some names or abbreviations may look strange or uncommon since there is no good translation. Most notably is "giro", which is a bank transfer service in Europe.

3.3.2 Classification

Besides the variable name, also the type of the variable plays an important role in the analysis. We would like to distinguish between variables with a *9-digit* type that need to be changed, and variables with a *10-digit* type that are already correct. Unfortunately, COBOL does not have a real type system. Instead, with each variable declaration, a description of the memory layout that this variable uses is given (so called *pictures* or *picture clauses*). Pictures give a character-by-character definition of the format of variable. The characters have special meanings. For example, 'X' is used to denote a memory position that can hold an alphanumeric character; '9' is used for a numeric characters, and many others exist. Typical bank account numbers may be described in COBOL as follows:

```
01  ACCNR        PIC 9999999999.
01  FMT-ACCNR    PIC 99.99.99.999.
01  ACCOUNTNR    PIC 9(10).
```

The first line describes a variable with name ACCNR that consists of 9 digits (the picture 9 indicates 1 digit, 99 indicates two digits, and so on). The second line declares a variable FMT-ACCNR consisting of 9 digits but formatted using dots. The last example, shows a variable with name ACCOUNTNR consisting of 10 digits (the number between brackets indicates repetition).

We will classify the variables with matching names based on their picture clauses. We distinguish four classes:

A. *9-digit* variables with numeric pictures such as 999999999, 99.99.99.999, 9(9), and alphanumeric pictures such as X(9), and XX.XX.XX.XXX.

B. *10-digit* variables with numeric pictures such as 9999999999, 999.99.99.999, 9(10), and alphanumeric pictures such as X(10), and XXX.XX.XX.XXX.

C. *Record* variables do not have their own picture description but consist of a number of sub-fields. These sub-fields can be used, for example, to address parts of an account number (some banks use the first 4 digits of account numbers to identify the branch where this account was opened).

D. *Other* variables whose names match with the patterns but whose pictures do not fall in the above classes.

3.3.3 Anti-patterns

As described in Section 3.3.1, we start with a number of patterns that might indicate that a variable is used as an account number. However, there are a number of variables that have names that match with these patterns but we know (for example, from code inspection) that they are not used for account numbers. We call such variables *false positives*. We have taken the following steps to reduce false positives:

- During the project, a number of *anti-patterns* have been identified that match with field identifiers that are certainly not used for account numbers;

- Whenever a variable name matches both a pattern and an anti-pattern, that variable is rejected. When a variable name matches a pattern and none of the anti-patterns, it is accepted.

This process can be applied iteratively: whenever inspection of the results shows false positives, anti-patterns are added and the analysis is repeated. Thus, the precision of the analysis can be increased by investing in these iterations.

3.3.4 Presentation

We will report our findings using hypertext documentation consisting of: (i) pages displaying statistics and pie-charts summarizing the analysis results, and (ii) hyperlinked and pretty-printed artifacts that will lead the maintainer to all affected sites (i.e., all occurrences of account numbers). We add this second type of reports since they allow the maintainer to inspect analysis results and check hypotheses about impact. Furthermore, they are useful for identifying false positives and adding anti-patterns. The account numbers in the hyperlinked code are colored to show their classification: e.g., red for 9-digits, green for 10 digits, etc.

3.3.5 Tool Support

We have created ISCAN, a lightweight impact analysis tool to derive the described information. The basic structure of this tool is depicted in Figure 3.1. It follows the extract-query-view approach quite common to reverse engineering tools.

We start with parsing the system artifacts using an island parser. The parse results are processed in two ways:

Source Model Extraction: we extract source models describing for each artifact, the account numbers that were found, their classification and details about their origins (file and position information of the actual code). This data will be used later on for statistics and pie charts. The origins can be used for hyperlinking the results.

Island Markup: to enable problem-directed pretty-printing, we add markup to the artifacts, tagging all account numbers for later reference.

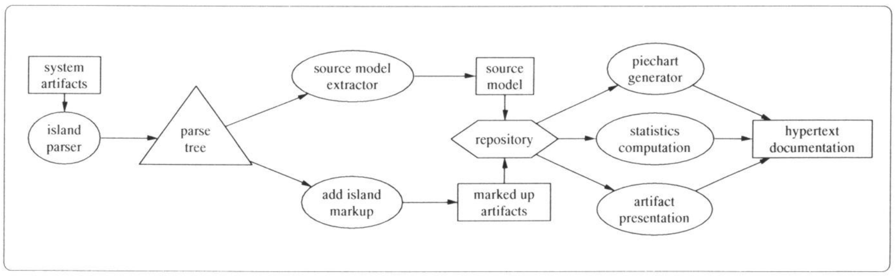


Figure 3.1: ISCAN architecture.

The results are stored in a repository which is used by a number of tools: a statistics tool queries the source models to generate statistical overviews, summarizing the account numbers per program, per system and in the complete portfolio. Another tool generates pie-charts that give a different view of the impact and affected code. A third tool presents the artifacts as a hyperlinked website. This tool uses the markup to pretty-print the code, visualizing the classification with different colors. Furthermore, it generates cross-references such as tables of content and indexes.

3.4 Island Grammars

One of the challenges of building a lightweight impact analysis tool is parsing a system's artifacts to extract the information we need. There are a number of reasons why it is hard (or even impossible) to parse the artifacts using common parser based approaches:

Grammar availability: We might want to analyze legacy systems written in a language for which there is no grammar available. Writing such grammars from scratch is tedious and expensive, and may not pay back at all. For example, van den Brand *et al.* report a period of four months for the development of a fairly complete COBOL grammar [BSV97b].

Completeness: The source code of a system may be incomplete. For example, some of the header files (or copybooks) may be lost making a full reconstruction impossible, or collecting all files may be too time consuming, making it unfeasible for cost estimation purposes only.

Dialects: Legacy languages such as COBOL (but also languages like C) have a number of, slightly different, vendor-specific dialects. A parser for one dialect may not accept code written in another.

Embedded languages: Several programming languages have been upgraded with embedded languages for database access, transaction handling, screen definition, etc. We might want to consider both languages in our analysis. Most parser based approaches have difficulties with that.

Customer-specific idioms: Some systems use specific idioms (e.g., assigning values to "special" variables) in combination with libraries to interface with other systems, or to bypass limitations in a compiler or runtime system. A standard parser will not recognize such constructions.

Preprocessing: The use of preprocessor directives can hinder parsing but analyzing already preprocessed code might give results that are not expected by the maintainer (since his mental views are based on unprocessed code).

It has been proposed to use *lexical analysis* techniques to remedy these problems [MN96, DK98]. Lexical analysis provides a flexible and robust solution that can handle incomplete and syntactically incorrect code. Additionally, it often takes less time to develop a lexical analyzer than a syntactical one.

However, there are also disadvantages to a lexical approach: the analysis results typically have lower accuracy and completeness than those of syntactical analysis. Lower accuracy means an increase of false positives (analysis finds properties for code which it does not have in reality). Lower completeness means an increase of false negatives (analysis misses properties that are present in reality).

In this chapter, we set out to use syntactical analysis based on *island grammars* (described in Chapter 2) to remedy these problems.

Definition 3.4.1 *An* island grammar *is a grammar that consists of two parts: (i) detailed productions that describe the language constructs that we are particularly interested in (so called* ***islands****), and (ii) liberal productions that catch the remainder of the input (so called* ***water****).*

In a way, island grammars mix the behavior of parsing with that of lexical approaches by analyzing the interesting parts of a grammar and brushing aside the non-interesting parts. By doing that, they combine the accuracy of syntactical analysis with the speed, flexibility and robustness of lexical analysis. Figure 3.2 gives an overview of the approaches.

Note that island grammars do not require the use of a particular grammar specification formalism or parsing technique. However, the limitations of the chosen formalism and technique may influence the island grammar. In this chapter, we express island grammars in SDF, a syntax definition formalism that is supported by *generalized LR* parsing [HHKR89, Vis97]. We benefit from the expressive power of this combination which makes development of island grammars easier. Other formalisms and parsing techniques can, and have been used. For example, JAVACC (the Java parser generator by MetaMata/Sun Microsystems) has been used for an island grammar developed together with our industrial spin-off, the Software Improvement Group, as part of the documentation generator DOCGEN [DK99a]. The requirements originating from the LL parsing technique used in JAVACC made development and extension of this grammar unwieldy. The tooling described in Chapter 2 enables re-implementation based on SDF and generalized LR parsing.

	lexical	syntactical analysis	
	analysis	full grammar	island grammar
accurate	–	+	+
complete	–	+	+
flexible	+	–	+
robust	+	–	+

Figure 3.2: Lexical vs. syntactical analysis.

3.4.1 Syntax Definition in SDF

Before we continue with an island grammar example, we give a short overview of the syntax definition formalism SDF. Syntax definitions in SDF combine the definition of lexical and context-free syntax in the same formalism. The definitions are purely declarative (e.g., as opposed to definitions in YACC that can use semantic actions to influence parsing) and describe both concrete and abstract syntax.

SDF definitions can be modular: productions for the same non-terminal can be distributed over different modules and a given module can reuse productions by *importing* the modules that define them. This allows for the definition of a base or kernel grammar that is extended by definitions in other modules. An example of this is module Water in Figure 3.3 that is extended by module DataFields in Figure 3.4.

SDF provides a number of operators to define optional symbols ($S?$), alternatives ($S_1|S_2$), iteration of symbols ($S+$ and $S*$), and more. These operators can be arbitrarily nested to describe more complex symbols. Furthermore, SDF provides a number of disambiguation constructs such as relative priorities between productions, preference attributes to indicate that a production should be preferred or avoided when alternatives exist, and associativity attributes for binary productions (for example, $S\ op\ S \rightarrow S$ {left}). Productions can be labeled with identifiers using the {cons} attribute. These labels appear in the parse tree so we can see which production was used to construct a given (sub)term.

SDF is supported by a parser generator that produces *generalized LR* (GLR) parsers. Generalized parsing allows definition of the complete class of context-free grammars instead of restricting it to a non-ambiguous subclass such as LL(k), LR(k) or LALR(1), which is common to most other parser generators [Tom86, Rek92]. This allows for a more natural definition of the intended syntax because a grammar developer no longer needs to encode it in a restricted subclass. Moreover, since the full class of context-free grammars is closed under composition (unlike restricted subclasses), generalized parsing allows for better modularity and syntax reuse. For more information on SDF, we refer to [HHKR89, Vis97].

3.4.2 Island Grammar Example

Figures 3.3 and 3.4 show an example island grammar that describes COBOL data fields. Note that productions in SDF are reversed with respect to BNF: on the right-hand side of the arrow is the non-terminal that can be produced by the symbols on the left-hand side.

The grammar contains three modules: The module Layout specifies the lexical non-terminal symbol LAYOUT containing white-space characters. This symbol has special meaning in our parsers since it can be recognized between any two symbols in a context-free production.

The module Water uses definitions from module Layout (line 5) and adds two context-free non-terminals: the symbol Input that can be produced from a list of zero or more Chunks (line 7) and the symbol Chunk that can be produced from Water (line 8). Later, we will add more productions for Chunk, thus providing alternatives that

```
module Layout                                              (1)
lexical syntax                                             (2)
  [\␣\t\n]              → LAYOUT                           (3)

module Water                                               (4)
imports Layout                                             (5)
context free syntax                                        (6)
  Chunk*                → Input                            (7)
  Water                 → Chunk                            (8)
lexical syntax                                             (9)
  ~[\␣\t\n]+            → Water        {avoid}             (10)
```

Figure 3.3: Base for island grammars.

can be recognized instead of Water. The lexical non-terminal Water consists of a list of one or more characters that are not white-space (line 10). The attribute "{avoid}" prevents the parser from using this production if others are applicable. This allows us to specify *default behavior* that can be overridden by other productions (without generating ambiguities).

The grammar specified by module Water is extremely robust: it describes almost all programming languages. It is, however, not very useful by itself since the terminal symbols in a parsed sentence are indistinguishable. We can turn this into a useful grammar by adding *islands* that specify constructs of interest: The module DataFields in Figure 3.4 adds such an island by specifying that a Chunk can also be produced by a Level number followed by a DataName (line 4). DataNames are characters followed by zero or more characters or digits (line 7). Level numbers lie between 01 and 49 (lines 8 and 9)

This very simple grammar allows us to generate a parser that searches for data fields in COBOL code. Although this may not be a spectacular example (something similar

```
module DataFields                                          (1)
imports Water DataParts                                    (2)
context free syntax                                        (3)
  Level DataName        → Chunk        {cons(Data)}        (4)

module DataParts                                           (5)
lexical syntax                                             (6)
  [A-Z][A-Z0-9\-]*      → DataName                         (7)
  [0][1-9]              → Level                            (8)
  [1-4][0-9]            → Level                            (9)
```

Figure 3.4: COBOL data fields.

could be done, for example, using a tool like `grep`), we will show below how we can extend this grammar to do a more complicated analysis. Furthermore, the modularity of SDF allows us to reuse the base grammar developed here for other island grammars.

The grammar in Figure 3.4 is not very discerning. Consider an input program that contains the following line:

```
IF C > 10 AND C < 20
```

The grammar will recognize this line as water containing the text `IF C >`, followed by a data field with level 10 and name `AND`, followed by water containing `C < 20`. Something similar will happen for all other number–name sequences in the code. Obviously, this is not correct, so we need to improve our grammar.

The solution is to restrict the grammar so it will only look for data fields in the data division of the program. This can be done by refining the grammar into the one that is shown in Figure 3.5. The production on line 4 and 5 specifies that non-terminals of type DdChunk are only to be recognized in a context that starts with the text `DATA DIVISION` and ends with the text `PROCEDURE DIVISION`. Line 6 defines these DdChuncks to be our data fields that can be produced by a Level and a DataName (similar as in our original grammar). Furthermore, our grammar needs to be made robust against other data that can occur in this context (i.e., parts of the input that do not match with the data field definition). We do this by also allowing water to be recognized between the markers (line 7).

This example shows some of the advantages that island grammars have over lexical approaches. Most importantly, it is much easier to use structure while specifying patterns to be analyzed. For example, it would be really hard to limit `grep` so it would only match in the data division. In other lexical approaches one would use state manipulation to achieve such results. However, when parts of the analysis logic are hard-coded, adapting the analyzer or combining two analyzers into a single new one becomes a tricky job. In contrast, solutions based on island grammars can easily be combined and are declarative, making them easier to understand.

```
module DataFieldsWithContext                                  (1)
imports Water DataParts                                       (2)
context free syntax                                           (3)
  "DATA DIVISION" DdChunk*                                    (4)
  "PROCEDURE DIVISION"  → Chunk                               (5)
  Level DataName        → DdChunk    {cons(Data)}             (6)
  Water                 → DdChunk                             (7)
```

Figure 3.5: COBOL data fields in context.

3.5 Impact Analysis using Island Grammars

To perform impact analysis, we create an island grammar in which the islands are based on the patterns, anti-patterns and classification described in Section 3.3. When we then parse the artifacts using a parser that is generated from this grammar, the parse trees will contain the analysis results.

Unfortunately, the complete grammar that was used for our case study is too large to show completely in this chapter. It consists of 148 productions which is mainly due to the large amount of patterns and anti-patterns (125 to be precise). Below, we will describe and show the most interesting parts.

When developing the grammar, we start with an "empty" island grammar that "parses" the complete input as water. This is the grammar shown in Figure 3.3.

Next, we extend this grammar with island productions for data fields with generic field identifiers. The resulting grammar can be used to extract all data fields from COBOL sources. However, we are interested in more specific information: we are looking for data fields whose names match with the account number patterns. Therefore, we refine the identifier syntax so it only matches with the account number patterns. The consequence of this refinement is that all data fields whose names do not match are now parsed as water.

The following step is filtering all data fields whose names match with anti-patterns. In SDF, we can use the `reject` attribute to prevent that names that match with an anti-pattern can be parsed as valid data names. The resulting grammar will only parse data fields whose names match with one of the patterns and with none of the anti-patterns.

The grammar parts responsible for recognizing data names by matching patterns and rejecting anti-patterns are shown in Figure 3.6. The actual data names are described in module DataNames: line 4 defines that a DataName contains at least one of the patterns, line 5 defines that a DataName containing one of the anti-patterns is not

```
module DataNames                                                (1)
imports Patterns AntiPatterns                                   (2)
lexical syntax                                                  (3)
  DNPart Pattern DNPart      → DataName                         (4)
  DNPart AntiPattern DNPart → DataName          {reject}        (5)
  [A-Z0-9\-]*                → DNPart                           (6)

module Patterns                                                 (7)
lexical syntax                                                  (8)
  "ACCNO" | "GNR" | ...      → Pattern                          (9)

module AntiPatterns                                             (10)
lexical syntax                                                  (11)
  "DAGNR" | ...              → AntiPattern                      (12)
```

Figure 3.6: Matching patterns in data names.

```
module Pictures
context free syntax
  "999999999"  |  "9(9)" | "99.99.99.999"        → Pict       {cons(Short)}   (1)
  "9999999999" | "9(10)" | "999.99.99.999"       → Pict       {cons(Long)}    (2)
  PictureString                                  → RestPict   {cons(Other)}   (3)
  Pict                                           → RestPict   {reject}        (4)
  Pict | RestPict                                → Picture                    (5)
lexical syntax                                                                (6)
  [0-9XxAa\(\)pZzVvSszBCRD\/\,\$\+\-\*\:]+       → PictureString              (7)
```

Figure 3.7: Recognizing and classifying picture clauses.

valid (i.e, should be rejected), and line 6 defines the possible pre- and postfixes for the patterns.

The patterns are defined in modules Patterns and AntiPatterns. For brevity, we show only a few patterns of the actual list. The anti-pattern DAGNR in line 13 is a Dutch abbreviation for day number which would normally give a false match with GNR.

Classification of the account number variables is done by building up all potential picture clauses from a number of patterns for each class. This is shown in Figure 3.7. We start with a production describing the 9-digit pictures (line 1), followed by the 10-digit pictures in line 2. Furthermore, we want a production for all remaining pictures of variables that match a pattern. We let the parser construct that class by specifying that it consists of an arbitrary PictureString (line 3 and 7) but prevent that the 9-digit and 10-digit pictures are parsed by this production using the reject in line 4. In a way, this reject allows us to "subtract" a set of pictures from the large set described by the production in line 7.

The grammar that combines data names with picture clauses into data descriptions is shown in Figure 3.8. We can use these data descriptions to refine the grammar from Figure 3.5 by replacing the definition of DdChunk in line 6 with a production:

```
module DataDesc                                                (1)
imports DataNames Pictures                                     (2)
context free syntax                                            (3)
  Level DataName "."          → DataDesc     {cons(Rec)}       (4)
  Level DataName Water*                                        (5)
  "PIC" Picture               → DataDesc     {cons(Field)}     (6)
lexical syntax                                                 (7)
  [0][1-9]                    → Level                          (8)
  [1-4][0-9]                  → Level                          (9)
```

Figure 3.8: Recognizing valid data descriptions.

DataDesc → DdChunk. The resulting island grammar can be used to extract account numbers from a COBOL source and classify them using their picture clauses.

3.6 Generation of Impact Analyzers

This section describes implementation details of how we have built the ISCAN impact analyzer. As discussed in the introduction, we expect that there is general interest for the kind of lightweight impact analyzers described in Section 3.3. Therefore, we set out to build tooling that can help the maintainer to create such tools. Our design goal is to minimize the amount of work needed for creation of a new analyzer. We achieve this goal using generative programming [CE00].

We have composed a generative framework for the creation of impact analyzers using island grammars. A maintainer can instantiate the framework using simple specifications detailing the problem at hand. This results in generation of a new impact analyzer that performs an analysis dedicated to the given problem. An overview of the generator framework is shown in Figure 3.9. The gray boxes depict maintainer inputs.

The minimal amount of work that needs to be done to create a new analyzer is very small: it consists of writing the island grammar that specifies affected sites in the artifacts. This grammar plays a central role in the generation of the remaining parts. It is used to generate an island parser and it is part of the inputs needed for the generation of source model extraction and artifact markup.

For the remaining steps in the process, we supply generic defaults. These include:

- A source model extractor that stores information regarding islands recognized by the island parser in a repository.
- A transformation that adds markup to the artifacts, tagging islands recognized by the island parser with their respective types.
- Tools for computing statistics and generating pie charts based on the types of islands available in the grammar.

These generic components can be refined by the maintainer to perform a more specific task (the dashed inputs in Figure 3.9).

3.6.1 Source Model Extraction

The source model extractor is created using MANGROVE/JAVA, a generator for source model extractors based on island grammars that is described in Section 2.3. The extractor processes the results of the island parser using JAVA. The default extractor specification that we provide is a simple JAVA class that stores information regarding all islands that were recognized by the parser in a repository. This class can easily be refined by a user to perform a more specific task, for example, storing only information about particular islands or computing extra information based on a combination of islands.

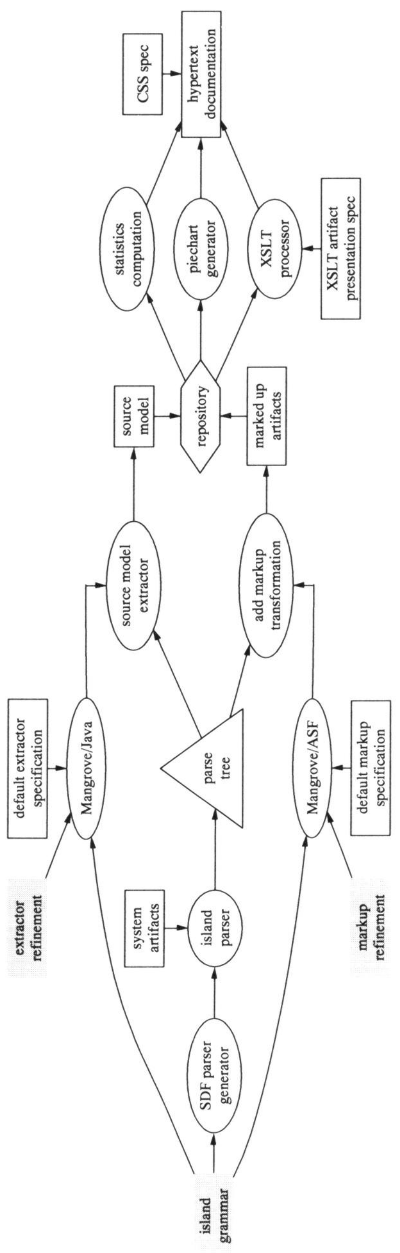


Figure 3.9: Implementing the ISCAN architecture.

3.6.2 Island Markup

The transformation that adds markup is generated using MANGROVE/ASF (also described in Section 2.3). It processes parse results in a functional fashion using the term rewriting language ASF [BHK89]. Specifications written in ASF can be executed using the ASF+SDF Meta-Environment [Kli93, BDH⁺01]. This environment contains support for the generation of term traversal functions [BKV01]. We use these in our default specification to tag all islands that are recognized by the parser with their respective types. This specification can also be refined by the user to perform more specific tasks.

We have chosen to do this transformation using the ASF+SDF Meta-Environment since it allows us to keep the original layout intact while transforming the artifacts [BV00]. Preserving layout is an important feature in a maintenance tool since it helps a maintainer to orientate when visiting a system that he has seen before.

3.6.3 Presentation

We use XML to mark up the artifacts. The marked up artifacts are used for pretty-printing and to generate indexes and tables that cross-reference the various classes and sources. Our current back end generates a series of HTML documents. The transformation of XML to HTML is done using XSL transformations (XSLT). These transformations can be done either on the server side, for example using the XALAN XSLT processor,[2] or on the client side using a modern browser such as NETSCAPE 6 or INTERNET EXPLORER 5. The account numbers in the generated documentation are colored to show their classification: red for 9-digits, green for 10 digits, etc. The actual colors that are used can be changed easily by editing a style-sheet.

3.7 Applications

In this chapter, we have focused on solving a specific case: the impact of expanding 9-digit bank account number into 10-digit numbers. There are many more of such problems to which our technique can be applied. In this section we describe a number of similar problems that we have seen in practice:

Product codes: We have encountered a problem that was very similar to the bank account number analysis in another project that was done by our spin-off, the Software Improvement Group. The problem there was a large software system that used product codes that consisted of 2 digits. The goal of the project was a transformation of this system that expanded the product codes to consist of 3 digits (surprisingly with a maximum of 299 instead of 999).

Trading natural gas: Liberalization of the gas-market in Western Europe makes it possible for consumers to pick the gas supplier they like. To enable this, the various gas networks have been interconnected, making it easier for producers to sell

[2] http://xml.apache.org/xalan-j/

their gas in more remote markets. Before liberalization, gas trading contracts were based on capacities in m^3 per hour. However, the caloric value of natural gas differs between gas reserves, so the actual energy value that is purchased/sold with 1 m^3 also differs per gas reserve. Since this is not a competitive price model when trading between various gas reserves, gas trading companies want to convert from capacities in m^3 per hour to capacities in kW per hour. Obviously, this conversion implies mass changes in their trading and accounting software.

Selling natural gas: Another change that gas trading companies want to make in their software has to do with consumer accounting. Historically, gas supply days run from 6am one day to 6am the next day. This poses several problems for service integration, for example, when one would like to combine gas and electricity billing. Therefore these supply days need to be changed into the standard 0am-12pm schedule.

Date format conversion: Changing the date representation in a system from the U.S. date format (MM/DD/YYYY), or the European date format (DD/MM/YYYY), into the international ISO date format (YYYY-MM-DD).

3.8 Related Work

3.8.1 Impact Analysis

Bohner and Arnold give a tutorial style overview of research topics in the area of software change impact analysis [BA96]. The articles in this book focus on the traditional full-blown impact analysis that one would use to process a change request, and not on the kind of lightweight impact analysis that we focus on. The techniques described in the book will be too expensive to be practical for the estimation and planning phase of a software change project. However, they will be needed in the next phases of the project.

Most of the traditional impact analysis approaches are based on program slicing [GL91] and program dependence graph analysis [LM93]. Han describes an impact analysis approach that is based on direct analysis of the system artifacts [Han97]. Similar to our approach, it analyzes the parse trees to determine impact and change propagation. Han argues that such a direct approach is better suited for providing impact analysis and change propagation as integral parts of a software engineering environment.

Fyson and Boldyreff describe the use of program understanding to support impact analysis [FB98]. They use information derived by a program understanding system to populate a so called ripple propagation graph. By tracing the edges of this graph, one can identify all systems that are affected by a change.

3.8.2 Lexical Approaches

Several tools are available to perform lexical analysis. The most well-known tools are probably `grep` and PERL that allows one to search text for strings using regular expressions.

Murphy and Notkin describe the Lexical Source Model Extractor (LSME) [MN96]. Their approach uses a set of hierarchically related regular expressions to describe language constructs that have to be mapped to the source model. By using hierarchical patterns they avoid some of the pitfalls of plain lexical patterns but maintain the flexibility and robustness of that approach.

The MultiLex system of Cox and Clarke [CC00] uses a similar hierarchical approach. The main difference with LSME is that it focuses at extracting information at the abstract syntax tree level whereas LSME extracts higher level source models.

These tools offer no immediate support for impact analysis. They are directed at extracting the facts from system artifacts and not at querying, combining and presenting those facts to the maintainer to answer a question or to perform impact estimation.

3.8.3 Rapid System Understanding

Van Deursen and Kuipers describe techniques for rapid system understanding that are based on lexical analysis [DK98]. They describe an open architecture for system understanding that can be easily adapted to perform a problem-directed analysis. This makes it easy to use their technique for performing impact analysis for estimation and planning.

3.8.4 Island Grammars

Island grammars are a technique for syntactical analysis that allows us to mix the behavior of parsing with that of lexical approaches by analyzing the interesting parts of a grammar and brushing aside the non-interesting parts. Thereby, island grammars combine the accuracy of syntactical analysis with the speed, flexibility and robustness of lexical analysis. In the previous chapter, we describe the definition of island grammars using the syntax definition formalism Sdf and present Mangrove, a generator for source model extractors based on island grammars that supports refinements in various programming languages and show how it can be used. For a more detailed discussion of related work regarding island grammars and source model extraction using syntactical and lexical analysis, we refer to Chapter 2.

3.9 Concluding Remarks

3.9.1 Evaluation

By using an island grammar to perform impact analysis, we limit ourselves to types of analysis that can actually be described using grammars. These are the kind of analyses that are based on determining the presence, absence, and classification of features in an artifact. They exclude, for example, analyses that are based on data flow information or dependency tracking. Note that when a more detailed analysis is needed, an approach based on island grammars can be improved to a certain extent by doing more involved computations in the tools that extract source models and markup artifacts.

We argue that the type of analysis that can be described using island grammars is sufficient for our goal: lightweight impact analysis for estimation and planning. This is supported by the fact that others revert to lexical analysis techniques to achieve this goal (e.g., [DK98]). The use of island grammars has several advantages over lexical approaches. Most importantly, it is much easier to use structure in the specification of the patterns. Furthermore, solutions based on island grammars can easily be combined and are declarative, making them easier to understand.

We identify two potential sources of problems with island grammars: (i) *false positives* that occur when the grammar allows constructs to be recognized in places where they should not have been recognized. (ii) *false negatives* that occur when the grammar is too restrictive and does not allows constructs to be recognized in places where they should be recognized. These errors can be solved by strengthening the grammar, we refer to Chapter 2 for a discussion of possible approaches.

The expressive power of an island grammar is limited by the chosen syntax definition formalism and more important by the chosen parsing technique. We express island grammars in SDF, a syntax definition formalism that is supported by *generalized LR* parsing techniques. Consequently, we inherit their expressive power, which allows us to express the complete class of context free languages.[3]

To get an indication of the speed and scalability of our approach, we have tested the generated impact analyzer on representative parts of the earlier described software portfolio (the complete portfolio could not be used due to disclosure restrictions). We have performed two tests: (a) one on a single system, and (b) one on a collection of four systems. Figure 3.10 gives an overview of the test results. The analysis time is the user CPU time as reported by the GNU `time` command and the maximum memory usage was observed using `top`. The tests were performed on a computer with an AMD Athlon processor (1.2 Ghz) and 512 Mb main memory running linux 2.4.9-12.

Since the tests were done on representative systems (with similar average, largest and smallest program size), we think that these results can be extrapolated. Thus, impact analysis of the complete portfolio of 50,000,000 LOC will take approximately

[3] And some non-context free languages because the *reject* attribute allows us to compute the difference or intersection of two languages. For more details, see the discussion in [Vis97, p. 52–56].

test	no. of programs	total size (LOC)	analysis time (s)	speed (LOC/s)	memory usage
(a)	206	233,252	403	579	49 Mb
(b)	818	901,899	1,549	582	54 Mb

Figure 3.10: Benchmark results

one day (24 hours), which is more than acceptable for estimation purposes on a project of this size.

3.9.2 Future Work

We are interested in investigating how we can extend our approach with dependency tracking to perform a more detailed analysis. We want to do this using the *type inferencing* technique described in Chapters 4 and 5. The basic idea is as follows: we will use the data fields found by the lightweight impact analysis as *seeds*. Initially, these seeds get a unique type. These types will be propagated through the statements in the program and track all related (type equivalent) fields encountered during this propagation.

We need to make the following additions to our framework: First, the island grammar is refined so the generated parser will recognize assignments and expressions as islands. Then, we extend the source model extractor, so it emits primitive type relations for the seeds, type equivalencies for expressions and subtyping for assignments that are encountered. These relations can be used to find all fields that are type equivalent with the seeds following the algorithm described in Chapter 5 (Figure 5.5, page 90). We can classify these sets using the classification that was found for the seeds since all type equivalent fields should be in the same category. Finally, we can use this new information to generate a more detailed overview of the impact on the code.

3.9.3 Contributions

Lightweight impact analysis is a prerequisite for estimating and planning large scale software maintenance projects. This chapter shows that island grammars can be used to generate such lightweight impact analyzers.

We have given a detailed description of the process of translating an impact analysis problem into an island grammar. We have discussed the advantages that this approach has over other techniques for impact analysis. We have presented a generative framework that allows a maintainer to create lightweight and problem-directed impact analyzers. We have demonstrated 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.

Acknowledgments

The author would like to thank Arie van Deursen, Paul Klint and Joost Visser for commenting on earlier versions of this chapter.

PART II

Type Inference

Chapter 4

Type Inference for COBOL

Types are a good starting point for various software reengineering tasks. Unfortunately, programs requiring reengineering most desperately are written in languages without an adequate type system (such as COBOL). To solve this problem, we propose a method of automated type inference for these languages. The main ingredients are that if variables are compared using some relational operator their types must be the same; likewise if an expression is assigned to a variable, the type of the expression must be a subtype of that of the variable. We present the formal type system and inference rules for this approach, show their effect on various real life COBOL fragments, describe the implementation of our ideas in a prototype type inference tool for COBOL, and discuss a number of applications. The work presented in this chapter was published earlier as [DM98].

4.1 Introduction

The many different variables occurring in a typical program, can generally be grouped into *types*. A type can play a number of roles:

- It is an indication of the set of values that is allowed for a variable;
- A type groups variables that represent the same kind of entities;
- A type helps to hide the actual representation (array versus record, length of array, ...) used;
- Types for input and output parameters of a procedure provide a "signature" of the expected use of that procedure.

Traditionally, types are associated with strongly-typed languages, in which explicit variable and type declarations help to detect programming errors at compile time instead of run time.

In this chapter we will be concerned with a rather different use of types. In our opinion, types are a good starting point for various software reengineering activities. We argue that the use of types as described in this chapter is in fact the underlying theory of the approach followed by a number of existing reverse engineering tools. For example, types can be used for migrating from a procedural to an object-oriented language, isolating reusable components from legacy sources, searching for potential year 2000 infections, or for searching code that will be affected by the introduction of the Euro: the single European currency.

Unfortunately, systems for which such reengineering activities are most necessary, are generally written in languages with a rather limited type system. This makes reengineering for such languages difficult. To solve this problem, we propose methods to *infer* a set of types from programs written in such languages automatically. These automatically inferred types can then be the starting point for objectification, year 2000 remediation, etc.

The language we deal with in this chapter is COBOL. We show how to infer a set of types automatically from (a system of) COBOL programs. We present several varieties of our type system, taking sub-typing, byte representations and inter-program types into account. We describe how we made a prototype tool that performs type inference on COBOL code.

We have evaluated our approach using a case-study where we apply the ideas described above to MORTGAGE: a 100,000 LOC COBOL system from the banking area. The examples in this chapter are taken from that system.

We conclude by describing a number of important applications of our technique in the area of software reengineering.

4.2 Approach and Motivation

At first sight, COBOL may *appear* to be a typed language. Every variable occurring in the statements of the procedure division, must be declared in the data division first. A typical declaration is shown on Figure 4.1. Here, three variables are declared: `TAB100-FILLED`, which is an integer (picture "9") comprising three bytes initialised with value zero; `TAB100-POS`, which is a single character byte (picture "X") occurring 40 times, i.e., an array of length 40; and `TAB100` which is a record defined at level `01`, having the two variables with higher level numbers, namely `05`, as fields.

Unfortunately, the variable declarations in the data division suffer from a number of problems, making them unsuitable to fulfil the roles of types as listed in the begin-

```
       01  TAB100.
           05  TAB100-POS      PIC   X(01)  OCCURS 40.
           05  TAB100-FILLED   PIC  S9(03)  VALUE 0.
```

Figure 4.1: Fragment of COBOL data division.

ning of this chapter. First of all, since it is not possible to separate type definitions from variable declarations, when two variables for the same record structure are needed, the full record construction needs to be repeated. This violates the principle that the type hides the actual representation chosen.

Besides that, the absence of type definitions makes it difficult to group variables that represent the same kind of entities. Although it might well be possible that such variables have the same byte representation. Unfortunately, the converse does not hold: One cannot conclude that whenever two variables share the same byte representation, they must represent the same kind of entity.

In addition to these important problems pertaining to type definitions, Cobol only has limited means to accurately indicate the allowed set of values for a variable (i.e., there are no ranges or enumeration types). Moreover, in Cobol, sections or paragraphs that are used as procedures are typeless, and have no explicit parameter declarations.

In our approach, we use types to group variables that represent the same kind of entities. We start with the situation that every variable is of a unique primitive type. We then generate equivalences between these types based on their usage: if variables are compared using some relational operator, we infer that they must belong to the same type; and if an expression is assigned to a variable, the type of the variable must be that of the expression. We also propose a more refined scheme, in which a subtype relation between the types of the expression and the variable is inferred for assignments.

Furthermore, we use a similar approach to infer a minimal set of literal values that should be included in certain types. This information can be used to replace hard wired literal constants in a program with symbolic constants (i.e., replace them by variables that have the same initial value and are not changed in the program). Type information is important for such renovations since the constants for each type might need to be changed independently as a result of maintenance of the program.

Finally, from the minimal set of values of a given type and the usage of variables of that type, we infer whether such a type is an enumeration type: if variables of such a type only get assigned values from this set and there are no computations that might change that value then the type is an enumeration type.

4.3 Notation

In this chapter, we will consider the following primitive types:

Definition 4.3.1 *The set T of* primitive types *is defined by the following productions:*

$$\begin{array}{rcll}
N & ::= & \textit{Natural numbers} & \\
I & ::= & \textit{Set of } \text{identifiers} & \\
B & ::= & \textit{Set of } \text{byte sorts} & \\
P & ::= & B^{+} & \textit{(Pictures of bytes)} \\
T & ::= & \mathbf{elem}(I, P) & \textit{(Elementary variable)} \\
 & \mid & \mathbf{rec}(I, T^{+}) & \textit{(Record type)} \\
 & \mid & \mathbf{array}(I, T, N) & \textit{(Array type)}
\end{array}$$

In other words, we distinguish type constructors for elementary data types, for records, and for arrays (with a given length). All types have a name as their first component. The precise choice of the set of byte sorts B can be chosen at will: for our purposes, it consists of the COBOL byte markers such as X (character byte), 9 (decimal digit), etc., as occurring in COBOL picture clauses.

We will use T_A to refer to the primitive type that can be derived for a given variable A from the data division of a program in which A is used.

Below we define the language constructs that are used to describe the type inference rules in the rest of this chapter.

Definition 4.3.2 *The set S of* syntactic constructs *is defined by the following productions:*

$$\begin{array}{rcll} L & ::= & \textit{Set of literals} & \\ V & ::= & I & (\textit{Identifier}) \\ & \mid & I(E) & (\textit{Array access}) \\ E & ::= & L & (\textit{Literal value}) \\ & \mid & V & (\textit{Variable}) \\ & \mid & E_1 \textit{ a-op } E_2 & (\textit{Arithmetic operator}) \\ C & ::= & E & (\textit{Expression}) \\ & \mid & E_1 \textit{ rel-op } E_2 & (\textit{Relational operator}) \\ & \mid & V := E & (\textit{Assignment}) \\ S & ::= & C+ & (\textit{Syntax}) \end{array}$$

The set L corresponds to literals such as numbers and strings, V are variable and array accesses, and E are arithmetic expressions. The set C consists of the set of constructs that are needed for our purposes: arithmetic expressions, relational expressions, and assignments. It contains only those language constructs that affect the type inference algorithm. The top or start set S is just a collection of constructs from C.

Following [Car97], we will use so-called *judgements* to express relations between syntactic constructs, and types. Let Γ be a *type environment*, i.e., a mapping from identifiers to types. We will distinguish the following five judgements:

- $\Gamma \vdash \diamond$

  Γ is a well-formed type environment.

- $\Gamma \vdash E : T$

  Expression E is of type T.

- $\Gamma \vdash S : T_1 \equiv T_2$

  An equivalence relation indicating that given construct S, types T_1 and T_2 are the same.

- $\Gamma \vdash S : T_1 \preccurlyeq T_2$

  A partial order indicating that given construct S, type T_1 is a subtype of type T_2.

- $\Gamma \vdash S : L \in T$

  Given construct S, literal L is an element of type T.

The sections to come will include a number of *inference rules* indicating for what particular language constructs these judgements hold.

4.4 Inference Rules

In this section we describe a method to find an *equivalence* relation between the primitive types within a single module (COBOL program). Later, we will extend this method to system level types and refine the results using subtypes.

4.4.1 The Data Division

Every variable declared in one of the various sections of the data division of a COBOL program corresponds to a type from the set T of primitive types in a straightforward manner. For simple variables, the PIC clause is used to obtain the sequence of byte sorts. OCCUR clauses result in arrays, and record definitions yield (nested) record types. To avoid name clashes between fields with the same name coming from different records, variables should be qualified using the full nested record structure. This is a trivial translation that can be done in a preprocessing phase on the incoming COBOL code. As an example, Figure 4.2 shows the type environment resulting from the COBOL variable declarations shown in Section 4.2. Observe that every COBOL variable obtains a unique type. In order to focus the presentation on the most relevant issues, we postpone the treatment of REDEFINEs until Section 4.7.

4.4.2 Types for Expressions

An arithmetic expression is constructed from variables, constants, and arithmetic operators such as $+, -, *, \ldots$. We derive the type of such an expression by distinguishing the following cases:

1. *Variable access*: If e is a variable, array access, or record field access, its type is the one obtained from analysing the data division.

2. *Arithmetic operators*: Let e be an arithmetic expression of the form e_1 *a-op* e_2. We then infer several types for this combined expression: every type of e_1 and

```
TAB100          ↦   record(TAB100,
                       array(TAB100-POS, elem(TAB100-POS[],X),40)
                       elem(TAB100-FILLED,S9999)),
TAB100-POS      ↦   array(TAB100-POS, elem(TAB100-POS[],X),40),
TAB100-POS[]    ↦   elem(TAB100-POS[],X),
TAB100-FILLED   ↦   elem(TAB100-FILLED,S9999)
```

Figure 4.2: Type environment derived from data division fragment.

every type of e_2 is also a type of e.

The rules formalising these cases are shown in Figure 4.3. As an example, an expression consisting of just the variable `A` will have one type, T_A, the primitive type derived for `A` from the data division. An expression `A + B` will have two different types: it is both of type T_A as well as of type T_B.

One might think that the type of an expression can be any of the types of the identifiers occurring in that expression. In general, however, this is not the case: an expression can contain an array access, for example `A(I+1) + B(J+1)`, but the type of variables occurring in the access (namely `I` and `J`) are not part of the type of the full expression.

Observe that we take advantage of the fact that in COBOL all arithmetic operators take arguments that must have the same type, namely a numeric type. If COBOL would contain other operators, for example involving both strings and numeric arguments, these operands should not receive the same type. Support for such operators could easily be added to our system by refining the inference rules for operators.

Furthermore, there are no rules for literal expressions (constants): At this stage we are only interested in finding out type information about *variables*.

4.4.3 The Procedure Division

Now that we know how to derive types for variables and arithmetic expressions, we can define how to infer relations between the types of the syntactic constructs from S. We distinguish the following cases:

1. *Arithmetic expression*: If $s \in S$ is an arithmetic expression, as we have seen in the previous section, the types of its operands are defined to be equivalent.
2. *Relational operator*: If $s \in S$ is a relational operator, such as $>, <, =, \ldots$, the types of the operands are defined to be equivalent.
3. *Assignment*: If $s \in S$ is an assignment of the form $v := e$ (recall that this corresponds to COBOL statements such as `MOVE`, `COMPUTE`, `MULTIPLY`, ...), we define that the types of e and v are equivalent.

$$\frac{(\Gamma_1, i \mapsto t, \Gamma_2) \vdash \diamond}{(\Gamma_1, i \mapsto t, \Gamma_2) \vdash i : t} \quad \text{Variable Types}$$

$$\frac{\Gamma \vdash e_1 : t_1}{\Gamma \vdash e_1\ a\text{-}op\ e_2 : t_1} \quad \text{A-Op Left}$$

$$\frac{\Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1\ a\text{-}op\ e_2 : t_2} \quad \text{A-Op Right}$$

Figure 4.3: Types for variable access and arithmetic expressions.

4. *Array access*: If S contains two constructs that both have array accesses to the same variable, say $v(e_1)$ and $v(e_2)$, then the types of the index expressions are defined to be equivalent. Note that this includes any pair of accesses to the same array v in a program.

The rules formalising these cases are shown in Figure 4.4. Note that the Array Index rule uses a *context* variable of the form $S[\ldots]$, which represents the source tree S with a subtree left open. We refer to [DHK96] for more details.

As an example, let us infer the type relations for the expression `A + B < D`. The subexpression `A + B` leads, via rule A-Exp, to an equivalence between T_A and T_B. As was shown in the previous section, this subexpression has both type T_A and type T_B. These two types are used when inferring the relations between the types of the complete expression: Following rule Rel-Op, any type of `A + B` is equivalent to the type of `D`. Hence we infer two more equivalences, namely between T_A and T_D as well as between T_B and T_D. Thus, the expression `A + B < D` results in three equivalences: $T_A \equiv T_B$, $T_A \equiv T_D$, and $T_B \equiv T_D$.

4.4.4 Example

For practical purposes, the most important result of the type inference procedure are the equivalence classes for types. As an example, consider Figure 4.5, which shows a Cobol fragment manipulating strings. At first sight, the exact relationship between the seven declared variables will be unclear. Applying our type equivalence procedure to this fragment, will infer that `N100`, `TAB100-MAX`, and `TAB100-FILLED` all belong to the same type, due to the statements

```
MOVE TAB100-MAX TO N100.
```

and

```
MOVE N100 TO TAB100-FILLED
```

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \; \textit{a-op} \; e_2 : t_1 \equiv t_2} \quad \text{A-Exp}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \; \textit{rel-op} \; e_2 : t_1 \equiv t_2} \quad \text{Rel-Op}$$

$$\frac{\Gamma \vdash v : t_1 \quad \Gamma \vdash e : t_2}{\Gamma \vdash v := e : t_1 \equiv t_2} \quad \text{Assignment}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash S[v(e_1)][v(e_2)] : t_1 \equiv t_2} \quad \text{Array Index}$$

Figure 4.4: Rules to infer equivalences between types.

The equivalence class of these three types corresponds to the index type of the TAB100-POS array.

The information that these three variables belong to the same type, can be graphically displayed in an editor (for example by giving them the same colour) which would help the programmer to understand relationships between variables when browsing the program. Moreover, this information can be used when migrating a Cobol application to a typed language. The typical Pascal type for this equivalence class would be a range from 1 to 40 used as array index type.

Other applications of type information in reverse engineering are described in Section 4.11.

4.5 System-Level Types

The previous section describes a way of finding sets of equivalent primitive types within a single module (Cobol program). Given the type relations per program, we can infer further type equivalences based on inter-program relations in the following manner:

- Make all identifiers unique per program, by qualifying them with the program name.

```
01  N000.
    05 N100              PIC  S9(03)  COMP-3.
    ...
01  TAB000.
    05  TAB100-NAME-PART
        10  TAB100-POS    PIC   X(01)  OCCURS 40.
    05  TAB100-MAX        PIC  S9(03)  COMP-3 VALUE 40.
    05  TAB100-FILLED     PIC  S9(03)  COMP-3 VALUE ZERO.
...
R300-COMPOSE-NAME SECTION.
   MOVE TAB100-MAX TO N100.
   MOVE ZERO        TO TAB100-FILLED.

   PERFORM UNTIL N100 EQUAL ZERO
      IF TAB100-POS (N100) EQUAL SPACE
         SUBTRACT 1 FROM N100
      ELSE
         MOVE N100 TO TAB100-FILLED
         MOVE ZERO TO N100
      END-IF
   END-PERFORM.
```

Figure 4.5: Cobol fragment for manipulating strings.

```
LINKAGE SECTION.
01  L001-FUNCTION         PIC  S9(05)  COMP-3.
01  L001-RAR001-FIXED     PIC   X(274).
01  L001-FORMATTED-NAME   PIC   X(46).
01  L001-ENTITY.
    05  L001-ENTITY-NR    PIC  S9(11)  COMP-3.
    05  L001-ENTITY-TYPE  PIC   X(01).
01  L001-STATUS           PIC  S9(05)  COMP-3.
```

Figure 4.6: Linkage section (formal parameters) of program RA36.

Variables declared in copybooks that are included in the data division should be qualified using the copybook's name — in this way variables declared in copybooks included in multiple programs will have the same type.

- In a program call, the actual parameters (the COBOL USING clause) are assigned to the formal parameters (the COBOL linkage section), resulting in an inferred equivalence between their types.
- Read and write operations of different variables to the same database result in an inferred equivalence between the variable's types.

A fairly typical call is shown in Figures 4.6 and 4.7. Regarding type equivalence, a first observation is that in the call statements, RAR001-FIXED is an array of 274 bytes. In other statements (not shown), it is assigned to variables declared as a record also consist of 274 bytes. This is typical COBOL programming style, and done to keep the interface of the call statement simple. Our type inference approach will find equivalences

```
01  L000.
    05  L100-RA36.
        10  L100-FUNCTION          PIC  S9(05)  COMP-3.
        10  L100-RAR001-FIXED      PIC  X(274).
        10  L100-FORMATTED-NAME    PIC  X(046).
        10  L100-ENTITY.
            15  L100-ENTITY-NR     PIC  S9(11)  COMP-3.
            15  L100-ENTITY-TYPE   PIC   X(01).
        10 L100-STATUS             PIC  S9(05)  COMP-3.
    ...
CALL  'RA36'  USING
   L100-FUNCTION   L100-RAR001-FIXED
   L100-FORMATTED-NAME    L100-ENTITY L100-STATUS.
```

Figure 4.7: Call to program RA36, together with actual parameters.

between these byte arrays and full records. This allows us to retrieve the complete (complex) interface that programs actually use for their inter-program communication.

A second observation is that the `L100-ENTITY` parameter is in fact a record. The parameter passing is treated as an assignment from `L100-ENTITY` to `L001-ENTITY`. This, in turn is used to infer a type equivalence between these two records. When looking at the example, however, we immediately see that the fields of these records, namely `ENTITY-NR` and `ENTITY-TYPE`, should also be of the same type. This, however, is not inferred by the rules given so far.

Clearly, this is a situation which can occur not only at the inter-program level, but also within programs. What we need is a rule which says that if two structure types are inferred to be equivalent, and if these types have the same structure (without looking at the names), we can infer an additional equivalence between the sub-level types.

To formalise this, we first need the notion of *representation* (i, p, t, n are variables ranging over I, P, T, N, respectively):

Definition 4.5.1 *We define rep* : $T \rightarrow P$, *which gives the byte representation of a type inductively by*

$$\begin{array}{lcl} rep(\boldsymbol{elem}(i,p)) & = & p \\ rep(\boldsymbol{rec}(i,t_1\ldots t_n)) & = & rep(t_1)\ldots rep(t_n) \\ rep(\boldsymbol{array}(i,t,n)) & = & rep(t)^n \end{array}$$

The rules in Figure 4.8 then deal with inferring equivalence for subconstructs. The Fields rule states that if we know that two records are inferred to be equivalent, and if we know that they have exactly the same number of fields, and every two fields have the same representation, then we can infer that the fields must be equivalent as well.

The Arrays rule states that if we know that two arrays are inferred to be equivalent, and if we know that their elements have the same representation, then we can infer that these elements must be equivalent as well.

$$\frac{\begin{array}{c}(j = 1\ldots n)\ \ (\forall_{k:1\ldots n} : rep(f_k) = rep(f'_k)) \\ \Gamma \vdash S : \mathbf{rec}(i, f_1, \ldots, f_n) \equiv \mathbf{rec}(i', f'_1, \ldots, f'_n)\end{array}}{\Gamma \vdash S : f_j \equiv f'_j}\ \text{Fields}$$

$$\frac{\begin{array}{c}(rep(t) = rep(t')) \\ \Gamma \vdash S : \mathbf{array}(i, t, n) \equiv \mathbf{array}(i', t', n)\end{array}}{\Gamma \vdash S : t \equiv t'}\ \text{Arrays}$$

Figure 4.8: Rules for substructure completion.

4.6 Assessment of Type Equivalence

The rules provided so far describe how an equivalence relation between primitive types can be derived from a COBOL program. These rules are intuitive, and in general they provide meaningful equivalences. There are, however, a number of problematic situations for which inferring type equivalences is not satisfactory.

First of all, it may be the case that one variable is being used for different purposes in different slices of the program. For example, a variable `TMP` may be assigned the 8-digit variable `PHONE-NR` in one slice, and an 8-digit `DATE` in another. The rules provided so far will infer equivalences for both assignments. By transitivity of equivalence, we then get that `PHONE-NR` and `DATE` are of the same type.

A similar situation can occur in a procedure call. In COBOL, this can happen in a program `CALL`, where the variables in the `USING` clause are the actual parameters, and those in the `LINKAGE SECTION` the formal ones. Alternatively, a `PERFORM` statement can be used, in which case global variables can be used as formal parameters (for an example, see the next section). With the rules given so far, all actual and formal parameters of a procedure will obtain the same type. This may lead to undesirable situations, if the procedure, for example, deals with strings in general, and is given actual parameters of different sorts such as `STREET` or `CITY`.

Another situation that does occur in practice is that a single variable, for example `ZEROES`, is assigned to many different variables during the initialisation phase. Alternatively, one variable, for example `PRINT-LINE`, can receive values from many different variables occurring in a sequence of assignments involving output operations. Again, this will give all these variables the same type.

In all these situations, the inference rules lead to too many equivalences, to which we will refer as *type pollution*. In the next section, we discuss how *subtyping* can be used to address this problem.

4.7 Subtypes

A type is an indication for a set of permitted values. If the set of permitted values for type T_1 is a subset of the values of type T_2, type T_1 is said to be a *subtype* of T_2, written $T_1 \preccurlyeq T_2$. Subtyping makes a type system more flexible, since an element of a type can be considered also as an element of any of its supertypes, thus allowing an element to be used flexibly in many different contexts [Car97, Section 6].

The rule for reasoning about type assertions in the presence of subtyping is shown in Figure 4.9. In addition to that, we need rules to explicitly infer a subtype relationship

$$\frac{\Gamma \vdash e : t_1 \quad \Gamma \vdash S : t_1 \preccurlyeq t_2}{\Gamma \vdash e : t_2} \quad \text{Subsumption}$$

Figure 4.9: The has-type relation in the presence of subtyping.

$$\frac{\Gamma \vdash v : t_1 \quad \Gamma \vdash e : t_2}{\Gamma \vdash v := e : t_2 \preccurlyeq t_1} \quad \text{Sub-Assignment}$$

Figure 4.10: Subtype inference rule for assignments.

between two types. Assignments are the natural place for this: If v is assigned an expression e, the type of v should at least contain the values of e, i.e., the type of e is a subtype of the type of v. The rule formalising this is shown in Figure 4.10. With subtyping this rule should be used instead of the "Assign" rule from Figure 4.4, which infers a straight type equivalence.

Inferring subtypes has some important practical benefits. Consider, for example, the fragment of Figure 4.11, which invokes the procedure `R300-COMPOSE-NAME` two times. Since Cobol procedures cannot have parameters, the variable `TAB100-NAME-PART` is used to simulate an input parameter. In the first `PERFORM` statement, it is given the value of `RAR001-INITIALS`, in the second the value of `RAR001-NAME`.

Looking at the names and declarations, one can clearly see that the type of `RAR001-NAME`, a string of length 27 representing a person's last name, and the type of `RAR001-INITIALS`, a string of length 5 representing a person's initials, should be different. However, when inferring type equivalences for assignments, they would become equal, by transitivity via variable `TAB100-NAME-PART`. With subtyping, we do not infer such an equivalence, but infer that they should both have a common supertype, namely the type of `TAB100-NAME-PART` (which has length 40). As described above, similar situations can occur with variables that are used for collecting lines to be printed, temporary variables, etc.

```
01  RAR001-RECORD
    03  RAR001-VAST
        05  RAR001-NAME       PIC X(27).
        05  RAR001-INITIALS   PIC X(05).
      ...
R210-INITIALS SECTION.
   MOVE RAR001-INITIALS TO TAB100-NAME-PART
   PERFORM R300-COMPOSE-NAME
   EXIT.

R230-NAME SECTION.
   MOVE RAR001-NAME TO TAB100-NAME-PART
   PERFORM R300-COMPOSE-NAME
   EXIT.
```

Figure 4.11: Two calls to a procedure with a simulated parameter.

Using subtyping, REDEFINEs can be handled by a simple extension of our type language. In COBOL, REDEFINE clauses are used to define data structures that are known as variant records in Pascal (or unions in C); these can be dealt with by adding a *union type* constructor to the set of primitive types T. During analysis of the data division, the type generated for a number of redefined variables is the union type constructed from the types of the individual variables. Furthermore, a rule is added which infers a subtype relation between the components of a union type and the complete union type. The remaining type inference rules stay the same. For more information on union types, we refer to [Car97].

4.8 Literal Analysis

A natural extension of our type inference algorithm involves the analysis of literals that occur in a COBOL program.

The basic idea is that whenever a variable v is assigned a literal value l, or compared with l, then the type of v should at least contain the literal l. Moreover, whenever we infer that two types must be equivalent, elements contained in one should be contained in the other. Figure 4.12 formalises these ideas.

An example of the use of this literal analysis can be shown using the following piece of code:

```
EVALUATE RAR001-NATURE
   WHEN 001 GO TO R180-100
   WHEN 002 GO TO R180-100
   WHEN 003 GO TO R180-100
   WHEN 013 GO TO R180-100
   WHEN OTHER GO TO R180-999
END-EVALUATE.
```

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash e \ \textit{rel-op}\ l : l \in t} \quad \text{Right literal}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash l \ \textit{rel-op}\ e : l \in t} \quad \text{Left literal}$$

$$\frac{\Gamma \vdash v : t}{\Gamma \vdash v := l : l \in t} \quad \text{Literal assignment}$$

$$\frac{\Gamma \vdash S : l \in t_1 \quad \Gamma \vdash S : t_1 \equiv t_2}{\Gamma \vdash S : l \in t_2} \quad \text{Equivalent types}$$

$$\frac{\Gamma \vdash S : l \in t_1 \quad \Gamma \vdash S : t_1 \preccurlyeq t_2}{\Gamma \vdash S : l \in t_2} \quad \text{Subtypes}$$

Figure 4.12: Rules for inferring minimal literal containment in types.

In this example, the type NATURE is a number indicating the kind of entity described in some large record. Depending on this kind, different actions are taken. In our case-study of the MORTGAGE system, our technique was able to find all constants that are used for all variables of type NATURE.

Consider the following piece of code (also taken from the MORTGAGE system):

```
IF RAR001-NATURE EQUAL 8
   IF RAR008-NUMBER  EQUAL 1234 AND
      RAR008-ZIPCODE EQUAL '5678AB'
   ...
```

In this example, a selection is made based on a specific address that is included in the code.[1] Our analysis will help to identify such "special values" for a particular type, which provides insight in the nature and actual usage of that type.

The literal type information can also be used to improve the replacement of hard wired literal constants in a program with *symbolic constants*. The algorithm is simple: replace the constants by fresh variables that are initialised to the given literal value and are not changed in the program. For example, the tool set of Sneed [Sne98a] has an option called *reassign* for such constant replacements. His approach is to introduce only one symbolic constant which is substituted for all occurrences of the literal constant (e.g. all occurrences of the literal '18' are replaced by CONST-18 and a new data item '01 CONST-18 PIC 99 VALUE 18. is added to the data division).

This approach has the disadvantage that the value of such constants can never be changed during the remaining life time of the reverse engineered program because the literal values that were replaced could have been from different types. For example: consider a program with two literal values '18', one is used to check the number of passengers on a boat, the other is used to check their age. Either of these values might need to be modified during maintenance and by replacing them both by the same symbolic constant CONST-18 such changes can not be made.

The types we infer for literals allow a much more refined renovation: they can be used to replace all occurrences of a literal constant *of a given type* with a symbolic constant for that type. As a result, the constants can be modified independently of each other.

Note that generating names for these constants is no problem, they can either be derived from the name of the type or a fresh prefix can be generated for each new type, similar to the CONST-18 example above.

The results of the literal type inference described above provide an indication of the minimal set of values that should be included in a given type equivalence class. From this set of values of a given type and the usage of variables of that type, we infer whether such a type is an *enumeration type*, i.e., if variables of such a type only get assigned values from this set and there are no computations that might change that value then the type is an enumeration type.

[1] The actual address has been changed to protect the innocent.

4.9 Implementation

We have implemented our ideas in a tool performing type inference on COBOL code. The tool reads COBOL source code and its outputs are the types, typed literal elements, and enumeration types that occur in that code. The architecture of the tool is shown in Figure 4.13. The boxes represent data, the ellipses represent processes and the arrows depict the flow of data through the system. The solid objects in the figure describe the basic type inference tool. The dashed and dotted objects refer to the extension of our system with literal type detection (dashed) and enumeration type detection (dotted) described in Section 4.8.

We start with the step *extract primitive types* which finds a set P of primitive types given the data division of the source code. This set is stored in a type environment for the variables of the data division.

We then perform the *derive type relations* step, which combines the primitive types and the usage of variables in the procedure division. The result is a set of relations, which can either be equivalences ($T_1 \equiv T_2$) or partial orderings ($T_1 \preccurlyeq T_2$) for subtyping. For example, the COBOL statement `MOVE A TO B` results in the relation $T_A \preccurlyeq T_B$.

The *type resolution* step infers the types by computing $P/\equiv$: the partition of the set of primitive types that is induced by the derived equivalence relation. Thus the inferred types are the equivalence classes of primitive types modulo $\equiv$. The derived subtyping order $\preccurlyeq$ on primitive types can be used to compute a subtyping order on the inferred types: if $T_1 \preccurlyeq T_2$ then $[T_1]_\equiv \preccurlyeq [T_2]_\equiv$.

Obviously, it is not possible to fully automatically find a meaningful name (or representative) from a set of primitive types. However, we found that it is possible to derive a suggestion for the type name by lexical analysis of the names of the variables that are of a given derived type. Our case study shows that in almost all cases these variables have a common substring. We suggest to use this string as base for the type-name.

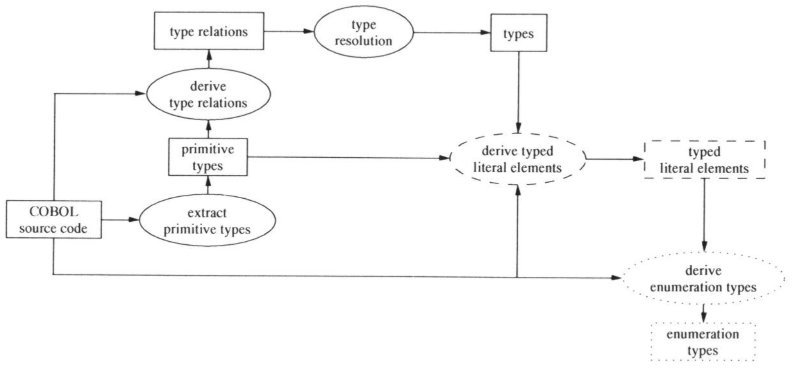


Figure 4.13: Type inferencing tool architecture.

4.9.1 Platform

We have implemented the architecture using the ASF+SDF Meta-Environment [Kli93, DHK96, BDK+96]. Furthermore, some pre- and post-processing was done using standard Unix tools like `perl`.

The ASF+SDF Meta-Environment is an interactive development environment for the algebraic specification of formal (programming) languages. It takes a syntax definition of a language and an algebraic specification that describes operations on programs written in that language. From these two, the system generates a programming environment that contains scanners, parsers and syntax-directed editors for the language, and tools that perform the specified operations on programs written in that language [DHK96].

To get an environment for analysing COBOL, we have instantiated the ASF+SDF Meta-Environment with a COBOL grammar [BSV97b] and generated *native patterns* and *traversal functions* from this grammar [BSV97a, SV98]. This gives us a tool that provides a default pass over the full abstract syntax tree of COBOL programs. This default pass can be specialised for particular constructs which allows us to focus only on the COBOL constructs that are important for our problem. In a single traversal of the source code we extract the primitive types, and derive the relations between types. Since the ASF+SDF Meta-Environment uses algebraic specifications, we were able to use the type-inference rules presented in the Figures 4.3, 4.4, 4.8, 4.9, 4.10, and 4.12 almost literally.

4.10 Case Study

In order to assess the effect of type inference on real life systems, we studied an existing legacy system called MORTGAGE,[2] a COBOL/CICS application of 100,000 lines of code. It consists of an on-line (interactive) part, as well as a batch part, and it is in fact a subsystem of a larger (1 MLOC) system.

We used the implementation of type inference described in the previous section to infer the equivalence classes as well the subtype relations between them. To enable us to assess the resulting types, we visualised the type relations as directed graphs in which variables are nodes, and arrows and lines represent subtype and equivalence relations respectively. Inspection of these graphs revealed the following issues.

First, assignments are the predominant factor responsible for creating type relations. In other words, COBOL programs contain more MOVE statements than (conditional) expressions.

Second, the sets of related (via subtyping or equivalence) variables are fairly small. For example, many variables are only once assigned to another variable. We encountered only very few cases in which there were more than 25 different variables involved. This is due to the fact that the types inferred reflect the actual *use* of variables. This gives an interesting comparison with languages that are strongly typed. In such languages, one would declare many different variables of type "int", which may be used for

[2] This system was also used as case study in [WBF97, DK98].

many different purposes. Type *inferencing* finds different types for all these purposes, based on their actual use (see also [OJ97]).

A question of interest is to what extent type pollution (inferring too many equivalences) as discussed in Section 4.6 is present in MORTGAGE, and whether the proposed solution, subtyping, is adequate. For most of the variables, pollution is not an issue, i.e., subtyping can be safely replaced by type equivalence. However, all forms of pollution as discussed in Section 4.6 do occur in MORTGAGE. Typical cases include the use of a single `MOVE` statement to initialise many different variables, the use of alphanumeric string variables to represent various types of strings, and the use of sections that use global variables to simulate formal parameters permitting values of different types (different sorts of keys, for example). In all these cases, subtyping provides the proper solution.

Many constants in MORTGAGE deal with enumeration types. Not all enumeration types in MORTGAGE contain a consecutive series of numbers: in some cases during maintenance certain numbers may have been removed; in other cases this indicates that a particular program deals with specific enumerated cases only.

Another group of constants occurring in MORTGAGE deals with *program names*, and are used in statements that invoke other modules, but in which the name of the module is contained in a variable. Our constant analysis helps to identify the possible values of such variables, which is necessary, for example, if one wants to derive the call graph of such programs.

In addition to the qualitative statements listed above, it would be useful to have some quantitative data on types as well, and to collect these for many different systems. We are in the process of collecting data such as the average and maximum of the size of equivalence sets, the number of types related via subtyping, and the number of supertypes per type; the number of equivalence relations divided by the number of subtype relations; and the percentage of declared variables that is never used (which may be up to 10%).

4.11 Concluding Remarks

4.11.1 Applications

Type inference for COBOL systems has many applications. We have presented one, literal analysis, in considerable detail in Section 4.8. Here we discuss a selection of other applications.

One of the most direct applications of type inferencing is in tool support for year 2000 and Euro conversions. Type inferencing will find a number of types, and matching on names or record structures in these types will classify certain types as "year", "month", "two-digit date", "currency", etc. Indeed several of the published year 2000 solutions [HP96, KMUO98] search for date-infections by propagating date-seeds via an equivalence relation between variables that is very similar to inferred type equivalence. Moreover, type inferencing can be used to realize the *static date analyser* discussed in [DWQ97].

An application using all types rather than just the date-related ones is migrating Cobol systems to a typed language, such as Pascal or C.

One step further is migrating Cobol to an object oriented language. A typical route is to use *subsystem classification techniques* [Lak97] for that purpose, which aim at decomposing a large system into, potentially reusable, components or classes. This is generally by applying a numerical clustering algorithm to group syntactic units based on various interconnection relations. One way is to group procedures based on the types they use. As Lakhotia [Lak97] remarks, however, this technique cannot be used if the source language does not support types. Type inferencing makes these techniques available for the Cobol domain as well.

A rather different potential application of type inferencing is during software maintenance: if types are inferred both before and after the modifications, a presentation of the difference between the inferred type sets to the programmer may help to detect inconsistencies and potential errors: for example, if the new typing scheme unifies two old types that are perceived as different, the modification made may contain an error.

4.11.2 Related Work

A principal source of inspiration to us was Lackwit, a tool for understanding C programs by means of type inference [OJ97]. New in our work is not only the significantly different source language: Also new is the inference of subtyping for assignments, and the use of type inference to classify literals.

The approach of Kawabe *et al.* [KMUO98] uses an equivalence relation between variables to deal with the year 2000 problem, which is similar to our inferred type equivalence. They pay a lot of attention to *noise reduction*, but have no solution similar to our subtyping approach. They formulate their work in terms of Cobol, and do not provide a formal type system. They discuss year 2000 as an application.

Chen *et al.* [CTJ+94] describe a Cobol *variable classification* mechanism. They distinguish a fixed set of categories, such as input/output, constant, local variable etc. They provide a set of rules to infer these automatically, essentially using data flow analysis. Their technique is orthogonal to ours: types we infer can be used in local or global variables, for database output or not, etc.

Newcomb and Kotik [NK95] describe a method for migrating Cobol to object orientation. Their approach takes all level 01 records as starting point for classes. Records that are structurally equivalent, i.e., matching in record length, field offset, field length, and field picture, but possibly with different names, are considered "aliases". According to Newcomb and Kotik, "for complex records consisting of 5-10 or more fields, the likelihood of false positives is relatively small, but for smaller records the probability of false positives is fairly large." [NK95, p. 240]. Our way of type inferencing provides a complementary way of grouping such 01 level records together, and will help to reduce this risk of false positives for small records.

Wegman and Zadeck [WZ91] describe a method to detect whether the value of a variable occurring at a particular point in the program is constant and, if so, what that value is. Merlo *et al.* [MGHDM95] describe an extension of this method that allows detection of all constants that can be the value of a particular variable occurrence.

This differs from our approach which finds all constants that can be assigned to *any* variable of a given type. Furthermore, the methods described in both papers take the flow of control into account where as our approach is flow insensitive (control flow is completely ignored). Consequently, their results are more precise (e.g., we report constants that are used in dead code) but their approach is also more expensive.

Gravley and Lakhotia [GL96] identify enumeration types that are modelled using symbolic constants. Their approach is orthogonal to ours since they group constants which are *defined* in the same context whereas we group constants based on their *usage* in the source code.

4.11.3 Future Work

We are currently in the process of extending our work in the following ways:

- Inference of input and output parameters for COBOL sections and paragraphs, by means of data flow analysis [Moo97]. This information can then be used to refine the inferred subtype relations.
- Extension of the empirical results, in order to further demonstrate the usefulness of type inferencing, and to assess the validity of the choices made. In particular, we want to apply our technique to other COBOL systems and collect quantitative data on the inferred types.[3]
- We are working on applying type inferencing to component extraction, following [Lak97, DK98].[4]
- Extension to new languages, most notably Fortran and IBM 370 assembler.
- Visualization of the inferred equivalence and subtype relations, the typed literal and enumerations types on the level of COBOL programs as well as visualization of (the usage of) system-level types in complete COBOL systems.[5]

4.11.4 Contributions

In this chapter we have proposed a formal system for inferring types from COBOL programs, which we explained by means of a number of real-life COBOL fragments. We formulated rules for inferring type equivalence classes, and we discussed how subtype relations can be inferred to refine the analysis and deal with, for example, variables representing lines to be printed or variables simulating input parameters. We discussed a number of applications, most notably the use of type inference to introduce variables for literals occurring in statements. We have implemented the type inference rules in the ASF+SDF Meta-Environment [Kli93, DHK96] and successfully applied this tool to a real life, 100,000 lines of code COBOL system.

[3] This work is reported on in the next chapter (Chapter 5).

[4] Support for object identification that is based on the combination of inferred types with concept analysis is presented in Chapter 6.

[5] The visualization of these aspects in the context of using types to support software exploration and re-documentation is described in Chapter 7.

Acknowledgements

We would like to thank Paul Klint (CWI) and Joost Visser (University of Amsterdam) for reading earlier versions of this chapter, and the anonymous WCRE'98 referees who were able to make useful comments and suggestions in spite of the font problem with the version they reviewed.

CHAPTER 5

An Empirical Study into COBOL Type Inferencing

For a typical COBOL program, the data division consists of 50% of the lines of code. Automatic type inference can help to understand the large collections of variable declarations contained therein, showing how variables are related based on their actual usage. The most problematic aspect of type inference is pollution, the phenomenon that types become too large, and contain variables that intuitively should not belong to the same type.

The aim of the chapter is to provide empirical evidence for the hypothesis that the use of subtyping is an effective way for dealing with pollution. The main results include a tool set to carry out type inference experiments, a suite of metrics characterizing type inference outcomes, and the conclusion that only one instance of pollution was found in the case study conducted. The work presented in this chapter was published earlier as [DM01].

5.1 Introduction

In this chapter, we will be concerned with the variables occurring in a COBOL program. The two main parts of a COBOL program are the *data division*, containing declarations for all variables used, and the *procedure division*, which contains the statements performing the program's functionality. Since it is in the procedure division that the actual computations are made, one would expect this division to be *larger* than the data division. Surprisingly, we found that in a typical COBOL system this is not the case: the data division often comprises more than 50% of the lines of code.[1] We even encountered several programs in which 90% of the lines of code were part of the data division.

[1] For three different systems, each approximately 100,000 LOC, we found averages of 53%, 43%, and 58%, respectively.

(As we have seen in the previous chapter, one reason for this is that COBOL does not distinguish between type and variable declarations.)

These figures have two implications. First of all, they suggest that only a subset of all declared variables are actually used in a COBOL program. If 90% of the lines are variable declarations, it is unlikely that the remaining 10% will use all these variables. Indeed, in the systems we studied, we have observed that less than 50% of the variables declared are used in the procedure division.[2]

These figures also indicate that maintenance programmers need help when trying to understand the data division part. Just reading the data division will involve browsing through a lot of irrelevant information. Thus, the minimal help is to see which variables are in fact used, and which ones are not. In addition to that, the maintenance programmer will want to understand the relationships that hold between variables. In COBOL, some of these relations can be derived from the data division, such as whether a variable is part of a larger record, whether it is a redefine (alias) of another variable, or whether it is a predicate on another variable (level 88).

But not all relevant relations between variables are available in the data division. When do two different variables hold values that represent the same business entity? Can a given variable ever receive a value from some other given variable? What values are permitted for this variable? Is the value of this variable ever written to file? Is the value of this variable passed as output to some other program? What values are actually used for a given variable? What are the operations permitted on a given variable?

In strongly typed languages, questions like these can be answered by inspecting the *types* that are used in a program. First, a type helps to understand what set of values is permitted for a variable. Second, types help to see when variables represent the same kind of entities. Third, they help to hide the actual representation used (array versus record, length of array, ...), allowing a more abstract view of the variable. Last but not least, types for input and output parameters of procedures immediately provide a "signature" of the intended use of the procedure.

Unfortunately, the variable declarations in a COBOL data division suffer from a number of problems that make them unsuitable to fulfill the roles of types as discussed above. In COBOL, it is not possible to separate type definitions from variable declarations. This has three unpleasant consequences. First, when two variables need the same record structure, this structure is *repeated.* Second, whenever a data division contains a repeated record structure, the lack of type definitions makes it difficult to determine whether that repetition is accidental (the two variables are not related), or whether it is intentional (the two variables should represent the same sort of entity). Third, the absence of explicit types leads to a lack of abstraction, since there is no way to hide the actual representation of a variable into some type name.

In short, the problem we face with COBOL programs is that types are needed to understand the myriads of different variables, but that the COBOL language does *not* support the notion of types.

[2] For the MORTGAGE system under study in this chapter, on average 58% of the variables declared in a program were never used, the percentages ranging from 2.6% for the smallest up to 95% for the largest program.

In Chapter 4, we have proposed a solution to this problem. Instead of deriving type information from the data division, we perform a *static analysis* on the programs to *infer* types from the usage of variables in the procedure division. The basic idea of type inference is simple: if the value of a variable is assigned or compared to another variable, we want to infer that these two variables should have the same type. However, just inferring a type *equivalence* for every assignment will not do. As an example, a temporary string value could receive values from names, streets, cities, etc., which should all have different types. Via transitivity of equivalence, however, all variables assigned to that string variable would receive the same type. This phenomenon, that a type equivalence class becomes too large, and contains variables that intuitively should not belong to the same type, is called *pollution*. In order to avoid pollution, we have proposed to introduce *subtyping* for assignments rather than type *equivalence* (Chapter 4).

In this chapter, we will carefully study the problem of pollution, and test the hypothesis that it is handled by deriving subtypes rather than equivalences. This is done by presenting statistical data illustrating the presence of pollution, and the effectiveness of subtyping for dealing with it. In particular, we look at the interplay between subtyping and equivalence (For example, consider two types T_A and T_B. When we have $T_A \preccurlyeq T_B$, and $T_B \preccurlyeq T_A$, we get $T_A \equiv T_B$ — How does this affect pollution?).

Moreover, we will discuss how *relational algebra* can be used for implementing Cobol type inferencing. Relational algebra has recently been proposed as a valuable tool for various reverse engineering and program understanding activities, such as architecture recovery [Hol98, FKO98]. It is based on Tarski's relational operators [Tar41], such as union, subtraction, relational composition, etc. The use of relational algebra helps us to completely separate Cobol-specific source code analysis from calculating with types. Moreover, it enables us to specify type relationships at an appropriate level of abstraction.

All experiments are done on Mortgage, a real-life Cobol/cics system from the banking environment. This system consists of 100,000 lines of code; with all copybooks (include files) expanded (unfolded), it consists of 250,000 lines of code. It conforms to the Cobol-85 standard, which is the most widely used Cobol version. Compared to a Cobol code base of 3 million lines we have available, Mortgage contains fairly representative Cobol code (it is neither the worst nor the best code).

5.2 Type Inference

In this section, we summarize the essentials of Cobol type inferencing: a more complete presentation is given in Chapter 4. We start by describing the *primitive types* that we distinguish. Then, we describe how *type relations* can be derived from the statements in a single Cobol program, and how this approach can be extended to *system-level analysis* leading to inter-program dependencies. Finally, we show how the analysis can be extended to include types for *literals*, discuss the notion of *pollution*, and conclude with an example.

5.2.1 Primitive Types

We distinguish three primitive types: (1) elementary types such as numeric values or strings; (2) arrays; and (3) records. Initially every declared variable gets a unique primitive type. Since variable names qualified with their complete record name must be unique in a Cobol program, these names can be used as labels within a type to ensure uniqueness. Furthermore, we qualify variable names with program or copybook names to obtain uniqueness at the system level. We use T_A to denote the primitive type of variable A.

5.2.2 Type Equivalence

By looking at the *expressions* occurring in statements, an *equivalence relation* between primitive types can be inferred. We distinguish three cases:

1. *Relational expressions:* from a relational expression such as $v = u$ or $v \leq u$ an equivalence between T_v and T_u is inferred.

2. *Arithmetic expressions:* from an arithmetic expression such as $v + u$ or $v * u$ an equivalence between T_v and T_u is inferred.

3. *Array accesses:* from two different accesses to the same array, such as $a[v]$ and $a[u]$ an equivalence between T_v and T_u is inferred.

When we speak of a *type* we will generally mean an *equivalence class of primitive types*. For presentation purposes, we may also give names to types based on the names of the variables part of the type. For example, the type of a variable with the name `L100-DESCRIPTION` will be called `DESCRIPTION`-type.

5.2.3 Subtyping

By looking at the *assignment statements*, we infer a *subtype relation* between primitive types. Note that the notion of assignment statements corresponds to Cobol statements such as `MOVE`, `COMPUTE`, `MULTIPLY`, etc. From an assignment of the form $v := u$ we infer that T_u is a *subtype* of T_v, i.e., v can hold at least all the values u can hold.

5.2.4 Union Types

From a Cobol *redefine clause*, we infer a *union type* relation between primitive types. When a given entry v in the data division redefines another entry u, we infer that T_v and T_u are part of the same *union type*.

5.2.5 System-Level Analysis

In addition to inferring type relations within individual programs, we derive type relations at the system-wide level. We infer that the types of the actual parameters of a program call (listed in the COBOL USING clause) are subtypes of the formal parameters (listed in the COBOL LINKAGE section), and that variables read from or written to the same file or table have equivalent types. Furthermore, we want to ensure that if a variable is declared in a copybook, its type is the same in all the different programs that copybook is included in. In order to do this, we derive relations that denote the origins of primitive types and the import relation between programs and copybooks. These relations are then used to link types via copybooks.

5.2.6 Literals

A natural extension of our type inference algorithm involves the analysis of literals that occur in a COBOL program. Whenever a literal value l is assigned to a variable v, we conclude that the value l must be a permitted value for the type of v. Likewise, when v and l are compared, l is considered a permitted value for the type of v. Literal analysis indicates permitted values for a type. Moreover, if additional analysis indicates that variables in this type are only assigned values from this set of literals, we can infer that the type in question is an *enumeration type*.

5.2.7 Pollution

The intuition behind type equivalence is that if the programmer would have used a typed language, he or she would have chosen to give a single type to two different COBOL variables whose types are inferred to be equivalent. We speak of *type pollution* if an equivalence is inferred which is in conflict with this intuition.

Typical situations in which pollution occurs include the use of a single variable for different purposes in different program slices; the use of a global variable acting as a formal parameter, to which a range of different variables can be assigned; and the use of a PRINT-LINE string variable for collecting output from various variables.

5.2.8 Example

Figure 5.1 contains a COBOL fragment illustrating various aspects of type inferencing. The first half contains the declarations of variables, containing their physical types, i.e., how many bytes they occupy. The second half contains the actual statements from which type relations between variables are inferred.

Going from bottom to top, we first see (line 41) that variable A00-FILLED is compared to N100, from which we infer that they belong to the same type. From line 39, we then infer an additional type equivalence, adding A00-MAX to this equivalence class. We thus obtain one type, for three different variables. If we also take a look at the data division, we see that this equivalence is in accordance with their declared picture layouts (in lines 13, 14, and 20), which are all numeric data elements. However, we cannot infer such equivalences from just the pictures, as entirely unrelated data structures may share the same physical layout (for example, N200 in line 21).

An assignment example is given in line 31, where NAME is assigned to NAME-PART. Here we infer that the type of NAME is a *subtype* of NAME-PART. In line 26, another vari-

```
/ variables containing business data.
 01     PERSON-RECORD.
    03 INITIALS         PIC  X(05).
    03 NAME             PIC  X(27).
    03 STREET           PIC  X(18).
    ...

/ variables containing char array of length 40,
/ as well as several counters.
 01     TAB000.
    03 A00-NAME-PART.
       05 A00-POS       PIC  X(01)  OCCURS 40.
    03 A00-MAX          PIC S9(03)  COMP-3 VALUE 40.
    03 A00-FILLED       PIC S9(03)  COMP-3 VALUE ZERO.

 ...

/ other counters declared elsewhere.
 01 N000.
    03  N100            PIC S9(03)  COMP-3 VALUE ZERO.
    03  N200            PIC S9(03)  COMP-3 VALUE ZERO.
 ...

/ procedure dealing with initials.
 R210-VOORLT SECTION.
    MOVE INITIALS TO A00-NAME-PART.
    PERFORM R300-COMPOSE-NAME.

/ procedure dealing with last names.
 R230-NAME SECTION.
    MOVE NAME TO A00-NAME-PART.
    PERFORM R300-COMPOSE-NAME.

/ procedure for computing a result based on the
/ value of the A00-NAME-PART.
/ Uses A00-FILLED, A00-MAX, and N100 for array indexing.
 R300-COMPOSE-NAME SECTION.
    ...
    PERFORM UNTIL N100 > A00-MAX
       ...
       IF A00-FILLED = N100
         ...
```

Figure 5.1: Excerpt from a real-life Cobol program.

able, `INITIALS`, is assigned to `NAME-PART` as well, giving rise to a second subtype relationship, now between `INITIALS` and `NAME-PART`. In this way, `INITIALS` and `NAME` share a common supertype (`NAME-PART`), but there is no direct relationship inferred between them. If we look at the declared physical layout we see that all three are strings of a different length (in lines 3, 4, and 12). `NAME-PART` is the largest, capable of accepting values from both `INITIALS` and `NAME`.

In fact, `NAME-PART` is a global variable acting as a *formal parameter* for the procedure `R300-COMPOSE-NAME` (COBOL does not support the declaration of parameters for procedures). What we infer is that the type of the actual parameter is a subtype of the formal parameter. Just deriving equivalences from assignments would lead to *pollution*: it would give all the actual parameters, in this case the two different concepts "initials" and "first name", the same type.

5.2.9 Practical Value

COBOL type inferencing provides a theory for grouping variables based on their usage. This is of great practical value for the understanding and (semi-automated) transformation of COBOL legacy systems. Example application areas are discussed in Chapter 4, and include the introduction of symbolic names for literal values (per type), extraction of system interfaces based on parameter types, migration to strongly typed languages such as Pascal, identification of candidate classes in legacy systems, and type-related modifications such as the Euro and year 2000 problem.

Another major application is to use type inferencing to support the migration of COBOL to the new COBOL standard, which is an object-oriented extension of COBOL-85 [ISO00]. This new version of COBOL does support types, and offers the possibility of using type definitions. Type inferencing supports the detection of these types in existing COBOL programs, thus allowing old systems to benefit from the new language features.

5.3 Implementation using Relational Algebra

This section describes how we use relational algebra to implement type inference for COBOL systems. We start by giving an overview of the tool architecture. Then, we describe the facts that are derived from COBOL sources. We continue with a discussion of how these facts are combined and abstracted to infer more involved type relations. Finally, we describe the extension of this approach to the system level.

5.3.1 Tool Architecture

The set of tools we use for applying type inference to COBOL systems is shown in Figure 5.2. It separates source code analysis, inferencing and presentation, making it easier to adapt the toolset to different source languages or other ways of presenting the types found.

In the first phase, a collection (database) of *facts* is derived from the COBOL sources. For that purpose, we use a parser generated from the COBOL grammar discussed in [BSV97a]. The parser produces abstract syntax trees (ASTs) in a textual representation called the AsFix format. These ASTs are then processed using a Java package which implements the visitor design pattern [GHJV94]. The fact extractor itself is a refinement of this visitor which emits type facts at every node of interest (for example, assignments, relational expressions, etc.).

In the second phase, the derived facts are combined and abstracted to infer a number of conclusions regarding type relations. Both facts and conclusions are stored in a simple ASCII format, as also used in, for example, Rigi [MOTU93]. One of the tools we use for inferring type relations is GROK [Hol98], a calculator for *relational algebra* [Tar41]. Relational algebra provides operators for relational composition, for computing the transitive closure of a relation, for computing the difference between two relations, and so on. We use it, for example, to turn the derived type facts into the required equivalence relation. In addition to relational algebra, we use Unix tools like `sort`, `uniq`, `awk`, etc. to manipulate the relation files.

In the final phase, we pass information about the type relations to the end-user. In this chapter, we conduct an analysis of the effects of pollution, for which we collect and present a range of statistical data. Other options include the generation of data structures in a language supporting explicit type definitions, and visualization of type information via graphs.

5.3.2 Derived Facts

The different kinds of facts derived from the COBOL sources are listed in Figure 5.3. The contain and union relations are derived from the data division, the remaining ones from the procedure division.

Observe that the relations in this figure indicate the degree of language independence of type inferencing: it can be applied to any language from which these facts can be derived. Other languages like Fortran, C, or IBM 370 assembly, can be analyzed by

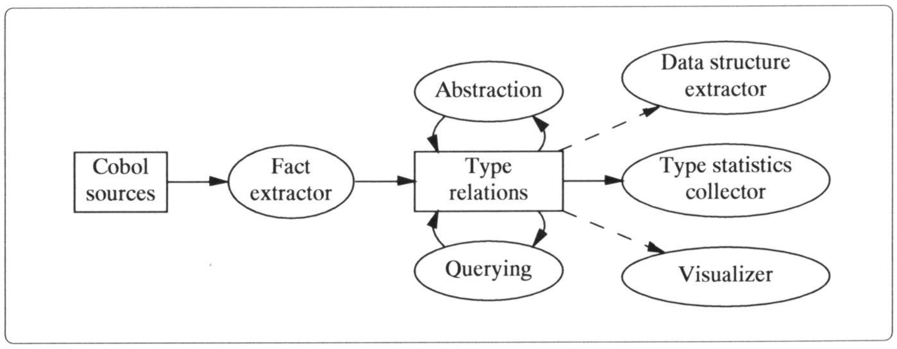


Figure 5.2: Overview of the type inference tool set.

relation	dom	rng	description
assign	T_v	T_u	an expression of type T_v is assigned to a variable of type T_u
expression	T_v	T_u	variables of types T_v and T_u are used in the same expression
arrayIndex	T_a	T_i	variable of type T_i is used as index in array of type T_a
contain	T_r	T_f	structured type T_r contains T_f
union	T_v	T_u	types T_v and T_u are part of the same union type
literalAssign	T_v	l	literal l is assigned to a variable of type T_v
literalExp	T_v	l	literal l is compared to a variable of type T_v
arrayLitIdx	T_a	l	literal l is used as index in array of type T_a

Figure 5.3: Derived Facts.

adding a parser and fact extractor for those languages. Furthermore, since the facts for different languages can easily be combined, this approach allows for the transparent analysis of multi-language systems where, for example, some parts are written in Cobol and other parts are written in assembly.

5.3.3 Inferred Relations

The resolution process infers relations between types from the facts that were derived from the Cobol system. Our resolution process is based on relational algebra and is implemented using grok [Hol98].

The three key relations inferred are typeEquiv, subtypeOf, and literalType, summarized in Figure 5.4. Besides the relations in Figure 5.4, some auxiliary relations are inferred. These include: arrayIndexEquiv for equivalence of types through array access (if variables i and j are used as indexes for the same array A, their types should be equivalent), subtypeEquiv for type equivalence through subtyping (if $T_A \preccurlyeq T_B$ and $T_B \preccurlyeq T_A$, we get $T_A \equiv T_B$), and transSubtypeOf for the transitive closure of subtypeOf.

The resolution algorithm is outlined in pseudo code in Figure 5.5. The operators used are those of relational algebra and can be mapped directly to grok operators. Note that function abstraction and unbounded iteration are not available in grok. For

relation	dom	rng	description
typeEquiv	T_1	T_2	type T_1 is equivalent to type T_2
subtypeOf	T_1	T_2	type T_1 is subtype of type T_2
literalType	T	l	type T contains literal l

Figure 5.4: Inferred Relations.

```
arrayIndexEquiv := arrayIndex$^{-1}$ $\circ$ arrayIndex
typeEquiv := arrayIndexEquiv $\cup$ expression
subtypeOf := assign
**repeat**
  subtypeEquiv := equiv(subtypeOf$+$ $\cap$ (subtypeOf$+$)$^{-1}$)
  typeEquiv := equiv(typeEquiv $\cup$ subtypeEquiv)
  subtypeOf := subtypeOf $\setminus$ typeEquiv
  subtypeOf := subtypeOf $\cup$ subtypeOf $\circ$ typeEquiv
                                  $\cup$ typeEquiv $\circ$ subtypeOf
**until fixpoint of** (typeEquiv, subtypeOf)
literalType := typeEquiv $\circ$ (literalExp $\cup$ literalAssign
                           $\cup$ (arrayIndex$^{-1}$ $\circ$ arrayLiteralIndex))

**fun** equiv($R$) := ($R$ $\cup$ $R^{-1}$)$*$
```

Figure 5.5: Outline of the resolution algorithm.

this reason, in the actual implementation the functions were written out explicitly and bounded iteration is used. The number of iterations was determined heuristically; for the case study conducted, 5 iterations were sufficient. We were informed that addition of unbounded iteration is considered for future releases of GROK.

5.3.4 System-Level Types

In order to do system-level type inference, the primitive types have to be unique for the whole system. As described in Section 5.2.5, this can be done by qualifying them with program names. Primitive types derived from copybooks that are included in the data division should be qualified using the copybook's name — this ensures that variables of those types will have the same type in all the programs that this copybook is included in.

However, this approach does not allow us to deal with system-level type inference without loading all COBOL sources in memory at once. We would need to analyze self-contained clusters of programs and copybooks, in order to qualify types with the

relation	dom	rng	description
decl	m	T_v	module m declares T_v
copy	m_1	m_2	module m_1 imports m_2
actualParam	$P.n$	T_v	nth actual parm. of P has type T_v
formalParam	$P.n$	T_v	nth formal parm. of P has type T_v

Figure 5.6: Derived System-Level Relations.

correct names. Such clusters are likely to become as large as the complete system.

To facilitate complete separation of the analysis of copybooks and programs, we derive all information as before, and add extra facts from COBOL sources concerning the use of copybooks and declaration of types. The extra relations are described in Figure 5.6.

Next, we compose the copy and decl relations, and infer a copyOf relation that indicates which types used in a program are actually "copies" of types that were declared in a copybook (Figure 5.7). This join is done on the imported module field m_2 of the copy relation with the module field m of the decl relation.

Finally, the copyOf relation between T_p and T_c is interpreted as a substitution on the derived relations replacing all occurrences of T_p by T_c. This substitution propagates type dependencies through copybooks.

At this point we have achieved the same database as we would have obtained by analyzing all sources at once, but now using a *modular* approach. Such a modular approach allows us to analyze large industrial-scale systems that are too big to be handled in memory at once.

Example 5.3.1 *Suppose we derive the following information from programs P and Q:*

subtypeOf P.A P.B	*copy P Z*	*decl Z Z.B*
subtypeOf Q.B Q.C	*copy Q Z*	

Program P and Q both use variable B and import copybook Z in which B is declared.

Joining the copy and decl relations yields two copyOf facts:

copyOf P.B Z.B *copyOf Q.B Z.B*

After substituting these in subtypeOf, we get:

subtypeOf P.A Z.B *subtypeOf Z.B Q.C*

Observe that, via transitivity of the subtypeOf relation, we can now infer that P.A is a subtype of Q.C, a relation that could not have been found without the propagation through the copybook.

We have written a dedicated C program to perform the substitution since standard Unix tools like `sed` or `perl` could not handle the amount of substitutions involved.[3] Time complexity of this program is $O(n \log n + m \log n)$ (where n is the number of tuples in copyOf, and m is number of tuples in the database), and its space requirements are $O(n)$.

[3] For example, for MORTGAGE, the copyOf relation contains 121,915 tuples.

relation	dom	rng	description
copyOf	T_p	T_c	T_p is a copy of T_c

Figure 5.7: Inferred System-Level Relations.

5.4 Assessing Derived Facts

In this section we study the nature of the facts that can be directly derived from the COBOL sources, i.e., without applying the resolution step. This means that we only look at the intra-module dependencies, and only consider direct subtype relationships, not transitive ones. This will help us to understand to what extent individual programs are responsible for causing pollution. In the next section, we will look at inter-module dependencies, and relationships arising from taking the transitive closure.

The database that is derived from the MORTGAGE sources contains 34,313 unique facts. An overview of these is shown in Figure 5.8. All duplicates were removed, thus, if variable v is assigned to variable u in two different statements in a certain program, this results in only one subtype relation between T_v and T_u. The majority of facts are from decl and contain. Type equivalence and subtype relationships are inferred from the remaining facts. An interesting observation is that the assign relation is almost 9 times as large as the expression relation. This means that variables in a COBOL program are much more often moved around (assigned) than tested for their value. In this section, we will particularly look at these assign-facts.

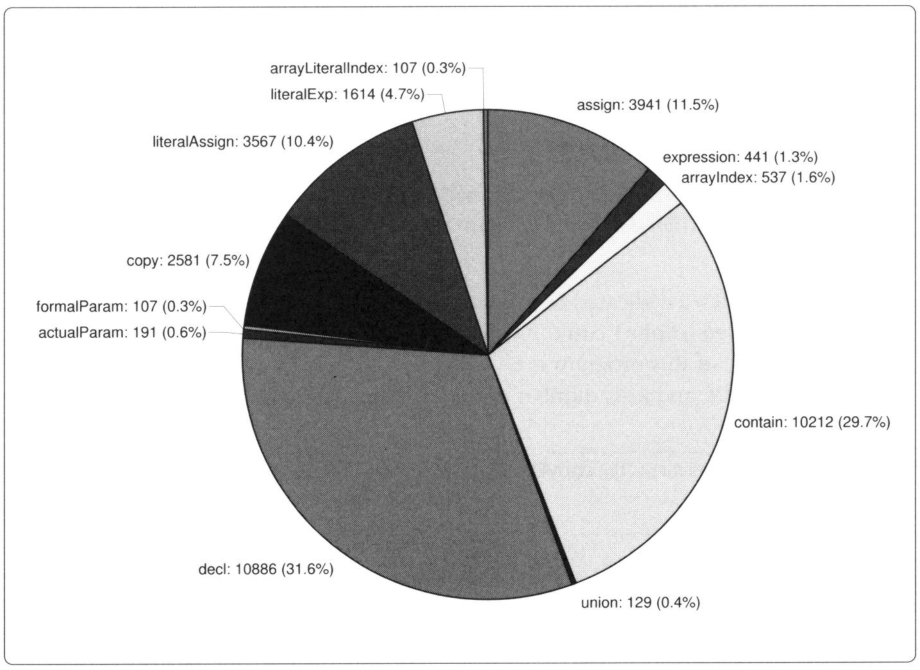


Figure 5.8: Facts derived from MORTGAGE.

5.4.1 Direct Subtypes per Type

A variable that receives values from many different other variables is a potential cause for pollution. Therefore, in this section we will search for those types that have many different subtypes, i.e., types of variables that are assigned values from many other variables.

In Figure 5.9 we show, for each program, the highest number of different subtypes that a single type has. The numbers at the x-axis can be seen as program IDs – they are given in order of increasing program size. As an example, the program with ID-number 20 (one of the smaller programs) has a pulse of length 2 associated with it, i.e., the type with the most different subtypes just has 2 different subtypes.

The dashed line indicates the *average* number of subtypes per type. It shows that most types have just 1 or 2 subtypes. To compute the average number of subtypes per type, only those types that have at least one subtype were taken into account (hence this average will always be larger than 1), ignoring types that were not used at all, or only in expressions. The overall average number of subtypes is 1.18.

Most programs do not contain types with more than 5 subtypes; one program contains a type with an exceptionally large number of 45 different subtypes. If we look at the COBOL code underlying these data, we can understand the high maximum of 45. This involves the type of a variable called `P800-LINE`, which is a string of length 132. It acts as the formal parameter of a section called `Y800-PRINT-LINE`. Whenever data is

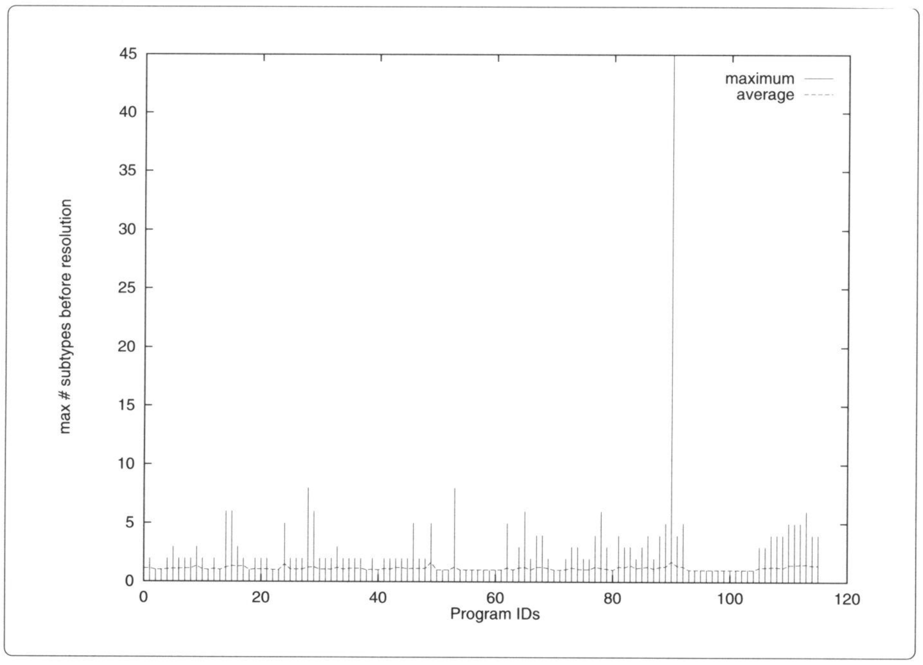


Figure 5.9: Max. no. of subtypes per program before resolution.

to be printed, it is moved into that variable and the Y800-PRINT-LINE section is called. Type inference concludes that the types of all the variables that are printed this way, are subtypes of the type of Y800-PRINT-LINE.

5.4.2 Direct Supertypes per Type

Another figure of interest consists of the number of *supertypes* per primitive type, i.e., types of variables that are assigned to many other variables. Figure 5.10 shows the number of supertypes per type. Again, most types that have a supertype have one or two supertypes, the average being 1.32. Most of the maxima are below 6, but a number of programs contain types with many more supertypes, for example with 17, 18, or 19 different ones.

If we look at the Cobol source code, we can explain the role of these types. The type with 19 supertypes occurs in a (fairly large) program with ID-number 104, and turns out to be the type of a CURSOR variable, used in a CICS interactive setting. We will refer to this type as CURSOR-type. The variable of this type navigates through the screen positions of a terminal. It is compared with, and copied into a number of different variables representing screen positions of certain fields, such as the position where to enter the name of a person. All these positions together, each declared with numeric picture, share one subtype: the CURSOR-type. Thus, the number 19 is not due to pollution, but rather provides meaningful information for understanding the program,

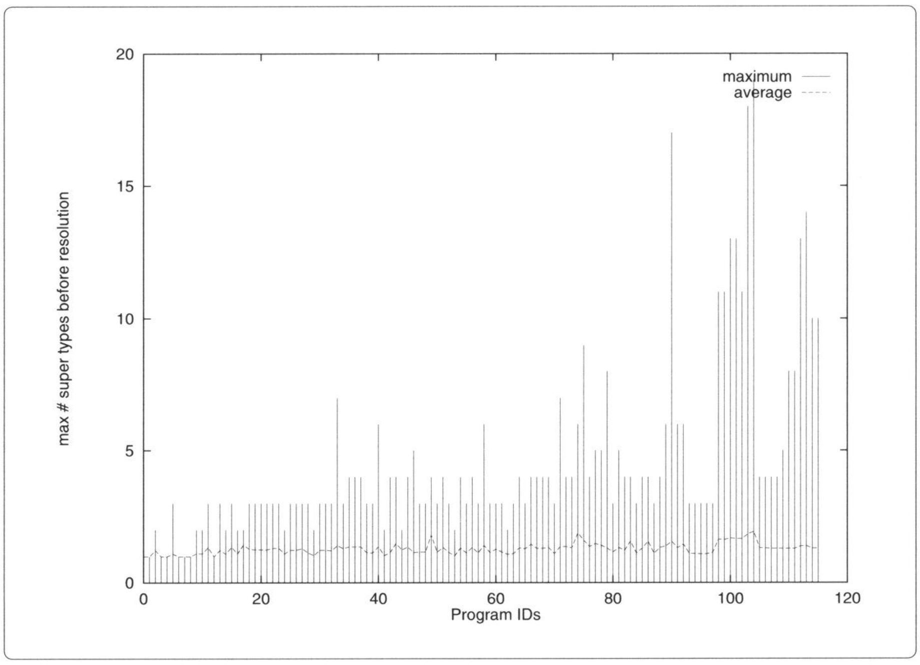


Figure 5.10: Max. no. of supertypes per program before resolution.

namely that all these types share the values of their common CURSOR-subtype. This CURSOR-mechanism is used by many different programs, and thus is the explanation for most of the maxima higher than 6 occurring in Figure 5.10.

One of the non-CURSOR cases occurs in the program with ID-number 90. It concerns a so-called DESCRIPTION-type which has 17 different supertypes. It is the type of an output field of a procedure for reading a value from a particular database. The particular database contains a wide variety of data, and depending on some of the input parameters, different sorts of data are returned. Each of these becomes a supertype of the DESCRIPTION-type.

5.4.3 Type Equivalence

In addition to looking at the subtype relations, we can look at the direct type equivalence relations we derive, i.e., we look at types that occur in the same relational or arithmetic expressions. The statistics derived needed for this is based on fewer input tuples, as we know from Figure 5.8 that there are almost 9 times fewer expression tuples than assign tuples. The resulting figure, however, is quite similar to Figure 5.10, so we omitted the figure.

If we look at the maxima, they are again 19, 18, and lower. As with the supertypes, one of the types responsible for this is the CURSOR-type. A variable of this type is compared with 18 other variables. Therefore, we conclude that the types of these 18 variables must be the same as the CURSOR-type. The resulting type represents a screen position.

Another type that is equivalent to many other types is the so-called DFHBMEOF-type. This is the type of a special CICS variable which has a constant value for a certain control character. After reading the input entered from a screen, the status characters for the strings that were read are compared with this CICS variable. The types of those status characters are thus equivalent to the type of that CICS variable in our approach.

5.5 Assessing Inferred Relations

In this section we examine the relations that result from applying the resolution step. This will help us to understand the merits of resolution and how it affects type pollution.

Before executing the resolution process, we prepare the derived facts for system-level analysis. The copyOf relation that is inferred from the copy and decl relations contains 121,915 tuples. The propagation of copyOf information in the derived database takes 6 seconds. The resolution was done using a GROK script implementing the algorithm in Figure 5.5 which takes 7 minutes for the case study at hand (on a Sun Ultra 10 (300MHz), 576 Mb memory).

After resolution, the database contains 202,848 tuples. An overview of these is shown in Figure 5.11. For a number of relations (such as arrayIndex or literalExp), the number of tuples in the resulting database is *smaller* than before since the substitution results in some tuples becoming duplicates. For others, such as subtypeOf, the number of tuples increases, via propagation of the equivalence relation.

5.5.1 Subtype Relation

One of the goals of the resolution process is to improve the subtypeOf relation by removing tuples for which we have more specific information, namely that they are part of the typeEquiv relation. On the other hand the subtypeOf relation is also extended with information of the typeEquiv relation. For example, if $T_A \preccurlyeq T_B$ and $T_B \equiv T_C$ then also $T_A \preccurlyeq T_C$. The percentage of subtypes that are added or removed as a result of both modifications is shown in Figure 5.12.

In this figure we see that for most programs, resolution reduces the number of subtypes. The average reduction in these programs is 18.4% with a maximum of 47.1%. There are however a couple of programs in which the number of subtypes grows. The average growth in these programs is 54.5% and the maximum is 393.8%. Inspection of these programs shows that the cause of these large numbers is again the CURSOR that was earlier described in Sections 5.4.2 and 5.4.3. The reason for this is that CURSOR is the subtype of a lot of types (say set S), *and* it is equivalent to a number of types (say set E). Since the resolution process ensures that all types in set E become subtypes of all the types in set S, the resulting database contains a rather large number of subtypes ($|S| \times |E|$ to be precise) just because of this CURSOR.

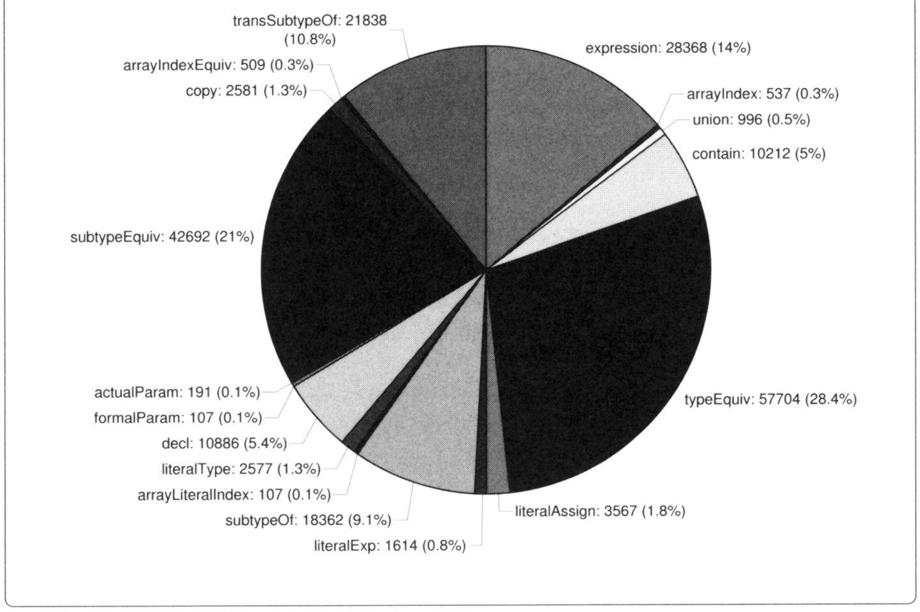


Figure 5.11: Information inferred from MORTGAGE.

As not all variables are used in comparisons (recall that in COBOL it is very common to just move variables), other types with many sub- or supertypes (such as `DESCRIPTION` and `P800-LINE`) but which are never used in comparisons, play no role of importance here.

5.5.2 Type Equivalence

The typeEquiv partitions types into equivalence classes. An overview of all classes that occur in MORTGAGE and their sizes is presented in Figure 5.13.

Figure 5.13(a) contains the classes if resolution is only done on a per-program basis, i.e., without taking system-wide propagation via copybooks and program calls into account. On this program level, resolution does not have a big influence on these equivalence classes. The explanation for this is that the classes at the program level are small and tightly connected, so all relations are already found by analyzing the code (e.g., if 3 variables are equivalent, they will all be compared to each other so the transitive closure does not find new tuples). The maxima are still 19 and 18 and the average class size is 3. Furthermore, approximately 90% of the classes have less than 5 equivalent elements.

Things get more interesting at the system level presented in Figure 5.13b. The maximum class size jumps to 201, followed by 118 but the total number of different classes

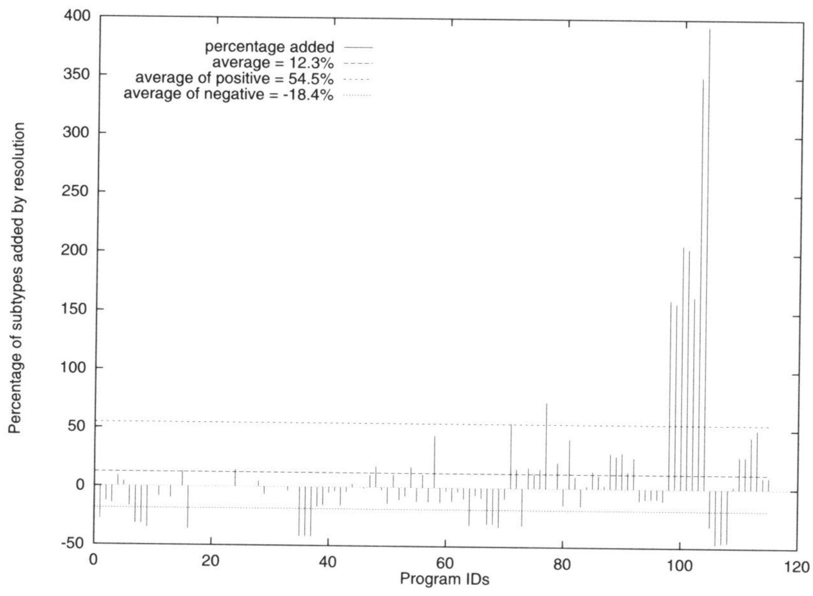


Figure 5.12: Subtypes added by resolution.

drops to 191, one third of the number of classes before resolution. Again, approximately 90% of the classes have less than 5 equivalent elements.

Inspection of the derived equivalence classes shows that the class with 201 elements contains all elements that are equivalent to the CURSOR-type. All CURSOR classes occurring in different programs are taken together, as the underlying CURSOR variable is declared in a copybook. When we look at the code we see that the elements in this class are typically used in a relational expression with the CURSOR-type, although in some cases they are both a sub- and supertype of it and therefore inferred to be equivalent.

The next biggest class has 118 elements and represents a type holding some CICS status information. It contains all elements equivalent to the DFHBMEOF-type described in Section 5.4.3, again coming from a copybook.

The class with 39 elements represents the index type for some array type. The elements in this class were typically found using the rule for array index equivalence. It contains the primitive types of variables that were used to access arrays in loops and those that were used for checking array bounds. Here the array variable was declared in a copybook.

The last class we will discuss here is the one with 24 elements. This class represents the so-called RELATION-ID-type and is worth mentioning since it contains a form of pollution that is not solved by subtyping. The spurious type is the so-called MORTGAGE-ID-type which is unrelated to the RELATION-ID-type according to the business logic. The reason that they end up in the same class is that both types are used as

class size	# of classes	percent of total
2	373	63.4%
3	99	16.8%
4	53	9.0%
5	10	1.7%
6	8	1.4%
7	29	4.9%
8	1	0.2%
10	1	0.2%
11	5	0.9%
12	1	0.2%
13	4	0.7%
18	2	0.3%
19	2	0.3%
sum	588	100.0%

(a) program level

class size	# of classes	percent of total
2	135	70.7%
3	22	11.5%
4	5	2.6%
5	4	2.1%
6	9	4.7%
7	6	3.1%
8	2	1.0%
10	1	0.5%
11	1	0.5%
12	1	0.5%
13	1	0.5%
24	1	0.5%
39	1	0.5%
118	1	0.5%
201	1	0.5%
sum	191	100.0%

(b) system level

Figure 5.13: Size and frequency of equivalence classes.

parameter of a "function" that does a sanity check on the number (11-check) *and* returns the corrected number when necessary. In the call both types become subtypes of the input type of that function. After the call, the output is moved back so the output type becomes a subtype of `RELATION-ID` and `MORTGAGE-ID`. Since the input and output type for this function is the same, `RELATION-ID` becomes a subtype `MORTGAGE-ID` and vice versa so they are considered to be equivalent.

We can solve such pollution by deriving an additional cast relation during fact extraction. Whenever a variable of a supertype is assigned to a variable of a subtype, we derive that the supertype is casted into the subtype. Furthermore, we can use data flow analysis to derive what are the input- and what are the output- parameters of a function. This mechanism also allows us to deal with explicit casts as, for example, can occur in C programs.

5.6 Related Work

A principal source of inspiration to us was Lackwit, a tool for understanding C programs by means of type inference [OJ97]. Lackwit performs a type analysis of variables based on their usage. The analysis results are used to find abstract data types, detect abstraction violations, identify unused variables, and to detect certain types of errors. New in our work is not only the significantly different source language, but also the use of subtyping for dealing with pollution, and the use of type inference to classify literals. Another paper discussing type inference for C is by Sniff and Reps [SR96], who use inferred types to generalize C functions to C++ function templates.

Our approach is also related to various tools for the analysis and correction of the year 2000 problem where date *seeds* are tracked through the statements in a program [HP96, NBOS99, KMUO98]. In year 2000 analysis, preventing pollution (called *classification noise* in [KMUO98]) is an important issue. We have not been able to find papers that propose the use of subtyping to do this. This chapter adds a strong empirical basis for using subtyping to reduce pollution.

Recently, two papers appeared which rely on type theory to deal with the year 2000 problem [RFT99, EHM$^+$99]. These papers do not address the problem of pollution, but do contain an interesting algorithm for propagating type information through the elements of aggregate data structures such as arrays or records. Our approach essentially treats each aggregate as a single scalar value. If, however, two entire records are moved, types can also be propagated through the individual fields. Such moves may even cross field boundaries if the two records differ in record layout, or if records are aliased using COBOL's redefine statement. [RFT99, EHM$^+$99] provide an algorithm that finds a minimal splitting of all aggregates such that types can be correctly propagated for the resulting "atoms". In the previous chapter (Chapter 4), we proposed a weaker method using an inference rule called *substructure completion*, which just ensures that type equivalences between structurally equivalent aggregates are propagated to the components. As discussed later, we plan to combine this algorithm with our type inferencing approach to see if we can further improve the accuracy.

Chen *et al.* [CTJ$^+$94] describe a (semi)-automatic approach for COBOL *variable* *classification*. They distinguish a fixed set of categories, such as input/output, constant, local variable etc., and each variable is placed into one or more of these classes. They provide a set of rules to infer this classification automatically, essentially using data flow analysis. Their technique is orthogonal to ours: the types we infer can be used for both local and global variables, for variables that are used for databases access and for those that are not, etc.

Newcomb and Kotik [NK95] describe a method for migrating Cobol to object orientation. Their approach takes all level 01 records as starting point for classes. Records that are structurally equivalent, i.e., matching in record length, field offset, field length, and field picture, but possibly with different names, are called "aliases". According to Newcomb and Kotik, "for complex records consisting of 5-10 or more fields, the likelihood of false positives is relatively small, but for smaller records the probability of false positives is fairly large." [NK95, p. 240]. Our way of type inferencing may help to reduce this risk, as it provides a complementary way of grouping such 01 level records together based on *usage*.

Wegman and Zadeck [WZ91] describe a method to detect whether the value of a variable occurring at a particular point in the program is constant and, if so, what that value is. Merlo *et al.* [MGHDM95] describe an extension of this method that allows detection of all constants that can be the value of a particular variable occurrence. This differs from our approach which finds all constants that can be assigned to *any* variable of a given type. Furthermore, the methods described in both papers take the flow of control into account where as our approach is flow-insensitive (control flow is completely ignored). Consequently, their results are more precise (e.g., we report constants that are used in dead code) but their approach is also more expensive.

Gravley and Lakhotia [GL96] identify enumeration types that are modeled using `#define` preprocessor directives. Their approach is orthogonal to ours since they group constants which are *defined* "in the same context" (i.e., close to each other in the program text) whereas we group constants based on their *usage* in the source code.

Concerning the tool used for implementing type inference, there is second suite of relational algebra tools available from Philips, as described by [FKO98]. An alternative to the use of relational algebra, is to view type inferencing as a graph traversal problem. A graph querying formalism such as GReQL [KW99] can then be used to compute the closures of several relations. A second alternative is to use one of several program analysis frameworks. Of particular interest is BANE, the Berkely ANalysis Engine as described by [AFFS98]. BANE provides constraint specification and resolution components, which can be to experiment with program analyses in which properties of types are expressed as constraints.

5.7 Concluding Remarks

5.7.1 Contributions

In this chapter, we carried out an empirical study into the relations between variables established by Cobol type inference. We argued that such relations are necessary in a

Cobol setting: Cobol programs contain a large number of variable declarations (50% of a program's lines of code consist of variable declarations), but only half of these variables are actually used. Inferred types help to understand how variables are used and how they are related to each other.

The empirical study aimed at finding out how the problem of *pollution* is handled by the use of subtyping. Pollution occurs when a counter-intuitive type equivalence is found for two variables. Since it is impossible to check by hand the hundreds of type equivalences classes found by type inferencing, we devised a suite of numeric measurements directing us to potential pollution spots.

We manually inspected, and explained, the results from these measurements. Of all inferred type equivalence classes, only one contains a clear case of pollution: in Section 5.5.2 we discuss how type casts could help to address this problem.

To conduct our experiments, we developed a tool environment permitting all sorts of experiments. An important new element is the use of relational algebra to do the inference of type conclusions from derived type facts. Moreover, we devised a modular approach to infer types for variables playing a system-wide role. Thanks to this modular approach, system-level type analysis scales up to large systems.

5.7.2 Future Work

Now that we have all machinery for conducting large scale type inferencing experiments in place, and now that we understand which data to collect, we are in a position to apply type inference to more Cobol systems. We intend to do this, and collect statistical data on other case studies as well.

A question of interest is how we can further improve the accuracy of our type inferencing approach by deconstructing aggregates into "atoms" of the appropriate size, following the algorithm of [RFT99, EHM+99]. An important problem to be solved is how to combine this algorithm with *subtyping*, in order to minimize the danger of pollution.

At the moment, we are conducting experiments with new ways of *presenting* type relations.[4] One way is to visualize type relations as graphs. We are integrating such graphs with the Cobol documentation generator covered in [DK99a]. This generator provides an abstract view of Cobol systems, highlighting essential relationships between programs, databases, screens, etc. Types play an important role in this form of documentation, as they help to characterize the interfaces of Cobol modules, or the interplay of variables occurring in the Cobol programs.

[4] This work on the presentation and visualization of type relations in the context of supporting software exploration and re-documentation is described in Chapter 7.

CHAPTER 6

Types and Concept Analysis for Legacy Systems

We combine type inference and concept analysis in order to gain insight into legacy software systems. Type inference for COBOL yields the types for variables and program parameters. These types are used to perform mathematical concept analysis on legacy systems.

We have developed **CONCEPTREFINERY**, *a tool for interactively manipulating concepts. We show how this tool facilitates experiments with concept analysis, and lets reengineers employ their knowedge of the legacy system to refine the results of concept analysis. The work presented in this chapter was published earlier as [KM00].*

6.1 Introduction

Most legacy systems were developed using programming paradigms and languages that lack adequate means for modularization. Consequently, there is little explicit structure for a software engineer to hold on to. This makes effective maintenance or extension of such a system a strenuous task. Furthermore, according to the *Laws of Program Evolution Dynamics*, the structure of a system will decrease by maintenance, unless special care is taken to prevent this [BL76].

Object orientation is advocated as a way to enhance a system's correctness, robustness, extendibility, and reusability, the key factors affecting software quality [Mey97]. Many organizations consider migration to object oriented platforms in order to tackle maintenance problems. However, such migrations are hindered themselves by the lack of modularization in the legacy code.

A software engineer's job can be relieved by tools that support remodularization of legacy systems, for example by making implicit structure explicitly available. Recovering this information is also a necessary first step in the migration of legacy systems to object orientation: identification of candidate objects in a given legacy system.

The use of concept analysis has been proposed as a technique for deriving (and assessing) the modular structure of legacy software [DK99b, LS97, SR97]. This is done by deriving a concept lattice from the code based on data usage by procedures or programs. The *structure* of this lattice reveals a modularization that is (implicitly) available in the code.

For many legacy applications written in COBOL, the data stored and processed represent the core of the system. For that reason, many approaches that support identification of objects in legacy code take the data structures (variables and records) as starting point for candidate classes [CDDF99, FRS94, NK95]. Unfortunately, legacy data structures tend to grow over time, and may contain many unrelated fields at the time of migration. Furthermore, in the case of COBOL, there is an additional disadvantage: since COBOL does not allow *type definitions*, there is no way to recognize, or treat, groups of variables that fulfill a similar role. We can, however, *infer* types for COBOL automatically, based on an analysis of the *use* of variables as is described in Chapters 4 and 5. This results in types for variables, program parameters, database records, literal values, and so on, which can be used during further analysis.

In this chapter, we use the derived type information about the legacy system as input to the concept analysis. This way, the analysis is more precise than when we use variables or records as inputs. The concept analysis is used to find candidate classes in the legacy system. External knowledge of the system can be used to influence the concepts that are calculated through CONCEPTREFINERY, a tool we have implemented for this purpose.

All example analyses described are performed on MORTGAGE, a relation administration subsystem of a large mortgage software system currently in production at various banks. It is a 100,000 LOC COBOL system and uses VSAM files for storing data. The MORTGAGE system is described in more detail in [DK98] and in Chapter 5.

6.2 Type inference for COBOL

COBOL programs consist of a *procedure division*, containing the executable statements, and a *data division*, containing declarations for all variables used. An example containing typical variable declarations is given in Figure 6.1. Line 6 contains a declaration of variable `STREET`. Its physical layout is described as *picture* `X(18)`, which means "a sequence of 18 characters" (characters are indicated by picture code `X`). Line 18 declares the numerical variable `N100` with picture `9(3)`, which is a sequence of three digits (picture code `9`).

The variable `PERSON` in line 3 is a record variable. Its record structure is indicated by level numbers: the full variable has level `01`, and the subfields `INITIALS`, `NAME`, and `STREET`, are at level `03`. Line 12 declares the array `A00-POS`: it is a single character (picture `X(01)`) occurring 40 times, i.e., an array of length 40.

When we want to reason about types of variables, COBOL variable declarations suffer from a number of problems. First of all, it is not possible to separate *type definitions* from *variable declarations*. As a result, whenever two variables have the same record structure, the complete record construction needs to be repeated.[1] Such practices do not only increase the chance of inconsistencies, they also make it harder to understand the program, since a maintainer has to check and compare all record fields in order to decide that two records indeed have the same structure.

In addition, the absence of type definitions makes it difficult to group variables that are intended to represent the same kind of entities. On the one hand, all such variables will share the same physical representation. on the other hand, the converse does not hold: One cannot conclude that whenever two variables share the same byte representation, they must represent the same kind of entity.

Besides these problems with type *definitions*, COBOL only has limited means to indicate the allowed set of values for a variable (i.e., there are no ranges or enumeration types). Moreover, COBOL uses *sections* or *paragraphs* to represent procedures. Neither sections nor paragraphs can have formal parameters, forcing the programmer to use global variables to simulate parameter passing.

To remedy these problems, we have proposed to infer types for COBOL automatically, by analyzing their *use* in the procedure division. In the remainder of this section, we summarize the essentials of COBOL type inferencing: a more complete presentation is given in Chapter 4. First, we describe the *primitive types* that are distinguished. This is followed by a description of the *type relations* that can be derived from the statements in a single COBOL program, and how this approach can be extended to *system-level analysis* leading to inter-program dependencies. Finally, we show how the analysis can be extended to include types for *literals*, discuss the notion of *pollution*, and conclude with an example.

6.2.1 Primitive Types

The following three primitive types are distinguished: (1) *elementary types* such as numeric values or strings; (2) *arrays*; and (3) *records*. Every declared variable gets assigned a unique primitive type. Since variable names qualified with their complete record name must be unique in a COBOL program, these names can be used as labels within a type to ensure uniqueness. We qualify variable names with program or copybook names to obtain uniqueness at the system level. In the remainder we will use T_A to denote the primitive type of variable A.

6.2.2 Type Equivalence

From *expressions* that occur in statements, an *equivalence relation* between primitive types is inferred. We consider three cases: (1) *relational expressions:* such as $v = u$ or $v \leq u$, result in an equivalence between T_v and T_u; (2) *arithmetic expressions:* such as $v + u$ or $v * u$, result in an equivalence between T_v and T_u; (3) *array accesses:* two different accesses to the same array, such as $a[v]$ and $a[u]$, result in an equivalence between T_v and T_u.

[1] In principle the COPY mechanism of COBOL for file inclusion can be used to avoid code duplication here, but in practice there are many cases in which this is not done.

```
DATA DIVISION.
/ variables containing business data.
 01 PERSON.
  03 INITIALS      PIC  X(05).
  03 NAME          PIC  X(27).
  03 STREET        PIC  X(18).
  ...
/ variables containing char array of length 40,
/ as well as several counters.
 01 TAB000.
  03 A00-NAME-PART.
   05 A00-POS      PIC  X(01)   OCCURS 40.
  03 A00-MAX       PIC S9(03)   COMP-3 VALUE 40.
  03 A00-FILLED    PIC S9(03)   COMP-3 VALUE 0.
  ...
/ other counters declared elsewhere.
 01 N000.
  03 N100          PIC S9(03)   COMP-3 VALUE 0.
  03 N200          PIC S9(03)   COMP-3 VALUE 0.

PROCEDURE DIVISION.
/ procedure dealing with initials.
 R210-INITIAL SECTION.
  MOVE INITIALS TO A00-NAME-PART.
  PERFORM R300-COMPOSE-NAME.

/ procedure dealing with last names.
 R230-NAME SECTION.
  MOVE NAME TO A00-NAME-PART.
  PERFORM R300-COMPOSE-NAME.

/ procedure for computing a result based
/ on the value of the A00-NAME-PART.
/ Uses A00-FILLED, A00-MAX, and N100
/ for array indexing.
 R300-COMPOSE-NAME SECTION.
  ...
  PERFORM UNTIL N100 > A00-MAX
   ...
   IF A00-FILLED = N100
    ...
```

Figure 6.1: Excerpt from one of the Cobol programs analyzed.

When we speak of a *type*, we will generally mean an *equivalence class of primitive types*. For presentation purposes, we will also give names to types based on the names of the variables part of the type. For example, the type of a variable with the name `L100-DESCRIPTION` will be called `DESCRIPTION`-type.

6.2.3 Subtyping

From *assignment statements*, a *subtype relation* between primitive types is inferred. Note that the notion of assignment statements corresponds to COBOL statements such as `MOVE`, `COMPUTE`, `MULTIPLY`, etc. From an assignment of the form $v := u$ we infer that T_u is a *subtype* of T_v, i.e., v can hold at least all the values u can hold.

6.2.4 System-Level Analysis

In addition to type relations that are inferred within individual programs, we also infer type relations at the system-wide level: (1) Types of the actual parameters of a program call (listed in the COBOL `USING` clause) are subtypes of the formal parameters (listed in the COBOL `LINKAGE` section). (2) Variables read from or written to the same file or table have equivalent types.

To ensure that a variable that is declared in a copybook gets the same type in all programs that include that copybook, we derive relations that denote the origins of primitive types and the import relation between programs and copybooks. These relations are then used to link types via copybooks.[2]

6.2.5 Literals

An extension of our type inference algorithm involves the analysis of literals that occur in a COBOL program. When a literal value l is assigned to a variable v, we infer that the value l must be a permitted value for the type of v. Likewise, when v and l are compared, value l is considered to be a permitted value for the type of v. Literal analysis infers for each type, a list of values that is permitted for that type. Moreover, if additional analysis indicates that variables in this type are only assigned values from this set of literals, we can infer that the type in question is an *enumeration type*.

6.2.6 Aggregate Structure Identification

When the types of two records are related to each other, types for the fields of those records should be propagated as well. In our first proposal in Chapter 4, we adopted a rule called *substructure completion*, which infers such type relations for record fields whenever the two records are identical (having the same number of fields, each of the same size). Since then, both Eidorff *et al.* [EHM+99] and Ramalingam *et al.* [RFT99] have published an algorithm to split aggregate structures in smaller "atoms", such that

[2] Another (possibly more precise) approach would be to derive a common supertype for all versions that appear in different programs. Our case studies, however, showed no need for such an approach.

types can be propagated through record fields even if the records do not have the same structure.

6.2.7 Pollution

The intuition behind type equivalence is that if the programmer would have used a typed language, he or she would have chosen to give a single type to two different COBOL variables whose types are inferred to be equivalent. We speak of *type pollution* if an equivalence is inferred which is in conflict with this intuition.

Typical situations in which pollution occurs include the use of a single variable for different purposes in disjunct program slices; simulation of a formal parameter using a global variable to which a range of different variables are assigned; and the use of a `PRINT-LINE` string variable for collecting output from various variables.

The need to avoid pollution is the reason to introduce *subtyping* for assignments, rather than just type equivalences. In Chapter 5, we have described a range of experimental data showing the effectiveness of subtyping for dealing with pollution.

6.2.8 Example

Figure 6.1 contains a COBOL fragment illustrating various aspects of type inferencing. It starts with a data division containing the declaration of variables. The second part is a procedure division containing statements from which type relations are inferred.

In line 40, variable `A00-FILLED` is compared to `N100`, which in line 38 is compared to `A00-MAX`. This results in an equivalence class between the primitive types of these three variables. Observe that these three variables are also declared with the same picture (in lines 13, 14, and 18).

In line 29, we infer from the assignment that the type of `NAME` is a *subtype* of the type of `NAME-PART`. From line 24, we infer that `INITIALS` is a subtype of of `NAME-PART` as well, thus making `NAME-PART` the common supertype of the other two. Here the three variables are declared with different pictures, namely strings of different lengths. In fact, `NAME-PART` is a global variable simulating a formal parameter for the `R300-COMPOSE-NAME` (COBOL does not support the declaration of parameters for procedures). Subtyping takes care that the different sorts of actual parameters used still have different types.

6.3 Concept Analysis

Concept analysis is a mathematical technique that provides a way to identify groupings of *items* that have common *features* [GW99]. It starts with a *context*: a binary table (relation) indicating the *features* of a given set of *items*. From that table, the analysis builds up so-called *concepts* which are maximal sets of items sharing certain features. The relations between *all* possible concepts in a binary relation can be given using a concise lattice representation: the *concept lattice*.

Recently, the use of concept analysis has been proposed as a technique for analyzing legacy systems [Sne98b]. One of the main applications in this context is deriving (and assessing) the modular structure of legacy software [DK99b, LS97, SR97, ST98]. This is done by deriving a concept lattice from the code based on data usage by procedures or programs. The *structure* of this lattice reveals a modularization that is (implicitly) available in the code. In [DK99b], we used concept analysis to find groups of record fields that are related in the application domain, and compared it with cluster analysis.

In the remainder of this section we will explain concept analysis in more detail.

6.3.1 Basic Notions

We start out with a set $\mathcal{M}$ of *items*, a set $\mathcal{F}$ of *features*,[3] and a *binary relation* (table) $T \subseteq \mathcal{M} \times \mathcal{F}$ indicating the features possessed by each item. The three tuple $(T, \mathcal{M}, \mathcal{F})$ is called the *context* of the concept analysis. In Figure 6.2 the items are the field names, and the features are usage in a given program. We will use this table as example context to explain the analysis.

For a set of items $I \subseteq \mathcal{M}$, we can identify the *common features*, written $\sigma(I)$, via:

$$\sigma(I) = \{f \in \mathcal{F} \mid \forall i \in I : (i,f) \in T\}$$

For example, $\sigma(\{\texttt{ZIPCD}, \texttt{STREET}\}) = \{P_4\}$.

Likewise, we define for $F \subseteq \mathcal{F}$ the set of *common items*, written $\tau(F)$, as:

$$\tau(F) = \{i \in \mathcal{M} \mid \forall f \in F : (i,f) \in T\}$$

For example, $\tau(\{P_3, P_4\}) = \{\texttt{STREET}\}$.

A *concept* is a pair (I, F) of items and features such that $F = \sigma(I)$ and $I = \tau(F)$. In other words, a concept is a maximal collection of items sharing common features. In our example, the pair $(\{\texttt{PREFIX}, \texttt{INITIAL}, \texttt{TITLE}, \texttt{NAME}\}, \{P_1\})$ is the concept of those items having feature P_1, i.e., the fields used in program P_1.

[3] The literature generally uses *object* for *item*, and *attribute* for *feature*. In order to avoid confusion with the objects and attributes from object orientation we have changed these names into items and features.

Items \ Features	P_1	P_2	P_3	P_4
NAME	×			
TITLE	×			
INITIAL	×			
PREFIX	×			
CITY		×		×
STREET			×	×
NUMBER				×
NUMBER-EXT				×
ZIPCD				×

Figure 6.2: The list of items and their features.

	items	features
c0	zipcd number-ext number street city prefix initial title name	
c1	zipcd number-ext number street city	p4
c2	street	p4 p3
c3	city	p4 p2
c4	prefix initial title name	p1
c5		p4 p3 p2 p1

Figure 6.3: All concepts identified for Figure 6.2.

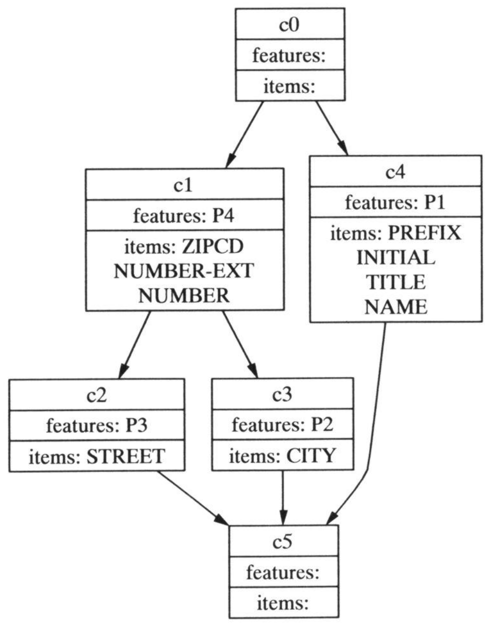


Figure 6.4: Lattice for the concepts of Figure 6.3.

All concepts that can be identified from Figure 6.2 are summarized in Figure 6.3. The items of a concept are called its *extent*, and the features its *intent*.

The concepts of a given table are partially ordered via:

$$(I_1, F_1) \leq (I_2, F_2) \quad \Leftrightarrow \quad (I_1 \subseteq I_2 \quad \Leftrightarrow \quad F_2 \subseteq F_1)$$

For example, for the concepts in Figure 6.3, we see that $\bot = c5 \leq c3 \leq c1 \leq c0 = \top$. This partial order allows us to organize all concepts in a *concept lattice*, with *meet* $\wedge$ and *join* $\vee$ defined as

$$\begin{aligned}(I_1, F_1) \wedge (I_2, F_2) &= (I_1 \cap I_2, \sigma(I_1 \cap I_2)\\(I_1, F_1) \vee (I_2, F_2) &= (\tau(F_1 \cap F_2), F_1 \cap F_2)\end{aligned}$$

The visualization of the concept lattice shows all concepts, as well as the relationships between them. For our example, the lattice is shown in Figure 6.4.

In such visualizations, the nodes only show the "new" items and features per concept. More formally, a node is labeled with an item i if that node is the *smallest* concept with i in its extent, and it is labeled with a feature f if it is the *largest* concept with f in its intent.

For a thorough study of the foundations of concept analysis we refer the reader to [GW99].

6.4 Combine Types and Concepts

In [DK99b] concept analysis was used to find structure in a legacy system. The variables of a Cobol system were considered items, the programs features, and the "variable used in program" property as a relation. Figure 6.2 is an example of such a relation, and Figure 6.4 show the corresponding lattice. This lattice can be seen as a candidate object oriented design of the legacy system. The concepts are individual classes and related concepts can be seen as subclasses or class associations.

The identification of variables in different programs was performed by comparing variable names, and variable declarations. If two variables shared a particular substring they were considered equal. This works well for systems that employ a coding standard which forces similar names for similar variables but fails horribly for systems where variable names are less structured. In this chapter this problem is solved by taking the *types* (as described in Section 6.2) of these variables, and relating them to programs in various ways.

6.4.1 Data for Concept Analysis

Before describing the concept experiments performed, first the relations derived from the legacy source will be explained. The four extracted relations are varUsage, typeEquiv, transSubtypeOf and formalParam. varUsage is the relation between a program and the variables that are used in that program. typeEquiv is the relation between a type name (the name of a type equivalence class) and a variable that is of this type. transSubtypeOf is the relation between a type and the transitive closure of all its supertypes, i.e. between two types where the second is in the transitive closure of all the supertypes of the first. formalParam is the relation between a program and the types of its formal parameters. An overview of these relations is given in Figure 6.5.

In the remainder of this section the set of all programs, variables, and types in a system will be denoted P, V, and T, respectively.

6.4.2 Experiments Performed

Type Usage

The first experiment performed is exactly the experiment performed in [DK99b], as described earlier. The type usage per program is taken as the context relation, instead of *variable* usage. This results in a lattice where the programs that use exactly the same set of types will end up in the same concept, programs that use less types will end up in a concept below, and programs that use more types will end up in a concept above that concept.

In order to arrive at the type usage concept lattice the varUsage table is taken as a starting point. For each variable, its type is selected from typeEquiv such that the result is a set of relations $\{(p,t) \in P \times T | (p,v) \in \text{varUsage}, (t,v) \in \text{typeEquiv}\}$. Then the types are considered items, and the programs features and the concept analysis is performed. For the example MORTGAGE system, the resulting concept lattice is shown in Figure 6.6. The list of items and features is not shown for (obvious) lack of space.

Filtering

Figure 6.6 may not be as insightful as we might hope. A way to decrease the complexity of this picture is by filtering out data before performing the concept analysis. A selection of *relevant* programs from all programs in a COBOL system can be made as described in [DK98]. COBOL systems typically contain a number of programs that implement low-level utilities such as file I/O, error handling and memory management. These programs can in general be left out of the analysis, particularly when we are only interested in the general structure of the system.

Relation name	Name of relation element	
varUsage	program	variable
typeEquiv	type	variable
transSubtypeOf	sub	super
formalParam	program	type

Figure 6.5: Derived and inferred relations.

Filtering out insignificant variables is also possible. Typically, certain records in a Cobol system contain all data that has to do with customers (and therefor is probably relevant) while other records may only be used as temporary storage.

Suppose a list of relevant programs is selected and only the data that originated from a certain set of records is deemed interesting. The first step, filtering out the uninteresting programs, is easy. All tuples from varUsage that have an irrelevant program as their program element are simply ignored. Suppose P_{rel} with $P_{rel} \subseteq P$ is the set of relevant programs which is derived in some way. Then all types that are related to the interesting variables need to be determined. Suppose V_{rel} with $V_{rel} \subseteq V$ is the set of all relevant variables. From the relation typeEquiv all types that are related to a relevant variable are selected. If T_{rel} with $T_{rel} \subseteq T$ is the set of all relevant types: $\{t \in T | (t, v) \in \text{typeEquiv}, v \in V_{rel}\}$ Then the type equivalent variables that are used in the selected relevant programs are selected: $\{(v, p) \in V_{rel} \times P_{rel} | (v', p) \in \text{varUsage}, (t, v') \in \text{typeEquiv}, t \in T_{rel}\}$

The result of the experiments with filtered data are much more comprehensible than those without filtering, basically because there are less concepts to try to understand. Figure 6.7 shows the concept lattice for the same system as in Figure 6.6, but with irrelevant programs filtered out according to [DK98]. The relevant data are the fields of the two records describing the persistent data in the system.

The lattice in Figure 6.7 contains some unexpected combinations. Concept 7 for instance, contains items that have to do with locations and addresses, but also a birth date. Close inspections reveals that this is not a case of type pollution, but these variables are really used in both program 31(c) (from concept 1) and program 22 (from concept 6). A possible explanation could be that these programs send birthday cards.

It is important to have some way to validate these lattices externally, to perfect the filter set. For our example system, one program implements a utility routine through which a lot of variables are passed, causing one type to contain a remarkable large number of variables. When we filtered out that program, the resulting lattice was much more intuitive.

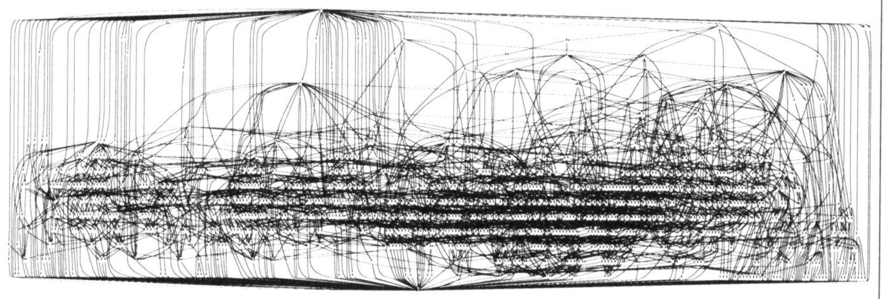


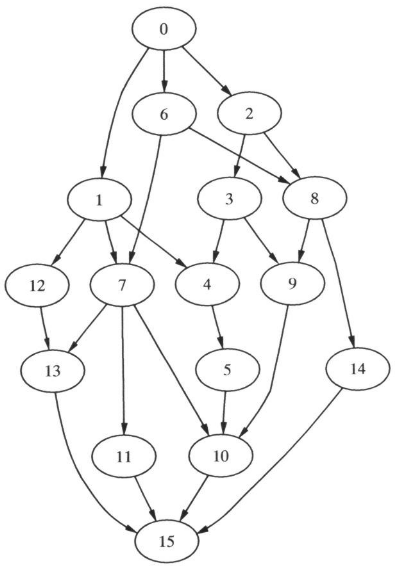
Figure 6.6: Types as items, programs using given types as features.

Parameter Types

Experiments have been performed on another concept analysis context; the context that has programs as items and the types of their formal parameter as features. When concept analysis is performed on this data set, all programs that share exactly the same set of parameter-types end up in the same concept. If two programs share some parameter-types, but not all, the shared parameter types will end up in the same concept. These will then form an excellent basis for developing an object oriented view on the system, as the shared types can be seen as the attributes of a class sharing programs as methods.

In its simplest version the items and features for these concepts are computed by just taking formalParam and ignoring the subtype relationship.

As was described in Section 6.2, the relation between actual parameter types and formal parameter types is inferred as a subtype relation. If the subtype relationship is ignored, then variables can only be identified as having the same type in different programs, when they are "passed" through a copybook. That is, if a variable is included in two different programs from the same copybook, it is considered type equivalent in



	items	features
0		
1	1.firstnm[] 1.telb-area 1.telb-nr 1.telp-area 1.telp-nr 1.contact-prs2 1.contact-prs1 1.sex 1.plbirth 1.cvlcd 1.openg	31c 31
2		10c
3		10
4	4.city-bank 4.pay 4.acctnr-bank 4.pay-type 4.paysoc-name 4.acctnr 1.city-bank 1.acctnr-bank 1.pay-type 1.paysoc-name 1.acctnr	
5	4.mortseqnr 4.mortgnr	09 09c
6	1.variable[]	22
7	1.region[] 1.street[] 1.addr-type[] 1.zipcd[] 1.countrycd[] 1.house-ext[] 1.citycd[] 1.birthdt 1.type 1.count	
8		
9	1.mod-dat	
10	4.relnr 1.p-relnr	
11	1.city[] 1.house[]	89c 89
12	1.prefix 1.initl 1.titlcd	36c 36
13	1.fixed 1.name	
14	1.startdt[]	12c
15		

Figure 6.7: Concepts involving relevant programs.

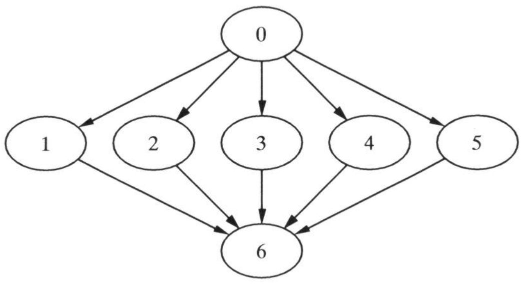
the two programs. Obviously, this is not the intuition we have when looking at formal parameters, where we would like to know how the types used in the calling program propagate to the called program. Therefor, subtyping *is* considered as type equivalence when looking at parameter types.

The context for parameter type usage per program while considering supertypes as equivalent is derived as follows: $\{(p,v) \in P \times V | (p,t) \in \mathsf{formalParam}, (((t',t) \in \mathsf{transSubtypeOf} \wedge (t',v) \in \mathsf{typeEquiv}) \vee (t,v) \in \mathsf{typeEquiv})\}$.

As described in the previous section, data may be filtered on either relevant programs or relevant data elements. In that case the context is arrived at as follows: $\{(p,v) \in P_{rel} \times V_{rel} | (p,t_1) \in \mathsf{formalParam}, (((t',t) \in \mathsf{transSubtypeOf} \wedge (t',v) \in \mathsf{typeEquiv}) \vee (t,v) \in \mathsf{typeEquiv})\}$ for some externally determined value of P_{rel} and V_{rel}.

An example of a concept lattice showing program as items and the types they use as formal parameters as features (when supertypes are considered type equivalent) filtered for the same set of relevant variables as Figure 6.7 is shown in Figure 6.8.

In this lattice, concept 3 is remarkable, because it contains by far the most programs. This turns out to be caused by the fact that these programs all use "record" as input parameter. Inspection of the source reveals that "record" is a rather large record, and that only some fields of this record are actually used in the programs. It is subject of future work to look at these types of parameters in more detail.



	items	features
0		
1	41	city[]
2	40	street[]
3	36c 35030u 31c 10c 05010r 01410u 01330u 01230u	record
4	36	fixed
5	31 10 09	p-relnr
6		

Figure 6.8: Programs as items, parameters as features.

6.5 Refinement of Concepts

When concept analysis is used for analyzing software systems, there will be a point where a user might want to modify an automatically derived concept lattice. For example, consider the applications of concept analysis to remodularization of legacy systems. A maintainer that performs such a task is likely to have knowledge of the system that is being analyzed. Based on that knowledge, he or she might have certain ideas to improve the modularization indicated by the derived lattice by combining or ignoring certain parts of that lattice.

To facilitate the validation of such ideas, we have developed ConceptRefinery, a tool which allows one to manipulate parts of a concept lattice while maintaining its consistency. ConceptRefinery defines a set of generic structure modifying operations on concept lattices, so its use is not only restricted to the application domain of remodularization or reverse engineering. Figure 6.9 shows the application of ConceptRefinery on the data of Figure 6.2.

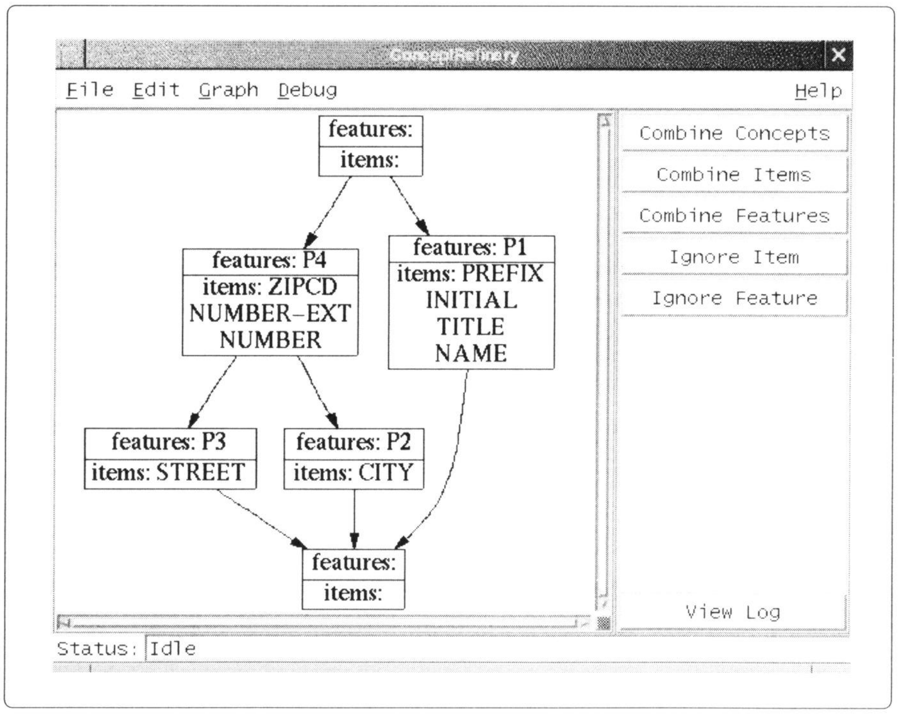


Figure 6.9: Screendump of ConceptRefinery.

6.5.1 Operations on Concept Lattices

We allow three kinds of operations on concept lattices. The first is to combine certain items or certain features. When we consider the context of the concept analysis, these operations amount to combining certain rows or columns in the table and recomputing the lattice.

The second operation is to ignore certain items or features. When we consider the analysis context, these operations amount to removing certain rows or columns and recomputing the lattice.

The third operation is combining two concepts. This operation has the following rationale: when we consider concepts as class candidates for an object-oriented (re-)design of a system, the standard concept lattice gives us classes where all methods in a class operate on all data in that class. This is a situation that rarely occurs in a real world OO-design and would result a large number of small classes that have a lot of dependencies with other classes. The combination of two concepts allows us to escape from this situation.

On the table underlying the lattice the combination of two concepts can be computed by adding all features of the first concept to the items of the second and vice versa.

6.5.2 Relation With Source

When a concept lattice that was previously derived from a legacy system is manipulated, the relation between that lattice and the code will be weakened:

- Whenever features, items or concepts are combined, the resulting lattice will represent an abstraction of the source system.

- Whenever features or items are ignored, the resulting lattice will represent a part of the source system.

The choice to allow such a weakening of this relation is motivated by the fact that we would rather be able to understand only part of a system than not being able to understand the complete system at all. However, in order for ConceptRefinery to be useful in a real-world maintenance situation, we have to take special care to allow a maintainer to relate the resulting lattice with the one derived directly from the legacy code. This is done by maintaining a concise log of modifications.

6.6 Implementation

We have developed a prototype toolset to perform concept analysis experiments. An overview of this toolset is shown in Figure 6.10. The toolset separates source code analysis, computation and presentation. Such a three phase approach makes it easier to adapt to different source languages, to insert specific filters, or to use other ways of presenting the concepts found [DK98, DM98].

In the first phase, a collection of *facts* is derived from the COBOL sources. For that purpose, we use a parser generated from the COBOL grammar discussed in [BSV97a]. The parser produces abstract syntax trees that are processed using a Java package which implements the visitor design pattern. The fact extractor is a refinement of this visitor which emits facts at every node of interest (for example, assignments, relational expressions, etc.).

From these facts, we infer types for the variables that are used in the COBOL system. This step uses the COBOL type inferencing tools presented in Chapter 5. The derived and inferred facts are stored in a MySQL relational database [YRK99].

In the next phase, a selection of the derived types and facts is made. Such a selection is an SQL queries that results in a table describing items and their features. A number of interesting selections were described in Section 6.4. The results of these selections are stored in a repository. Currently, this is just a file on disk.

In the final phase, the contents of the repository are fed into a concept analysis tool, yielding a concept lattice. We make use of the concept analysis tool that was developed by C. Lindig from the University of Braunschweig.[4] The concept lattice can be visualized using a tool that converts it to input for dot [GKNV93], a system for visualizing graphs. The lattices in Figures 6.4, 6.8, 6.6 and 6.7 were produced this way.

Furthermore, the lattice can be manipulated using CONCEPTREFINERY. This tool allows a user to select items, features or concepts and perform operations on that selection. These operations result in updates of the repository. We distinguish the following manipulations and describe the actions that are carried out on the repository: (1) *combining items or features* is done by merging corresponding columns or rows in the repository; (2) *ignoring items or features* is done by removing corresponding columns or rows in the repository; (3) *combining concepts* is done by adding all features of the

[4] The tool "`concepts`" is available from http://www.cs.tu-bs.de/softech/people/lindig/.

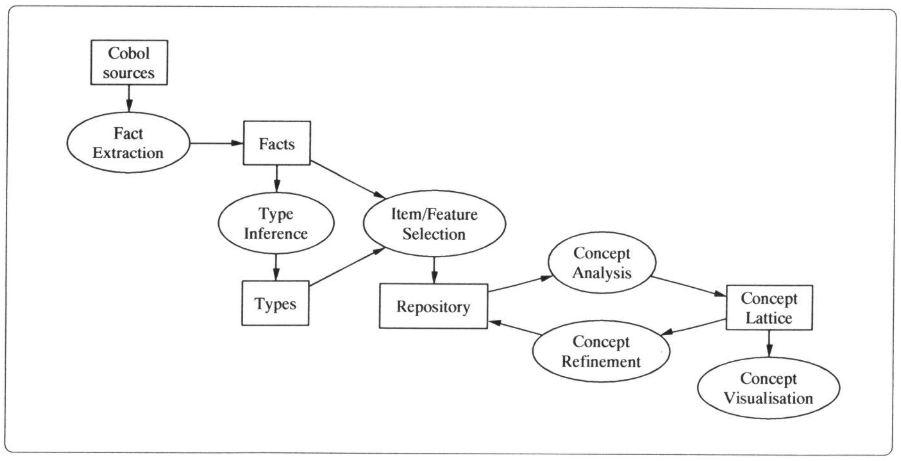


Figure 6.10: Overview of the toolset.

first concept to the items of the second and vice versa. The user interface of ConceptRefinery is shown in Figure 6.9. On the left hand side a visualization of the concept lattice is given. The items, features or concepts that need to be modified can be selected in this lattice. The right hand side shows all available operations. ConceptRefinery is implemented in Tcl/Tk [Ous94] and Tcldot: an extension for Tcl/Tk that incorporates the directed graph facilities of dot into Tcl/Tk and provides a set of commands to control those facilities.

6.7 Related Work

Several methods have been described for modularizing legacy systems. A typical approach is to identify procedures and global variables in the legacy, and to group these together based on attributes such as use of the same global variable, having the same input parameter types, returning the same output type, etc. [CCM96, LW90, OT93, Sch91]. A unifying framework discussing such *subsystem classification techniques* is provided by Lakhotia [Lak97].

Many of these approaches rely on features such as scope rules, return types, and parameter passing, available in languages like Pascal, C, or Fortran. Many data-intensive business programs, however, are written in languages like Cobol that do not have these features. As a consequence, these class extraction approaches have not been applied successfully to Cobol systems [CDDF99]. Other class extraction techniques have been developed specifically with languages like Cobol in mind. They take specific characteristics into account, such as the structure of data definitions, or the close connection with databases [CDDF99, FRS94, NK95]. The interested reader is referred to [DK99b] for more related work on object identification.

Concept analysis has been proposed as a technique for analyzing legacy systems. Snelting [Sne98b, Sne00] provides an overview of various applications. Applications in this context include reengineering of software configurations [Sne96], deriving and assessing the modular structure of legacy software [LS97, SR97], object identification [DK99b], and reengineering class hierarchies [ST98].

The extract-query-view approach adopted in our implementation is also used by several other program understanding and architecture extraction tools, such as Ciao [CFKW95], Rigi [WTMS95], PBS [SCHC99], and Dali [KC99].

New in our work is the addition of the combination of concept analysis and type inferencing to the suite of analysis techniques used by such tools. Our own work on type inferencing started with Chapter 4, where we present the basic theory for Cobol type inferencing, and propose the use of subtyping to deal with pollution. In Chapter 5, we cover the implementation using Tarski relational algebra, as well as an assessment of the benefits of subtyping for dealing with pollution. Type-based analysis of Cobol, for the purpose of year 2000 analysis, is presented by [EHM$^+$99, RFT99]: both provide a type inference algorithm that splits aggregate structures into smaller units based on assignments between records that cross field boundaries. The interested reader is referred to Chapters 4 and 5 for more pointers to related work on type inferencing.

6.8 Concluding Remarks

In this chapter we have shown that the combination of facts derived from legacy source code, together with types inferenced from those facts, forms a solid base for performing concept analysis to discover structure in legacy systems. This extends and combines our previous work on type inferencing for legacy systems and object identification using concept analysis. We implemented a prototype toolset for performing experiments. From these experiments, we can conclude that the combination of type inference and concept analysis provides more precise results than our previous concept analyses which did not involve types.

The combinations discussed in this chapter are the following concept analysis contexts:

1. type usage per program
2. types of parameters per program

The latter analysis appears to be particularly suitable as a starting point for an object oriented redesign of a legacy system.

When performing concept analysis to gain understanding of a legacy system, it proves very helpful if the reengineer is able to manipulate the calculated concepts to match them with his knowledge of the system, or to remove parts he know to be irrelevant. We have implemented ConceptRefinery, a tool that allows a software engineer to consistently perform this kind of modifications while maintaining a relation with both the original calculated concepts, and the legacy source code.

6.8.1 Future Work

We would like to extend ConceptRefinery to propose a grouping of concepts to the human engineer to consider when refining the lattice. To this end, we want to experiment with applying cluster analysis algorithms to the concept lattice.

We have discussed two particular concept analysis contexts in this chapter. We would like to see whether we could use the results of one of these concept analyses to improve the results of the other. I.e. to take the concept found by looking at the parameter types of programs and somehow use those to mark relevant and irrelevant concepts from the variable usage analysis.

Acknowledgments

The many pleasant discussions we had about this chapter with Arie van Deursen are greatly appreciated. We thank Joost Visser for his comments on earlier versions of this chapter.

CHAPTER 7

Exploring Legacy Systems Using Types

We show how hypertext-based program understanding tools can achieve new levels of abstraction by using inferred type information for cases where the subject software system is written in a weakly typed language. We propose TYPEEXPLORER, *a tool for browsing* COBOL *legacy systems based on these types.*

The chapter addresses (1) how types, an invented abstraction, can be presented meaningfully to software re-engineers; (2) the implementation techniques used to construct TYPEEXPLORER; *and (3) the use of* TYPEEXPLORER *for understanding legacy systems, at the level of individual statements as well as at the level of the software architecture which is illustrated by using* TYPEEXPLORER *to browse an industrial* COBOL *system of 100,000 lines of code. The work presented in this chapter was published earlier as [DM00].*

7.1 Introduction

Software immigrants, employees that are added to an existing software project in order to conduct maintenance or development, are faced with the difficult task of understanding an existing software system [SH98]. Even the original developers of a system generally have a hard time understanding their own code as time between development and maintenance goes by. As a consequence, maintenance tasks become difficult, expensive, and error prone.

To reduce these problems, much research is being invested in the development of tools to assist in program understanding. One line of research focuses on the use of hypertext for program comprehension purposes [Bro91, DK99a, BSL98, RV99, SCHC99]. Within a hypertext, various layers of abstraction can be integrated, ranging from the system's architecture to the individual statements in the source code. The maintenance engineer can navigate easily between these, using both top-down and bottom-up comprehension strategies, as well as the "opportunistic" combination of these [MV95, RV99].

Such a hypertext can be seen as a (special form of) system documentation. Part of it will be hand-written, especially those sections dealing with domain-specific issues or the system's requirements. However, documentation at the more technical level should be generated whenever possible, in order to keep it up to date and consistent with the sources at all times.

The fundamental problem with documentation generation (and in fact, the key challenge of reverse engineering) is to arrive at non-trivial levels of abstraction, going beyond just cross referencing information and source code browsing. Our research aims at achieving such a level of abstraction by looking at the *types* that are used in a software system.

For typed languages, such as Java, C, and Pascal, using types for program comprehension is relatively straightforward: types are explicit, and can help to determine interfaces, function signatures, permitted values for certain variables, etc. Many of the existing software systems, however, are written in older languages with very weak type systems. In particular COBOL, the language in which at least 30% of the world's software is written, does not offer the possibility of type definitions. The question we ask ourselves is whether types nevertheless can help in understanding such COBOL systems.

The solution we propose is to *infer* types for COBOL automatically, based on an analysis of the *use* of variables as described in Chapter 4. This results in types for variables, program parameters, database records, literal values, and so on, which can be used to understand the relationships between programs, copybooks, databases, screens, and so on.

In earlier work, we presented an algorithm and toolset for determining types in COBOL systems (presented in Chapters 4 and 5). The current chapter addresses the problems involved in integrating inferred types into hypertext-based program understanding tools. In particular, we will be concerned with the following three questions:

Presentation: Types are an abstraction not directly present in the (legacy) system — types do not exist in the code, but must be inferred first. How do we present this abstraction in such a way that it provides an understandable, meaningful and useful view on a legacy system?

Implementation: How do we implement tools to obtain this presentation?

Use: What maintenance or program understanding questions can be answered using such a presentation, not only at the individual module level, but also at the architectural level?

We will explain how we dealt with these issues while constructing TYPEEXPLORER, a tool for exploring COBOL systems using types. In Section 7.2 we give an overview of related work. Section 7.3 discusses the theory of type inferencing for COBOL. The design of the hypertext structure used by TypeExplorer is covered in Section 7.4. The techniques that were used for implementation are described in Section 7.5. We discuss the usefulness of TypeExplorer for various program understanding tasks and describe its application in a 100,000 lines of code Cobol case study in Section 7.6. Finally, we summarize our contributions, and list possibilities for future work in Section 7.7 .

7.2 Related Work

A growing body of literature on web-based program comprehension exists [Bro91, DK99a, BSL98, RV99, SCHC99, DCG+99]. Of these, Brown discusses a tool that automatically creates links between program analysis data and hypertext documentation [Bro91]. CHIME is a generator of tools that automatically insert certain links in source code elements [DCG+99]. PAS is a system that can be used to incrementally add *partitioned annotations of software* [RV99]. *Documentu* derives documentation from Cobol sources based on special comment tags added by the programmer [BSL98].

DocGen is a tool for generating hyperlinked visual and textual documentation from Cobol and batch job sources [DK99a]. Distinguishing characteristics of DocGen include extraction based on *island grammars* rather than full parsing, emphasis on industrial application[1], and integration of various abstraction layers, ranging from source code up to system architecture. We will see later how the type information derived by TypeExplorer can be integrated with documentation that was generated by DocGen.

Many architecture extraction tools (such as Rigi [WTMS95], PBS [SCHC99], Dali [KC99], and also DocGen and TypeExplorer) adopt the extract-query-view approach, extracting facts from sources, querying a database filled with facts, and presenting these facts in various ways, for example using hypertext. PBS, which has been applied mostly to analyze C systems such as Linux, uses Tarski relational algebra for querying, which is also used in the implementation of TypeExplorer. Dali emphasizes the need for an open tool set, in which many different tools can be plugged in, when necessary. New in our work is the addition of type inferencing to the suite of analysis techniques used by such tools.

Closest in aims to the integration of type analysis and program understanding is Lackwit [OJ97], a tool for analyzing C programs using type inferencing. Lackwit allows one to ask queries like "Which functions could directly access the representation of component X of variable Y?" Other work based on type inferencing includes "physical type checking of C", which is a stronger form of type checking for type casts involving pointers to structures [CR99], and the analysis of Fortran programs in order to find new type signatures for subroutines [WP99]. Type-based analysis of Cobol, for the purpose of year 2000 analysis, is presented by [EHM+99, RFT99]: both provide a type inference algorithm that splits aggregate structures into smaller units based on assignments between records that cross field boundaries.

[1] Documentation generation services using DocGen are available via the *Software Improvement Group*, http://www.software-improvers.com.

Our own work on type inferencing started with Chapter 4, where we present the basic theory for Cobol type inferencing. In Chapter 5, we described an implementation using Tarski relational algebra. Moreover, we carried out a detailed assessment of the benefits of using subtyping to deal with the problem of *pollution* (inferring too many type equivalences). In this chapter, we do not extend the theory of type inferencing: instead we explain how inferred types can be presented using hyper-text, and used to understand Cobol systems at various levels of abstraction.

More references to related work can be found in [DK99a] (documentation generation) and Chapters 4 and 5 (type inference for Cobol).

7.3 Type Inference for Cobol

Cobol programs consist of a *procedure division*, containing the executable statements, and a *data division*, containing declarations for all variables used.

From the perspective of types, Cobol variable declarations suffer from a number of problems. First of all, it is not possible to separate type definitions from variable declarations. Consequently, when two variables for the same record structure are needed, the full record construction needs to be repeated.[2] This not only increases the chances of inconsistencies, it also makes it harder to understand the program, as the maintainer has to check and compare all record fields in order to decide that two records indeed have the same structure.

Furthermore, the absence of type definitions makes it difficult to group variables that are intended to represent the same kind of entities. Clearly, all such variables will share the same physical representation. Unfortunately, the converse does not hold: One cannot conclude that whenever two variables share the same byte representation, they must represent the same kind of entity.

Besides these problems regarding type *definitions*, Cobol only has limited means to indicate the allowed set of values for a variable (i.e., there are no ranges or enumeration types). Moreover, Cobol uses *sections* or *paragraphs* to represent procedures. Neither sections nor paragraphs can have formal parameters, forcing the programmer to use global variables for parameter passing.

In Chapter 4, we propose a method to infer types for Cobol to remedy these problems. This method automatically infers types for Cobol variables by analyzing the *use* of these variables in the procedure division. The remainder of this section summarizes the essentials of Cobol type inferencing.

7.3.1 Primitive Types

We distinguish three primitive types: (1) elementary types such as numeric values or strings; (2) arrays; and (3) records. Initially every declared variable gets a unique primitive type. Since (qualified) variable names must be unique in a Cobol program, they can be used as labels within a type to ensure uniqueness. We qualify these names with

[2] In principle the COPY mechanism of Cobol for file inclusion can be used to avoid code duplication here, but in practice there are many cases in which this is not done.

program or copybook names to obtain uniqueness at the system level. We use T_A to denote the primitive type of variable A.

7.3.2 Type Equivalence

From *expressions* occurring in statements, an *equivalence relation* between primitive types is inferred. We distinguish three cases:

1. *Relational expressions* such as $v = u$ or $v \leq u$ result in an equivalence between T_v and T_u.

2. *Arithmetic expressions* such as $v + u$ or $v * u$ result in an equivalence between T_v and T_u.

3. *Array accesses* to the same array, such as $a[v]$ and $a[u]$ result in an equivalence between T_v and T_u.

We will generally speak of a *type*, meaning an *equivalence class of primitive types.* We will give names to types based on the names of the variables that are of that type. For example, the type of a variable with the name `L100-DESCRIPTION` will be called `DESCRIPTION`-type.

7.3.3 Subtyping

From *assignment statements* a *subtype relation* between primitive types is inferred. From the assignment $v := u$ we conclude that T_u is *subtype* of T_v, i.e., v can hold at least all the values u can hold.

7.3.4 Union types

From COBOL *redefine clauses*, a *union type* relation between primitive types is inferred. When an entry v in the data division redefines an entry u, we conclude that T_v and T_u are part of the same *union type.*

7.3.5 System-Level Analysis

The type relations described before are derived at the program level. We also derive a number of type relations at the system-wide level: (1) *program parameters:* the types of the actual parameters of a program call (listed in the COBOL `USING` clause) are *subtypes* of the formal parameters (listed in the COBOL `LINKAGE` section), (2) *file/table access:* variables read from or written to the same file or table have *equivalent* types, and (3) *copybooks:* a variable which is declared in a copybook gets the same type in all the programs that include this copybook.

7.3.6 Literals

Our type inference algorithm can easily be extended with analysis of literals in a COBOL program. Whenever a literal value l is assigned to, or compared with a variable v, we infer that l is a *permitted value* for the type of v. If additional analysis indicates that variables in this type are only assigned values from this set of literals, we can infer that the type in question is an *enumeration type.*

7.3.7 Aggregate Structure Identification

Whenever the types of two records are related to each other, types for the individual fields should be propagated as well. In Chapter 4, we adopted a rule called *substructure completion*, which infers such type relations for record fields whenever the two record structures are identical (having the same number of fields, each of the same size). Since then, both Eidorff *et al.* [EHM$^+$99] and Ramalingam *et al.* [RFT99] have published an algorithm which splits aggregate structures in smaller "atoms", such that types can be propagated through record fields even if the records do not have the same structure.

7.3.8 Pollution

We speak of *type pollution* when the types of two variables are inferred to be equivalent but would have been given different types in case a typed language was used. Typical situations in which pollution occurs include the use of a single variable for different purposes in different program slices; the use of a global variable for parameter passing; and the use of a `PRINT-LINE` string variable for collecting values from various variables.

Inference of *subtypes* for assignments, rather than just type equivalences was introduced to avoid pollution. In Chapter 5, we describe a range of experimental data showing the effectiveness of subtyping for dealing with pollution.

7.4 Presenting Types in Hypertext

This section describes how types can be presented in a hypertext to support program understanding. We cover the challenges that need to be addressed, as well as the solutions we adopted in TYPEEXPLORER.

7.4.1 Challenges

Inventing a name for a type

Recall from Section 7.3 that a *type* is an equivalence class of *primitive types*, and that each primitive type directly corresponds to a variable declaration. In TYPEEXPLORER, we need to invent names for these equivalence classes. One way is to pick an arbitrary element, and make that the name of the type.

An alternative is to try to distill meaningful names from the variable names involved, by determining the *words* occurring in them. Such words can be found by splitting the variable names based on special characters ('-', '_', etc.) or lexical properties (e.g., caseChange). The actual splitting should be a parameter of the analysis since it is influenced by the coding style that is used in a system. Candidate names of a given type can then be based on the frequency of words that occur in names of variable of that type. Since we want these names to be as descriptive as possible, one also needs to consider all combinations of words that occur in variable names. As an example, for the `A00-NAME-PART` variable, we not only want to see the words `NAME` and `PART`, but also the word `NAME-PART`.

Duality of subtyping

Our type inferencing algorithm uses subtyping to avoid pollution. In some cases, though, there would be no pollution even if plain equivalences between types would be used. One could even argue that using subtyping in those cases obscures understanding since it creates additional levels of indirection between types that would otherwise be considered equivalent. Thus, we are faced with the problem that for some types subtyping is necessary to avoid pollution, whereas for others subtyping should actually have been type equivalence.

Our solution is to include an additional abstraction layer, the *type cluster*. A cluster consists of all types that have an equivalence or subtype relation to each other (effectively regarding the subtyping relation as an equivalence relation). In case the TypeExplorer user is not interested in the subtyping details of a particular type, he can move up to the type cluster level.

Static/dynamic hypertext

We distinguish two versions of the hypertext. In the *off-line* (static) version all pages are generated in advance. The advantage of this version is portability; the complete documentation can be reproduced on a CD, taken anywhere, and browsed on almost any computer system (only requiring a standard webbrowser). Disadvantages are the static nature of the hypertext and the lack of dynamic querying.

In the *on-line* (dynamic) version the pages are generated on the fly based on queries on a database attached to the links clicked on. When the users makes updates, for example to improve the name of a type, such changes are propagated immediately. Advantages of this approach are the ability to generate hypertext based on queries by the user and the immediate response to changes. Disadvantages are the lack of portability and relatively high technical requirements on the computer system that is used for browsing.

What are good starting points for browsing?

To be flexible and generic enough to handle the multitude of program understanding tasks, the resulting hypertext should support multiple starting points. Example starting points are persistent data stores, program signatures, types matching a given name pattern (with an effect similar to seeding in year 2000 tools), or a specific variable directly in the source code. In the *off-line* version, the top-level index pages should easily lead to such starting points. In the *on-line* version, more flexibility is provided, as queries can be used to arrive at the desired HTML page.

Annotations

For programs, it is possible in some cases to derive a textual description explaining their behaviour based on the comment prologue [DK99a]. Since types are abstractions that are not directly present in one particular place in the source code, it is not possible to find meaningful texts explaining types automatically. Therefore, we give maintainers the ability to add (optional) annotations by hand. In practice, such a feature will be used mostly for types that play a significant role in the system. Furthermore, there can be a special annotation allowing a maintainer to improve the name given to a type. In the on-line version, annotations can be added on the fly, and have immediate effect; in the off-line mode annotations are incorporated after regeneration.

7.4.2 Information Available Per Type

The most important pages in TypeExplorer are those that explain an inferred type, so we will first discuss the contents of these pages. An overview of the various page elements is shown in Figure 7.1.

Pictures

The declared Cobol *pictures* of primitive types provide information about the bytes occupied and the intended use (number, character, ...). In most cases, all primitive

Element	Available Information
annotation	hand-written description of this type
structure	the picture or record declaration(s) of variables of type τ
values	all literal values found for τ
type graph	visualization of sub and supertypes of τ
usage	links to source code lines where a variable or literal of τ is used
parents	links to records with fields of type τ
programs	links to programs that use τ
copybooks	links to copybooks that use τ
words	list of domain concepts extracted from names of variables of type τ (based on heuristics)
type name	suggestion for name of this type based on these domain concepts

Figure 7.1: Information presented for a type τ.

types in an equivalence class will have the same picture. If the pictures are different, this means that the COBOL code using variables of this type relies on coercions, which may indicate bad programming style or potential programming errors.

Records

If the primitive types of a type τ are all *records*, the most common case is that all variables in this type are declared with the same number of fields, each of the same length. In this case, our rule of substructure completion will infer equivalences between these field types, If they are of different shape, *aggregate structure identification* [EHM+99, RFT99] can be used to find subfields that are small enough to unify the various records in τ. Thus, although the primitive records in τ may be of different shape, we infer one record type with the smallest necessary fields for τ, and list the fields of τ in its page.

Literals

The inferred literals provide information about the sort of values that are permitted for this type. Moreover, they show which literal values are actually used in the system analyzed. Since a supertype τ can hold at least the values of all its subtypes, we also list the literals in all subtypes of τ.

Usage

In addition to structural information about a type τ, we can provide data on its *usage*. We include links to source code lines in which a variable of type τ is used, as well to those lines in which a literal of type τ is used. Moreover, we include links to the documentation of all programs and copybooks that use the type.

For types used as *fields* in other records, we include a link to each of the parent records.

Type Graphs

An inferred type τ can be related to other types via subtype (or supertype) relationships. As part of the documentation generated for a type τ, we display all sub- and supertypes of τ in a *type graph*. An example type graph is shown in Figure 7.2. This figure comes from the actual type web derived for the case study described in Section 7.6.[3]

The nodes in the graph are types: the text in a node is the name chosen for a type. This name is obtained by picking one of its primitive types as representative. Clicking on the nodes brings up the page for the type clicked on. The type τ itself is shown in a (red) ellipse. In Figure 7.2 it has name *har006.feature*. An arrow from τ_1 to τ_2 means that τ_1 is a subtype of τ_2.

[3] For presentation purposes, we have translated the variable names from Dutch into English in the figure.

A number of observations can be made from this graph. First of all, the subtype relationship on types closely corresponds to the assignment relationship between variables. Thus, one can read an arrow $\tau_1 \rightarrow \tau_2$ also as: "variables of type τ_1 are assigned to variables of type τ_2."

Second, within the graph, one can recognize groups of related types: in Figure 7.2, examples are the three *kind* types on the right, or the four *payment* types in the middle.

Third, the type selected, *har006.feature*, happens to be a supertype of several other types. Thus, *har006.feature* can accept values of several different subtypes, dealing with various sorts of numbers, such as *country codes*, *title codes*, etc. Such a type with several different subtypes is typically the *input* parameter of a procedure or program, where each incoming edge corresponds to the subtype of an actual parameter. If we would not infer subtypes, but equivalences instead, all these types would become the same (via *har006.feature*).

Fourth, some types have dashed outgoing (or incoming) edges. This means that these types have other supertypes (subtypes), which are, however, not sub or supertypes of the type selected, *har006.feature*. An example is the left most *salutation* type. Its outgoing edge to *har006.feature* means that *salutations* are moved to *features*: its dashed outgoing edge means that *salutations* are moved elsewhere as well.

Fifth, the type *c502.num* only has outgoing edges. This typically means that *c502.num* is the output parameter of procedure or section. Furthermore, the fact that *c502.num* has no incoming edges means that there are no assignments from other types into *c502.num*. This can mean one of three things for variables of type *c502.num*:

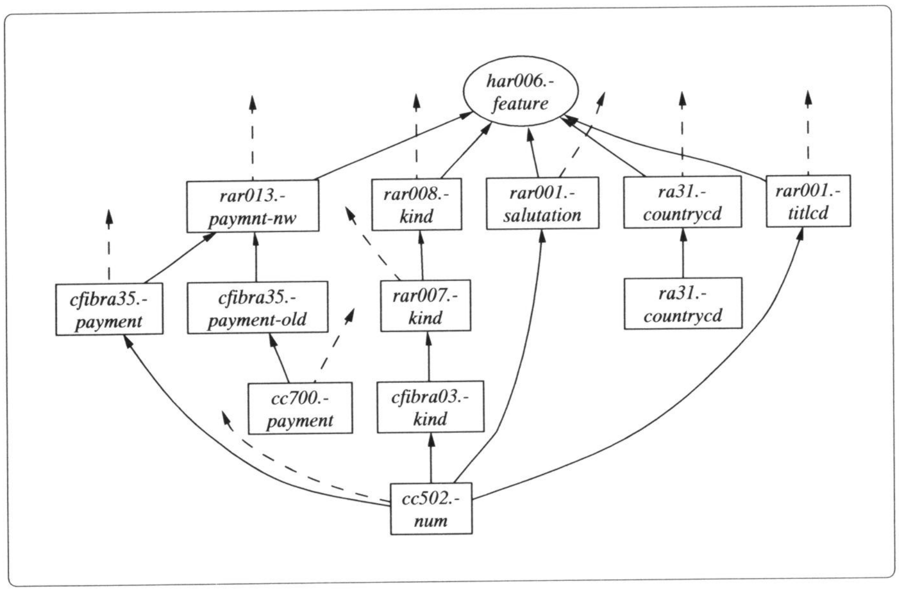


Figure 7.2: Example Type Graph

1. They never get a value within the programs analyzed, but only in external libraries.
2. They do get a value, but only from variables also of type *c502.num*
3. They do get a value, yet not as a scalar value, but viewed as an aggregate. This, is in fact the case for *c502.num*, which is filled as an array, digit by digit.

In short, type graphs can be used to show a number of interesting properties regarding types and variables. For the case studies conducted, most of the type graphs are reasonably small and understandable. The dashed arrows are an important tool to keep them small: If we would expand all dashed arrows transitively, the type graph for *haroo6.feature* would become several hundreds nodes larger.

7.4.3 Types in Programs and Copybooks

To present types in the context of programs and copybooks, we integrate them with system documentation that is automatically derived from legacy sources using DocGen. This hypertext describes the system at various levels of detail. At the program level we find copybooks that are included, flatfiles read or written, database tables that are updated or selected, screens that are presented to the user, etc. Zooming in from the program level, we arrive at the level of the individual sections, copybooks, and ultimately the full source. Zooming out, we arrive at the subsystem level that groups collections of batch (JCL) jobs, programs, copybooks, etc. corresponding to subsystem decompositions as used by the maintenance team (usually visible in naming conventions or directory structure) or as found by automatic clustering techniques. A more detailed account can be found in [DK99a].

One obvious (and straightforward) method of integration is to provide links from variables and literals occurring in the source code to their inferred type pages.

Moreover, we derive *signatures* for modules that are called or can be called by others. Such a signature documents the intended use of a module. It gives the types of the *formal* parameters, which are derived from the variables declared in the Cobol linkage section. This not only provides information about the formal parameters: the type graph of each of the formal parameters also contains subtypes for all actual parameters used in the system analyzed.

Second, we obtain types for the records that are written to or read from persistent data stores such as files or database tables. In particular in Cobol systems, such records are likely to hold business-related data. The types of these records indicate how such business data is used within individual programs, or across the entire software system analyzed.

Third, we can find *type-dependencies* between programs and copybooks. Clearly, if a program uses a variable declared in a copybook, the program depends on that copybook. A second possibility, which we encountered in our case study, is that a copybook C_p containing a section (to be included in the procedure division), uses variables declared in a separate copybook C_d (to be included in the data division).[4]

[4] Since Cobol sections cannot have parameters, global variables are the only way to pass data to sections.

This leads to an inferred type dependency between the using copybook C_p and the declaring copybook C_d. In our case study, the programmers had tried to document such dependencies in comments in both copybooks — however, our analysis found additional dependencies not documented at all.

Last but not least, we provide index files to types and programs, listing all words found in types, type names, types used in signatures, types used in persistent data stores, and so on. Moreover, we augment existing index files listing all programs, tables, and so on with additional type information, such as the type signature which concisely reveals the intended purpose of a program. These index files are included at the top-level, but also at the subsystem, program, type cluster, and copybook level.

7.5 Implementation

The architecture of the TypeExplorer tool set is shown in Figure 7.3. The dashed line between documentation and querying indicates the dynamic queries available in the on-line TypeExplorer.

The toolset follows an extract-query-view approach, separating source code analysis, inferencing and presentation. This approach makes it easier to adapt to different source languages or to other ways of presenting the types found. The TypeExplorer toolset incorporates the Cobol type inferencing tools presented in Chapter 5.

In the first phase, a collection (database) of *facts* is derived from the Cobol sources. For that purpose, we use a parser generated from the Cobol grammar discussed in [BSV97a]. The parser produces abstract syntax trees (ASTs) in a textual representation called the AsFix format. These ASTs are then processed using a Java package which implements the visitor design pattern. The fact extractor is a refinement of this visitor which emits type facts at every node of interest (for example, assignments, relational expressions, etc.).

In the second phase, the derived facts are combined and abstracted to infer a number of conclusions regarding type relations. One of the tools we use for inferring type relations is grok, a calculator for *Tarski relational algebra* [Hol98]. Relational algebra

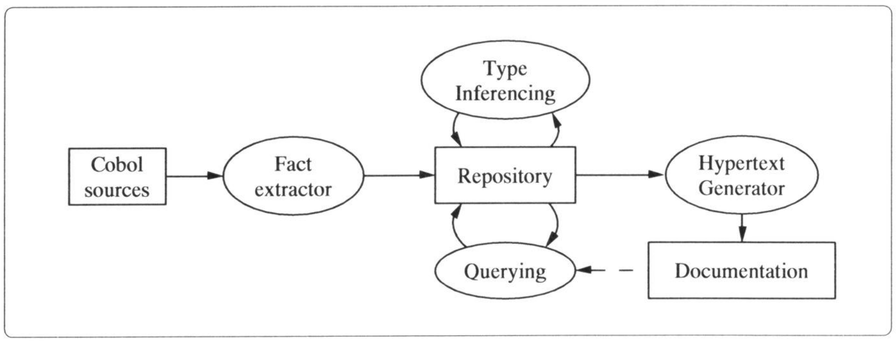


Figure 7.3: Overview of the TypeExplorer tool set.

provides operators for relational composition, for computing the transitive closure of a relation, for computing the difference between two relations, and so on. We use it, for example, to turn the derived type facts into the required equivalence relation. Finally we store the derived and inferred facts in the MySQL relational database.[5]

In the final phase, we query the database and generate hypertext documentation. We use PHP[6] to generate HTML code based on queries on the database. PHP is an HTML-embedded scripting language that was developed to allow web developers to write dynamically generated pages quickly. It contains support for a wide range of databases, including MySQL. The on-line version of TypeExplorer utilizes PHP as a server-side scripting engine to generate HTML code dynamically. For the off-line TypeExplorer, PHP is used at "compile time" to generate static HTML pages.

The pages documenting types contain pictures of type graphs showing the sub- and supertypes of a type. These type graphs are coupled to imagemaps that connect URLs to nodes in the picture allowing the user to navigate through the documentation by clicking in the graph. These graphs are extracted from the database in a Java program using the JDBC interface to MySQL.[7] The layout and imagemaps for these images are generated using the `dot` graph drawing package [GKNV93].

7.6 Using Type Explorer

TypeExplorer helps a software engineer to take a typeful look at his legacy system. In this section, we will discuss what sort of questions can be fruitfully answered by navigating through a legacy system using TypeExplorer. Clearly, TypeExplorer reveals so much information that many different questions can be answered using it. We will focus on two extremes: first, we will see that types are the natural way to reveal structure at the detailed level of individual *variables*; next we will cover how TypeExplorer helps to get a high level overview of the overall system *architecture*. Since the latter is, in our opinion, the most surprising application, we will spend most of our attention to architectural understanding using types.

Our running example will be a real life Cobol/cics system called Mortgage of approximately 100,000 lines of code. It consists of an on-line (interactive) part, as well as a batch part, and it is in fact a subsystem of a larger (1 MLOC) system. An example screen shot from a session using TypeExplorer is shown in Figure 7.4. It shows the main index, the page derived for copybook CY700, the page for type *cc700.c700-srt-adres*, as well as the type graph for one of the other types used in CY700.

7.6.1 Supporting Maintenance Tasks

One possible way of using TypeExplorer for Mortgage, is to support maintenance tasks related to specific domain concepts or variables. A (fairly common) example is to modify the representation of a group of variables (for example, expanding the

[5] MySQL is available from: http://www.mysql.org/.

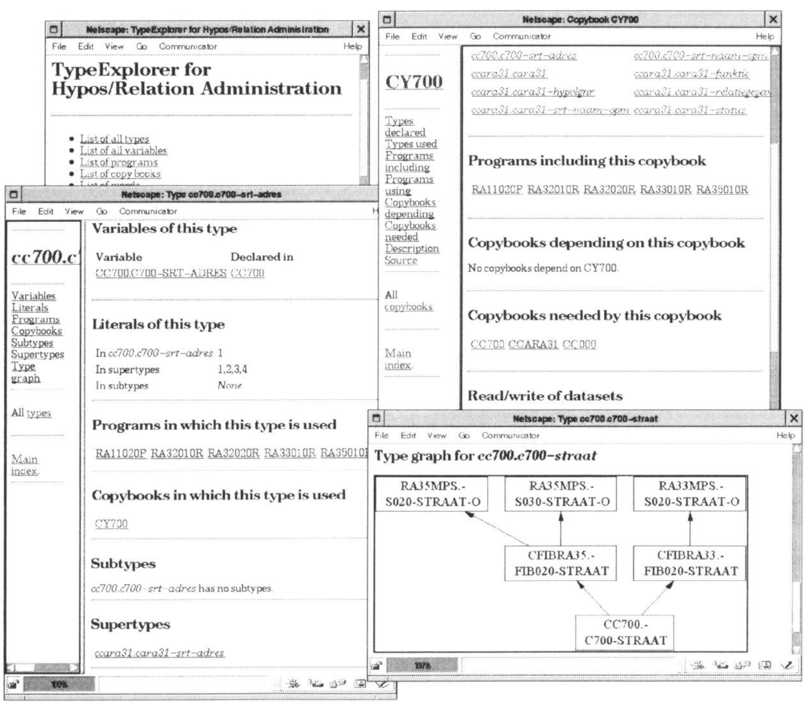
[6] The PHP Hypertext Preprocessor is available from: http://www.php.net/.

[7] Mark Matthews MySQL JDBC drivers: http://www.worldserver.com/mm.mysql/.

kind variables in Figure 7.2 from two to three digits). Since COBOL has no facilities to encapsulate such a representation using explicitly declared types, this usually involves a painful search for all other variables affected by this modification, including those via chains of assignments. TYPEEXPLORER helps the maintainer to operate at the higher type level, which immediately provides all related variables.

7.6.2 Architectural Structures

TYPEEXPLORER can be used to analyze the the as-implemented software *architecture* of a system. Bass *et al.* [BCK98] define this as the system's structure, which comprises software components, the externally visible properties of those components, and the relationships among them. Bass *et al.* emphasize that there generally are multiple structures (called *architectural structures*), and that no one structure holds the irrefutable claim to being *the* architecture. Example architectural structures manifest themselves at the level of modules, processes, data flow, control flow, and so on. We argue that the *type* structure of a system is an additional architectural structure, which is important not only for systems constructed using strongly typed languages, but also


Netscape: TypeExplorer for Hypos/Relation Administration
File Edit View Go Communicator Help
Netscape: Type cc700.c700-srt-adres
File Edit View Go Communicator
Netscape: Copybook CY700
File Edit View Go Communicator Help
Netscape: Type cc700.c700-straat
File Edit View Go Communicator Help
RA35MPS.- S020-STRAAT-O
RA35MPS.- S030-STRAAT-O
RA33MPS.- S020-STRAAT-O
CFIBRA35.- FIB020-STRAAT
CFIBRA33.- FIB020-STRAAT
CC700.- C700-STRAAT

TypeExplorer for Hypos/Relation Administration

- List of all types
- List of all variables
- List of programs
- List of copy books

cc700.c

Variables
Literals
Programs
Copybooks
Subtypes
Supertypes
Type graph

All types

Main index.

Variables of this type

Variable	Declared in
CC700.C700-SRT-ADRES	CC700

Literals of this type

In *cc700.c700-srt-adres*	1
In supertypes	1,2,3,4
In subtypes	*None*

Programs in which this type is used

RA11020P RA32010R RA32020R RA33010R RA35010I

Copybooks in which this type is used

CY700

Subtypes

cc700.c700-srt-adres has no subtypes.

Supertypes

ccara31.cara31-srt-adres

CY700

Types declared
Types used
Programs including
Programs using
Copybooks depending
Copybooks needed
Description
Source

All copybooks

Main index.

cc700.c700-srt-adres *cc700.c700-srt-naam-opm*
ccara31.cara31 *ccara31.cara31-funktie*
ccara31.cara31-hypolgnr *ccara31.cara31-relatiegegev*
ccara31.cara31-srt-naam-opm *ccara31.cara31-status*

Programs including this copybook

RA11020P RA32010R RA32020R RA33010R RA35010R

Copybooks depending on this copybook

No copybooks depend on CY700.

Copybooks needed by this copybook

CC700 CCARA31 CC000

Read/write of datasets

Type graph for *cc700.c700-straat*

Figure 7.4: The TYPEEXPLORER in action.

for legacy systems built using untyped languages such as Cobol. TypeExplorer helps to inspect this type structure.

Bass *et al.* [BCK98] provide three reasons why software architecture is important: (1) it helps in *communication among stakeholders*; (2) it makes *design decisions* explicit; and (3) it provides a *transferable abstraction of a system*. TypeExplorer helps to achieve these goals for type structures as well. To illustrate this, we will navigate through the Mortgage case study, and discuss some architectural issues of interest.

7.6.3 Exploring Mortgage's Architecture

When exploring Mortgage, a natural starting point is the index listing all programs together with their inferred signature. When doing this, one observation can be immediately made: The type of the first formal parameter of all batch programs is the same – the *program-fields* type. This raises the question why this is so, and what sort of type this *program-fields* type is. Inspection shows us that it is a record-type, storing the name of the program, the current status, the name of the files currently processed, etc. Moreover, it holds data which is not necessary for the proper execution of the program. Instead, the data is used to quickly find the program responsible for the problems if one of the batch runs crashes.

This shared first parameter shown by TypeExplorer thus immediately leads to an architectural requirement, namely that the system should support fast repairs and restarts at the proper position whenever one of the batch runs crashes in the middle of the night.

TypeExplorer also shows us that this convention is actually used. The *program-fields* record contains one field (the *subroutine* field) holding the name of the program currently being run. TypeExplorer lists all literal values that are used for (i.e., assigned to variables of) the type *subroutine*, This list exactly corresponds to the list of all batch programs, which is the result of the fact that each program correctly starts by setting the *subroutine* field to the program's name.

It is interesting to observe that Mortgage also clearly shows that just looking at the *names* of formal parameters is not sufficient. To see why this is so, we take a look at the *on-line* part of Mortgage (the part invoked from screens via cics). The first parameter of each on-line program is the same, namely DFHCOMMAREA. However, they all have a different type! All DFHCOMMAREA variables are strings of different lengths. The specific name DFHCOMMAREA is required by cics. The first thing each program does is to assign that variable to a more structured record variable. It is the type of that structured record variable that TypeExplorer recognizes as the appropriate type for the first parameter of the linkage sections, which it displays in the inferred signature.

TypeExplorer also helps us to understand the meaning of the program parameters. For example, many programs in Mortgage have integer-valued numbers as parameters (having picture string S(9) COMP-3). Often, these are in fact enumeration types, in which case TypeExplorer recognizes them as such. Several programs turn out to have a parameter named *function*, with 5 to 10 permitted values. Based on this function value, the program performs one of several functions. This leads us to two design decisions: different (but related) functions are grouped into programs, and the mechanism used is a switch on an enumerated value, instead of the Cobol feature in which one program can have multiple entry points.

Last but not least, TypeExplorer shows how such *function* enumeration parameters are passed from one program to another. As an example, one of the Mortgage programs contains a parameter for determining how a person's name is formatted (full first names, one initial only, with title, and so on), and another to format street names (capitalized, street abbreviated, and so on). One of the top level programs has 10 different parameters, corresponding to these formatting codes. The types inferred exactly show how each of the codes (which are all integer numbers) correspond to the parameters of the various formatting programs.

In short, TypeExplorer can be used to discuss whether requirements such as crash recovery are properly supported, how functionality is grouped in modules, and how modules are dependent via types. Other architectural issues can be identified using TypeExplorer by studying the type relationships between copybooks, the use of database record types across programs, and so on.

7.7 Concluding Remarks

In this chapter, we have shown how hypertext-based program understanding tools can be achieve higher levels of abstraction by using inferred type information for cases where the underlying software system is written in a weakly typed language. We proposed TypeExplorer, a tool for browsing Cobol legacy systems based on these types. The main contributions of this chapter are in the following areas:

Presentation: Although types are an invented abstraction, not directly present in the code, we showed how they can be made tangible by displaying a name for them, associated domain concepts, literal values, and variable use in the source code. Moreover, type graphs help to see types in context, and view their relationships to other types. Last but not least, type information can be integrated with pages documenting programs, databases and copybooks, extended them with type links for program signatures, copybook dependencies, and record types for persistent data stores.

Implementation: We have described an implementation based on the extract–query–view paradigm, using Tarski relational algebra, SQL, and PHP to realize both an on-line and off-line version of TypeExplorer.

Use: We have shown how navigating through a legacy system using TypeExplorer provides useful information both at the detailed level of individual programs and at the higher level of the overall architecture. We have applied TypeExplorer to an actual system, and used it to identify type-dependencies between programs, understand design decisions, and to highlight requirements such as support for crash recovery.

Our next step will be to distribute TypeExplorer to industrial users. Undoubtedly, this will raise additional requirements and questions, on which we will report in the near future. One possible extension is to propagate types via batch jobs (JCL) as well, thus arriving at better types for the datafiles processed.

Another area of future work is to use TypeExplorer to support the migration of Cobol to the new Cobol standard, which is an object-oriented extension of Cobol-85. This new version of Cobol does support types, and offers the possibility of using type definitions. Our tools provide the technology to take advantage of this new possibility.[8]

Acknowledgments

We would like to thank Jan Heering (CWI) for commenting on an earlier version of this chapter.

[8] The first steps towards this migration are presented in the previous chapter (Chapter 6) were we combine inferred types with concept analysis to support object identification.

Part III
Refactoring & Testing

Chapter 8

Java Quality Assurance by Detecting Code Smells

Software inspection is a known technique for improving software quality. It involves carefully examining the code, the design, and the documentation of software and checking these for aspects that are known to be potentially problematic based on past experience.

Code smells are a metaphor that is used to describe the patterns that identify the locations in a software system that could benefit from refactoring. In this chapter, we investigate how the quality of code can be automatically assessed by checking for the presence of code smells and how this approach can contribute to automatic code inspection.

We describe the design and implementation of jCOSMO, a prototype code smell browser that detects and visualizes code smells in JAVA source code and show how this tool was applied in a case study. The work presented in this chapter was published earlier as [EM02].

8.1 Introduction

Software inspection is a known technique for improving software quality. It was first introduced in 1976 by Fagan [Fag76] and has since been reported on by numerous others, for example [Rus91, GG93]. Software inspection involves carefully examining the code, the design, and the documentation of software and checking these for aspects that are known to be potentially problematic based on past experience.

It is generally accepted that the cost of repairing a bug is much lower when that bug is found early in the development cycle. One of the advantages of software inspection is that the software is analysed *before* it is tested. Thus, potential problems are identified in the beginning of the cycle so that they can be solved early, when it's still cheap to fix them.

Traditionally, software inspection is a formal process that involves labor-intensive manual analysis techniques such as formal code reviews and structured walk-throughs. Inspection is a systematic and disciplined process that is guided by well-defined rules. These strict requirements often backfire, resulting in code inspections that are not performed well or sometimes even not performed at all.

These problems are addressed by tools that automate the software inspection process. We distinguish two approaches:

1. Tools that *automate* the inspection *process*, making it easier to follow the guidelines and record the results.

2. Tools that perform *automatic code inspection*, relieving the programmers of the manual inspection burden.

We concentrate on the second type: tools that perform automatic inspection. Such tools are interesting since automatic inspection and reporting on the code's quality and conformance to coding standards allows early (and repeated) detection of signs of project deterioration. Early feedback enables early corrections, thereby lowering the development costs and increasing the chances for success.

8.1.1 Code Smells

The existing tools that support automatic code inspection (for example, the well-known C analyzer LINT [Joh78]) tend to focus on improving code quality from a technical perspective. The fewer bugs (or defects) there are present in a piece of code, the higher the quality of that code. From this perspective, code inspection boils down to low-level bug-chasing and we see this reflected in the tools which typically look for problems with pointer arithmetic, memory (de)allocation, `null` references, array bounds errors, etc.

In this chapter, we will focus on a different aspect of code quality: Inspired by the metaphor of "*code smells*" introduced in the refactoring book [Fow99], we review the code for problems that are generally associated with bad program design and bad programming practices.[1]

Beck and Fowler introduce the metaphor of "*code smells*" to describe the patterns in code that indicate that refactoring can be applied. "*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*". It improves the design of a software system after it was written by tidying up code and reducing its complexity. The resulting software is easier to understand and maintain.

Code smells can be used to answer the question of *when* and *what* to refactor. The idea is not necessarily that no code smells are permitted, but rather that code

[1] Please note that we do not regard this type of quality to be more important or better than the former. Both aspects should be considered when trying to improve overall software quality. We decided to focus on the second type since it is currently much less supported than the first type.

smells are hints which tell us that refactoring may be beneficial. Some examples of code smells are: duplicated code, methods that are too long, classes that contain too much functionality, classes that violate data hiding or encapsulation rules or classes that delegate the majority of their functionality to other classes.

8.1.2 Coding Standards

Another important code quality aspect of large scale software development is conformance to coding standards. Coding standards ensure that everyone in the company can understand (and work with) each others' code. If conformance is not achieved, i.e. if the code is not written and organized according to the programming guidelines, it becomes much harder for a large team of programmers to develop, integrate, and maintain a particular piece of software. This becomes even more important in an environment where developers are geographically distributed, as is, for example, commonly found in open source development projects.

Unfortunately, conformance to coding standards is not always easy to achieve in practice. All developers involved in the project have to know and appreciate the guidelines enough to build software according to them. Experience shows that just publishing a set of programming guidelines is not enough. If developers do not really understand (the ideas behind) a particular rule, feel restricted by it, or maybe just do not believe that this rule can be useful, they are more likely to ignore that rule during development. In other cases, the set of guidelines may be so large that it is easy to overlook some of them during development. When the project comes under time constraints, these effects are often even stronger.

Consequently, overall code quality can be improved by ensuring that the code conforms to the coding standards. This process is supported by automatic conformance checking. By allowing for the definition of additional (project specific) smells, automatic smell detection turns into a conformance checking process.

In this chapter, we investigate whether detection of code smells can contribute to automatic software inspection. Section 8.2 discusses a general approach for building a software inspection tool that is based on detection of code smells. Section 8.3 introduces the case study that was used to investigate the feasibility of the approach. Section 8.4 describes the implementation of jCosmo, the prototype code smell browser that was developed in the case study. We conclude with an evaluation of the case study, an overview of related work, and the discussion of future work and contributions in Section 8.5.

8.2 Approach

There are a number of important questions that need to be answered before we can automate detection of code smells in program code and use them for software inspection: What code smells are we going to detect? How are we going to detect these smells? How are we going to present the results?

The remainder of this section discusses these questions and the issues that surround them in more detail. This results in a generic approach for building software inspection tools that are based on code smell detection.

8.2.1 What smells are we going to detect?

In the refactoring book, Beck and Fowler present a list of code smells that they use to look for refactoring potential [Fow99]. These smells range from simple patterns that everyone discourages, such as "code duplication" and "long methods", to more complex patterns that originate from object oriented design issues, such as "parallel inheritance hierarchies" (if you extend or change one hierarchy you will need to do the same with the other) and "message chains" (this smell is also known as the Law of Demeter: a client should not navigate through the object structure, for example, as is done in the call `a.getThis().getThat().foo()`).

Other examples of code smells are: "large class" especially w.r.t. classes with many fields, "feature envy" for methods that access more methods and fields of another class than of its own class, "switch statements" where inheritance should be used for specialization, "data class" for classes that do not contain functionality, only fields, "refused bequest" for classes that leave many of the fields and methods they inherit unused, and "data clumps" for clusters of data that are often seen together as class members or in method signatures but are not grouped in a class.

We can make a few observations about this list that influence our design. First, such a list of code smells can never be complete: there will always be domains and projects where a different set of code smells should be applied. For example, in Chapter 9, we present a number of smells that can occur in unit test code and describe the corresponding refactorings to remove them.

Second, code smells are subjective: they are based on opinions and experiences. Creators of a list include those patterns that they found to be useful indicators of potentially problematic aspects of the code. However, not all smells may be supported by concrete evidence and some of them may be inspired by aesthetic considerations. For example, some developers strive to minimize the use of typecasts and consider typecasts to be a smell, while others see absolutely no harm in typecasts and do not want to regard them as smells.

Finally, code smells are not precise: "*One thing we won't try to do here is give precise criteria for when a refactoring is overdue. In our experience no set of metrics rivals informed human intuition*" [Fow99, p.75]. This is related to the subjectivity of code smells. For each project, one needs to decide what the actual parameters are: for example, which variable naming convention are used and what is the maximum size of classes and methods that is allowed, etc.

From these observations, we can only conclude that one of the main design requirements for a code smell inspection tool is that the smells should be configurable by the user. As tool builders, we can predefine a number of smells but configurability is needed to allow for: (1) definition of additional smells, (2) removal of smells that should not be considered, and (3) more precise definition of a smell so that its parameters can be tuned.

8.2.2 How are we going to detect these smells?

Examination of the list of code smells shows that each of them is characterized by a number of *smell aspects* that are visible in source code entities such as packages, classes, methods, etc. A given code smell is detected when all its aspects are found in the code.

We distinguish two types of smell aspects: *primitive smell aspects* that can be observed directly in the code, and *derived smell aspects* that are inferred from other aspects. An example of a primitive aspect is "method m contains a switch statement", an example of a derived aspect is "class C does not use any methods offered by its superclasses".

This distinction is used in the design of the smell detection process that is separated into the following steps:

1. Find all entities of interest in the code.
2. Inspect them for primitive smell aspects.
3. Store information about entities and primitive smell aspects in a repository.
4. Infer derived smell aspects from the repository.

This process constructs so-called *source models* from the program text. These source models are the abstraction of a system's source code that is needed for smell detection. The structure of these source models is described by a meta-model. Our meta-model was designed with analysis of JAVA programs in mind. It contains information regarding program entities such as packages, classes, interfaces, exceptions, methods, constructors, static blocks, and fields. Furthermore, it describes the relations between these entities such as composition, inheritance, interface implementation, thrown exceptions, inner classes, method calls, field accesses, and field assignments.

Our meta-model is very similar to those used by other JAVA analysis tools such as Shimba [SYM00] and RevJava [Flo02]. Furthermore, it has considerable overlap with the Famix meta-model that was designed with generic OO reverse engineering and refactoring in mind [Tic01]. Although our current focus is on the JAVA programming language, we feel that this overlap indicates that it is possible to generalize our approach to other object oriented languages.

So how do we find and inspect all entities of interest in the code? Since the code smells are described in terms of program patterns and not in terms of behavior patterns, dynamic (runtime) information is not needed for smell detection. Therefore, source models can be extracted using static analysis of the program: First, a parser reads the source code and produces a parse tree containing all structural information contained in the code. Second, an analyzer reads these parse trees and traverses them according to the program structure. During this traversal, the analyser visits all program entities and stores their structure and relations in the repository. When primitive smell aspects are observed, these are also stored in the repository.

Note that is is possible to extend our approach to include code smells that need runtime information for their detection. For this, we need to add a separate extractor that is used to augment the source models with the necessary dynamic information.

In general, such a "dynamic" extractor will collect this information using source code instrumentation. In the case of JAVA analysis, an attractive alternative is monitoring the runtime environment via its debugging interface.

8.2.3 How are we going to present the results?

The next question we have to ask ourselves is how the smells should be presented to the user after they have been detected. There are several ways in which this can be done. The intended use of the tool will have substantial influence on the the type of presentation used. We distinguish three classes of users for the detection results:

a. Programmers that use detected smells during development or maintenance of a system to improve the code.

b. Code inspectors (or reviewers) that use detected smells to assess the quality of the code.

c. Tools that use the detected smells to perform further analysis or transformations on the code, for example, software refactoring tools. Generally, these tools do not need specific presentations, they just use the repository content for further processing.

Note that we do not consider software integrators as a separate class but assume that they switch between the inspector and developer roles.

Smell presentation for programmers should be integrated with the normal development process. This can be done without much intrusion by treating smells similarly to compilation errors and warnings. Depending on the IDE, smells will then be shown in separate message panes or integrated with the class browser (as for example is done with compiler errors in IBM's VisualAge for Java and the new Eclipse platform). Another possibility is building a dedicated smell browser that can be used to explore the repository (similar to the Windows Explorer interface). Since the main focus of this chapter is software inspection, we will not investigate this approach any further.

Software inspectors have special presentation requirements for assessing the quality of (potentially large) software systems. They need to be able to get a quick overview of the complete system, showing *if* the system contains bad smells, *what* parts are affected, and *where* the concentration of smells is the highest.

We can support these requirements by generating graphical representations of the software system, in particular by visualizing the source model using structured graphs. The nodes in these graphs are the program entities of the source models (i.e., packages, classes, methods, etc.) and the edges are the relations between program entities (i.e., composition, inheritance, method calls, etc.). Since these nodes and edges can be distinguished based on their types, we can represent them using different colors and selectively hide them in various views on the graph. Graph layout algorithms can be applied in these views to improve their comprehensibility.

There are a number of options for the visualization of code smells in these graphs. For example, one can vary node attributes such as color and size for the nodes that represent code entities that possess code smells. Furthermore, code smells can be visualized using additional nodes that are connected to the entity in which they are present. It is also possible to use code smells during the computation of the graph layout, for example, by ordering them in such a way that nodes with the most smells come first (i.e., in the upper left corner of the picture).

8.2.4 Modular Architecture

As we have seen, it is important that our code smell inspection tool is easily extendable because new smells are likely to be added during its lifetime. In particular, it should be possible for users to add their own code smells. We can support this in our design by using a modular architecture that encapsulates the detection of primitive smell aspects and inference of derived smell aspects in separate units. With such an architecture, addition of new code smells is as simple as extending the set of detectors or inference rules.

The presentation side of our tool should also be robust against addition of new smells. This has two aspects: (1) addition of new smells should not break existing visualizations, and (2) preferably, we want new smells to be included in the main views without extra work.

An overview of the architecture is shown in Figure 8.1. In this figure, the boxes depict inputs and outputs, the ellipses depict processing. Double lined shapes are used to indicate that this item can occur a number of times (for example, there exist several extractors for the different primitive smell aspects but only one extractor for the program structure).

8.3 Case Study

To investigate the feasibility of the described approach, we have performed a case study in which we developed a prototype software inspection tool that is based on code smell detection and applied it on a Java software system.

The system analysed, called CharToon, is a tool for developing 2 1/2 dimensional animations of facial expressions [RHN99]. It was originally developed as a research prototype at the CWI under the direction of Paul ten Hagen. The research group has

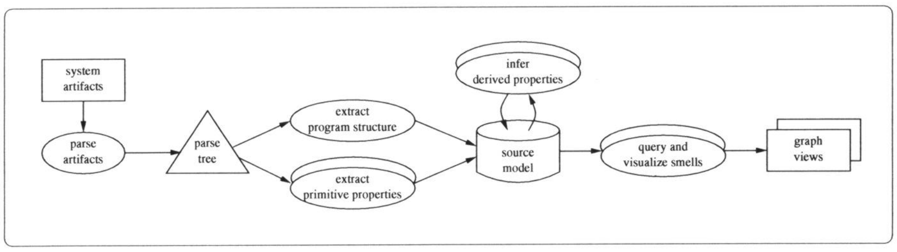


Figure 8.1: Architecture of code smell browser.

formed a spinoff company and is in the process of reviewing their code intensively in preparation for releasing it as a commercial product.

One of the principal software engineers involved in this task consulted with us to see whether we could provide tool support for refactoring and quality assurance. Based on our discussions and guided by the company's coding standards, we have added detection of some specific code smells to our prototype. The remainder of this section discusses these additional smells in more detail.

8.3.1 Instanceof as Code Smell

In JAVA, the `instanceof` operator is used to check that an object is an instance of a given class or implements a certain interface. These are considered code smell aspects because a concentration of `instanceof` operators in the same block of code may indicate a place where the introduction of an inheritance hierarchy or the use of method overloading might be a better solution.

We have found two typical patterns in which this occurs: The first is characterized by a sequence of conditional statements that test an object for its type. When the type is found, the object is cast to that type and a method is called. This can be refactored by introducing a common interface that defines the method and lets the runtime environment call the appropriate method using dynamic method dispatch (also known as late binding).

The second pattern is characterized by a method that takes a variable of the type Object (the supertype of all JAVA classes) as a parameter and has a body which contains a sequence of conditional statements that perform different actions depending on the object's type. In this case the original method can be broken up into a series of overloaded methods, each taking one of the types tested for before as a parameter. This removes the instanceof statements, making the code more modular and easier to understand.

8.3.2 Typecast as Code Smell

Another code smell that was added for the case study involves typecasts. Typecasts are used to explicitly convert an object from one class type into another. Many people consider typecasts to be problematic since it is possible to write illegal casting instructions in the source code which cannot be detected during compilation but result in runtime errors.

One typical pattern where typecasts create this smell can be observed when objects are stored in one of the container classes from the JAVA API. Because these classes are written as generic containers for objects of any type, items are automatically upcasted to their generic supertype Object when they are put in a container. When the programmer retrieves items from a container, they have to be explicitly downcasted to whatever type they used to be. However, since this type is not always known (and storage methods accept all types of objects), it is possible to perform illegal casting which results in a runtime error.

This pattern can be remediated by creating a wrapper around the container that is specific to the type that the container is going to hold. This way, casts are hidden in the wrapper class, and static type checking makes sure that the correct type is put into the container. In addition, the wrapper class can get a descriptive name that makes it clear what objects are contained. This is especially useful if the container is passed around in the program.

8.4 Prototype Implementation

To illustrate the approach described in Section 8.2, we have implemented jCosmo, a prototype code smell browser. This tool was created following the architecture described in Figure 8.1. It consists of two main phases: the code smell extraction and the visualization. During extraction, the source code is parsed and a source model is generated that describes program structure and code smells. This source model is read during visualization to generate different views on the source code and its smells.

8.4.1 Extraction

The extraction of program structure and primitive smell aspects was implemented using the Asf+Sdf Meta-Environment [BDH$^+$01], an environment for developing language centered tooling that was developed at the CWI (Centre for Mathematics and Computer Science) in the Netherlands under the direction of Paul Klint. This environment supports the generation of parsers, syntax directed editors and language processing tools such as interpreters, type-checkers and source-to-source transformations. It takes two types of input: (1) *language syntax definitions* written in the formalism Sdf [Vis97] and (2) *language processing definitions* written in the term rewriting language Asf [BHK89].

The parser generator produces *generalized LR* (GLR) parsers. Generalized parsing allows definition of the complete class of context-free grammars instead of restricting it to a non-ambiguous subclass such as LL(k), LR(k) or LALR(1), which is common to most other parser generators [Tom86]. This allows for a more natural definition of the intended syntax because a grammar developer no longer needs to encode it in a restricted subclass. Moreover, since the full class of context-free grammars is closed under composition (unlike restricted subclasses), generalized parsing allows for better modularity and syntax reuse. For more information on Sdf, we refer to [Vis97].

Programming in Asf is done in a functional fashion by means of term rewriting: rules that describe how a given term can be translated into another term. These rewrite rules can be defined using pattern matching on concrete syntax defined in the Sdf grammar. The patterns can contain variables that are bound during matching and can be reused to build the reduct. The use of concrete syntax has the advantage that the extractor writer does not have to learn a new language for processing terms.

For our prototype, we have instantiated the Asf+Sdf Meta-Environment using an Sdf definition of the Java grammar and a set of Asf modules that specify the processing that is needed for extraction of program structure and primitive smell aspects. The extraction is performed in two steps: first, the JAVA input is parsed using a parser generated from the SDF grammar. Then the parse trees are processed using term rewriting traversals in ASF to derive the desired source model. This model describes the structure and primitive smell aspects detected in the code. The source models are represented in plain text in the three tuple notation that is known as RSF (Rigi Standard Format).

We have chosen to analyse JAVA source code instead of JAVA byte-code as is done by most other tools that operate on JAVA software. This choice has a number of advantages:

- Source code analysis allows us to treat parts of software systems that do not compile by themselves because of incompleteness. When the source code parsing is based on island grammars, it is even possible to analyse source code that contains syntax errors (see Chapter 2).
- Extraction of program structure and primitive smell aspects from source code can be expressed using pattern matching over the concrete syntax of the JAVA programming language. This makes it very easy for users to adapt and extend the extractor in order to detect new code smells since they do not have to learn a new language.
- Some coding standards cannot be checked on byte-code since the byte-code contains less information than the original source code (i.e., regarding layout and variable names).

In case only the byte-code for a system is available, it can still be analysed with our tool by first feeding it through a decompiler. However, one can wonder why the code quality of such a system needs to be assessed in the first place...

After extraction of program structure and primitive smell aspects, these facts are combined and abstracted to infer a number of derived smell aspects regarding code smells. These derived smell aspects are also stored in the repository. One of the tools we use for inferring these derived aspects is GROK [Hol98], a calculator for *relational algebra* [Tar41]. Relational algebra provides operators for relational composition, for computing the transitive closure of a relation, for computing the difference between two relations, and so on. We use it, for example, to compute the "refused bequest" smell where child classes do not use the methods that were offered by their parents.

8.4.2 Visualization

The visualization was implemented using the Rigi software visualization tool that was developed at the University of Victoria, Canada, under the direction of Hausi Müller [TWSM94]. The Rigi infrastructure is based on a general graph model. This graph model is adapted to a specific domain by defining the entity types and relations of interest in a domain model. Usually, graphs are created using a parser that extracts facts from a software system and stores them in this graph model using Rigi Standard Format (RSF). The graphs are visualized using a programmable graph editor.

Generally, Rigi graphs consist of the artifacts that software engineers use to understand a software system. Examples are software components such as subsystems, procedures, variables, and the dependencies between them, such as composition, calls, and control- and data-flow. For our prototype, a new domain was added that describes the structure of Java software systems and their code smell aspects.

Unfortunately, it is not possible in Rigi to vary the color or size of a node to indicate that it possesses a smell; all nodes of a given type have the same color and all nodes have the same size. Therefore, we choose to present smells as additional nodes that are connected to the code entities that possess them. Each smell has its own node type, which has a distinct color in the graph. An advantage of this visualization is that it is easy to see which parts of the system have the most smells, and would benefit most from refactoring.

An alternative approach is to store code smell information as attributes of a node. We did not use this method since these attributes have no visual representation in the graph editor. Generally, they are used for querying the graph and their values can be inspected in a separate window.

To present the results of smell detection, we have extended the Rigi user interface with a separate jCosmo toolbar from which the user can browse the detected smells and invoke all dedicated jCosmo functionality. This includes various visualizations that provide customized views of extracted data and special filtering functions that can be used to show or hide certain nodes, arcs or labels. Moreover it provides a special "pruning" function that allows the user to select a particular subgraph and hide the rest.

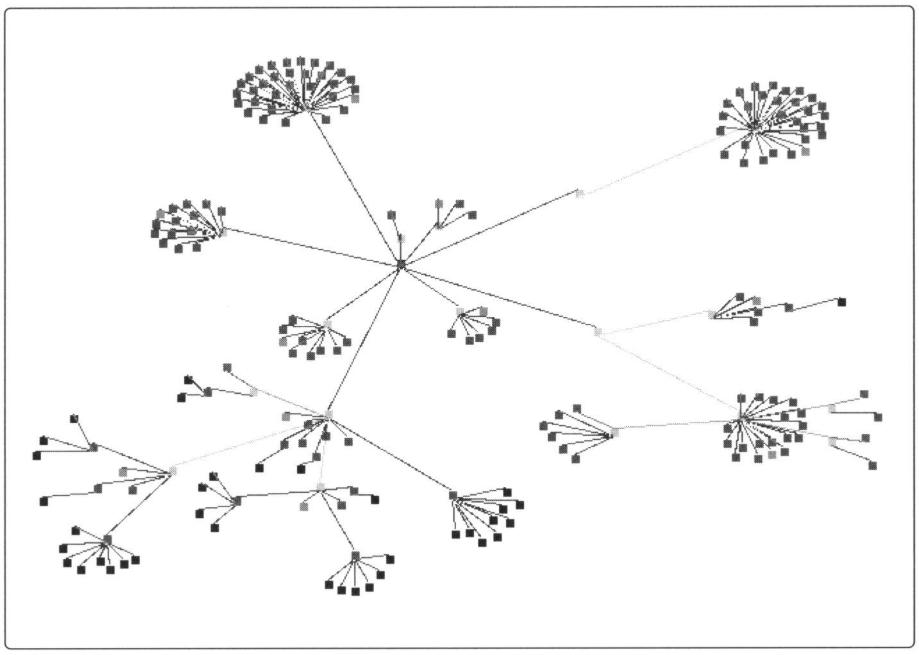


Figure 8.2: Complete smell graph for a test system.

The first view provided shows all the packages, classes, interfaces, methods and constructors, and their attached smell nodes. This gives a basic overview of the system and the distribution of the code smells. An example of this view is shown in Figure 8.2.[2] This figure shows the complete smell detection graph for a small test system consisting of 12 classes, and approximately 1450 lines of code.

A disadvantage of the previous view is that it does not scale up well for large systems. Therefore, we provide an improved view where class members such as methods and constructors are collapsed into their enclosing classes. Smell nodes are shown attached to their containing class (if a method contains a smell, it is "inherited" by the class that contains the method). By using a spring layout algorithm, it becomes clear where the code smells are clustered in the system. An example of this view is shown in Figure 8.3.[3] This figure shows the collapsed smell detection graph for CharToon (147 classes, 46,000 LOC). The class nodes in this view are so-called composite nodes: to view the smell distribution within a class, the user can double-click the class and a new view is opened which shows that class with its members and their smells.

For clarity, we have decided to hide node labels in these views. They can be redis-

[2] Please note that this figure shows a greyscale screendump of a view that was developed for color displays. The color version is available at: http://www.cwi.nl/projects/renovate/javaQA/wcre2002/curvedraw.gif.

[3] The color version is available at: http://www.cwi.nl/projects/renovate/javaQA/wcre2002/chartoon.gif.

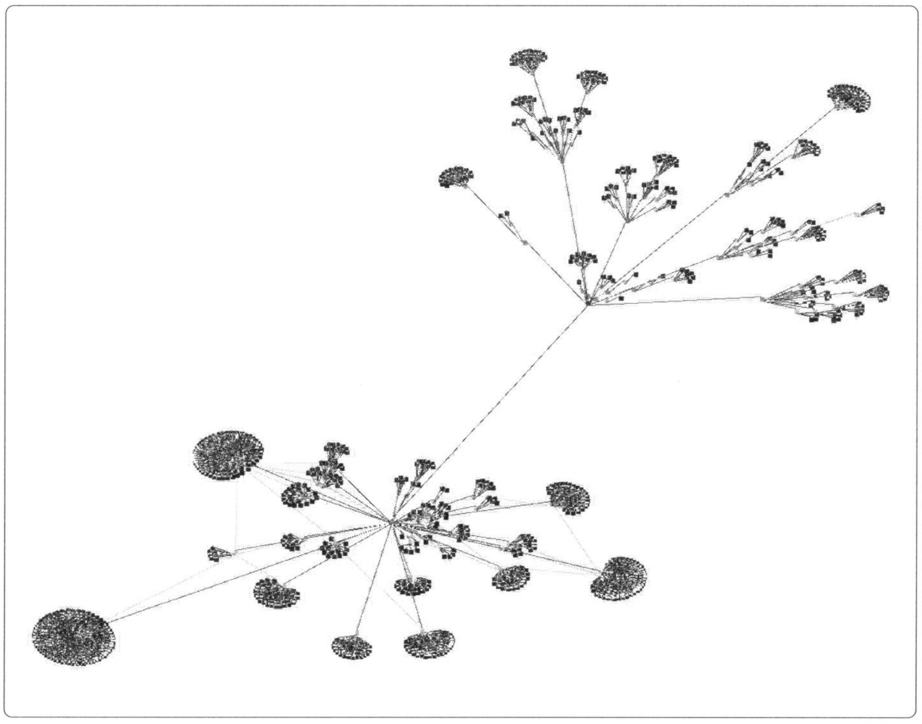


Figure 8.3: Collapsed smell graph for CharToon.

played at any time via the jCosmo user interface or by using a command. All typecast smell nodes are labeled with the types they are casting to and from (when known). This information is used by a number of filtering functions that hide certain typecasts or show only casts to a certain type. Using these functions, which are also accessible via the jCosmo user interface, a user can quickly and easily see where each kind of typecast is concentrated.

A third view orders the program entities based on code smells instead of program structure. It uses the number of smells that were detected in an entity to determine its place in the graph layout. The nodes in these graphs are ordered in a grid: all packages are arranged on the top row, distributed along an x-axis according to the average number of smells they contain. All classes are arranged on a single row below that, distributed in the same way. Finally, all methods in the system are arranged on the bottom row according to the number of smells they contain.

To prevent cluttering of the graph, this view hides the containment and inheritance arcs. Furthermore, all smells are collapsed into their containing nodes. After selecting a node in the view, the user can filter the graph using jCosmo's prune command. Pruning leaves only the parents and children of the selected node. Thus, by pruning after selecting a method node, one sees the class and package in which it is defined, whereas pruning after selecting a class node shows all methods of that class and the package in which it is defined, and pruning after selecting a package node shows all classes and methods in that package.

8.5 Concluding Remarks

8.5.1 Evaluation

The CharToon system consists of 46,000 LOC (without comments or empty lines) and 147 classes. The extraction step takes about 30 sec. on a computer with an AMD Athlon processor (1.2 Ghz) and 512 Mb main memory running linux 2.4.9-12. The extracted source model contains 33,840 facts.

Because of the number of classes and methods involved, the most useful view for examining CharToon is the one that collapses all members into their classes while leaving the smell nodes attached to the class. This view is shown in Figure 8.3. Many code smells were indicated (shown as dark nodes in the figure).

Using node filtering, we have examined the distribution of each separate smell without disturbing the layout of the graph. This immediately revealed that all but two of the instanceof nodes were clustered in one class. Opening this class node revealed that within the class the instanceofs were fairly evenly distributed between the methods. This suggested that they were not linked to a switch statement, but that this might be code that could benefit from the introduction of overloaded methods. Inspection of the source code showed that such a refactoring could be performed.

It was also clear that most of the switch statements were in the same package. This might be a hint that some of these statements may be switching on the same type and could be eliminated by the introduction of an inheritance hierarchy.

Furthermore, the majority of code smells originated from the use of typecasts. Using the predefined filtering functions, new views were created that show only the typecast to one particular type at a time. These views showed that there were clusters of identical typecasts in particular areas of the system which suggests that a small amount of refactoring could remove a large number of these smells.

Feedback from the CharToon maintainer was generally positive. He felt that the jCosmo views were useful for conformance checking and refactoring support. Additionally, they provided useful information for (re)documenting the system to other developers and management. He was very interested in being able to repeat detection after major revisions, so that conformance to coding standards and changes in the software quality could be monitored.

His only concerns had to do with possible difficulties when installing the tool and learning the interface. To address these issues, we added the jCosmo toolbar as described earlier, as well as a support web page with instructions for downloading, installing and running the tool and links to available documentation. To make installation even easier, we are working on a single packaged distribution of jCosmo that installs all the necessary components.

8.5.2 Related Work

Automatic Code Inspection

There are a number of tools that perform some sort of automatic code inspection. The most well-known include the C analyzer Lint [Joh78] and its Java variant JLint [AB01] that check for type violations, portability problems and other anomalies such as flawed pointer arithmetic, memory (de)allocation, `null` references, array bounds errors, etc. IllumaSM (formerly known as InstantQA) is a defect analysis service provided by Reasoning that identifies the location of potential crash-causing and data-corrupting errors. Besides providing a detailed description of each defect found, they report on defect metrics by measuring the software's defect density (the average number of defects found per thousand lines of source code) and its relation to standard industry norms.

Generally, these tools focus on improving code quality from a technical perspective. The fewer bugs (or defects) there are present in a piece of code, the higher the quality of that code. This differs from our approach which focuses more on code quality as seen from a program design and programming practice perspective.

More closely related to our approach is RevJava, a Java analysis tool developed at the Software Engineering Research Centre in the Netherlands [Flo02]. RevJava performs design review and architectural conformance checking. It reads Java byte-code from which facts are derived and metrics are collected. This information is used to apply critics to the system that check whether particular design rules were violated. A large number (70) of these critics are predefined and users can add their own. Reporting is done by means of a class browser that shows the rule violations for each class, method, etc, or using a browser that starts from the critics and shows all entities that violate that critic. There is no support for visualization of rule violations which makes

RevJava less suitable for getting an overview of large software systems.

Software Metrics

Another approach to assessing the quality of software systems is based on software metrics. Typically, these metrics are computed over facts that were extracted from the system's source code, which is similar to our analysis. Chidamber and Kemerer describe a suite of software metrics for object oriented systems [CK94]. Systä *et al.* report on using this suite for JAVA quality analysis in their tool Shimba [SYM00]. As in our approach, Shimba represents programs as graphs where the nodes are program entities. The computed metrics are stored as attributes of a node which can be inspected in a separate window and used for querying. The metrics are not used to determine the color or layout of nodes in the graph.

CodeCrawler is a program understanding tool that combines software metrics and graphs [DDL99]. Again, nodes represent program entities, however CodeCrawler's distinguishing feature is that it reports on the metrics of that entity by varying the size, color and position of the nodes in the graph.

There are also a number of commercial tools that use software metrics to compute the complexity and quality of software systems and present results using colored structure charts, scatterplots, metric charts and Kiviat diagrams. These tools include, amongst others, the McCabe QA and McCabe Reengineer tools by McCabe & Associates, and the Hindsight tool by IntegriSoft.

AntiPatterns

Antipatterns are an extension of the design pattern idea: where design patterns describe good solutions to frequently occurring problems, antipatterns are patterns that describe frequently observed bad solutions for a given problem. Antipatterns explain why that solution looks attractive, why it turns out to be bad, and what positive patterns are applicable instead.

Brown *et al.* describe a number of antipatterns that can be found in software development, software architecture and (software) project management [BMIM98]. The software development antipatterns are very similar to code smells in that they describe commonly seen patterns in code that could benefit from refactoring. However, antipatterns are generally at a somewhat higher level, referring to source code entities at the class level or higher.

Examples of development antipatterns include "the blob" for large classes that monopolize processing, "golden hammer" for the misapplication of a familiar solution for every possible problem, "poltergeists" for classes with limited responsibility and lifetime, and "cut and paste programming" for duplicate or near-duplicate code.

Refactoring tools

Finally, the work described in this chapter is related to the growing body of work on tools that support the (automatic) refactoring of software systems. The original refactoring tool is the Smalltalk Refactoring Browser that was developed by John Brant and Don Roberts [RBJ97]. Recently, several other commercial and open-source tools started to offer refactoring support. These include development environments such as the Eclipse platform, Borland's JBuilder and IDEA by IntelliJ and refactoring tools such as jFactor by Instantiations, ReTool by Chive, and XRefactory that act as add-ons to popular programming environments.

All these tools have in common that they focus on the actual *code transformation* and do not analyse *when* a certain refactoring can (or should) be applied. The smell detection described in this chapter could be used to add such an analysis to a programming environment allowing for an "intelligent" refactoring assistant that signals when a given refactoring can be applied.

8.5.3 Future Work

Beck and Fowler describe a number of smells that we can characterize as "maintenance smells". What we mean by this is that these smells are not obvious from the code itself but that they manifest themselves during maintenance of the code. These smells include:

- Divergent Change: when different parts of a class are changed in different situations.
- Shotgun Surgery: a smell that occurs when making changes requires changing many different classes.
- Parallel Inheritance Hierarchies: This is the case when making a new subclass in one place also makes it necessary to add a new subclass in another place.

Automatic detection of these smells cannot be done by analysing the program code as was described earlier; one has to *analyse the changes* that are made to the program to find out whether the program suffers from these smells. An interesting topic of future research is to investigate if the data in a configuration management system (esp. version managers such as CVS) could be used to check for these smells. Such an approach seems feasible since analysis of this type of data has already been done, for example, in the context of software evolution research [BKPS97, EGK+01].

8.5.4 Contributions

We have discussed the design considerations of a software inspection tool that is based on code smell detection. We have shown how code smells can be broken up into aspects that can be automatically detected. Furthermore, we have described how the code smell concept may be expanded to include coding standard conformance. We have investigated the feasibility of the described approach using a case study in which a prototype tool was developed and applied on a software system.

For the development of our prototype, jCosmo, we have implemented an extendable Java code smell detector which can be reused in other tools. We have extended

Rigi with an additional user interface that allows code smell browsing and visualization and developed several strategies for visually representing code smells within their program context.

Since the smell detection is fully automated, it can be tied into the development cycle providing continuous quality assessment and conformance checking. 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. Furthermore, since the analysis does not require the complete application, subsystems can be inspected before integration. This allows for incremental checking of large software systems which is especially interesting for distributed development.

Availability

The jCosmo tool is available under the GNU public license and a distribution can be downloaded from: http://www.cwi.nl/projects/renovate/javaQA/.

Acknowledgments

We would like to thank Serge Barthel of Epictoid for providing us with the source code and feature requests used in the CharToon case study.

CHAPTER 9

Refactoring Test Code

Two key aspects of extreme programming (XP) are unit testing and merciless refactoring. Given the fact that the ideal test code / production code ratio approaches 1:1, it is not surprising that unit tests are being refactored. We found that refactoring test code is different from refactoring production code in two ways: (1) a distinct set of bad smells is involved, and (2) improving test code involves additional test-specific refactorings. To share our experiences with other XP practitioners, we describe a set of bad smells that indicate trouble in test code, and a collection of test refactorings to remove these smells. The work presented in this chapter was published earlier as [DMBK01].

9.1 Introduction

"If there is a technique at the heart of *extreme programming* (XP), it is unit testing" [Bec99]. As part of their programming activity, XP developers write and maintain (white-box) unit tests continually. These tests are automated, written in the same programming language as the production code, considered an explicit part of the code, and put under revision control.

The XP process encourages writing a test class for every class in the system. Methods in these test classes are used to verify complicated functionality and unusual circumstances. Moreover, they are used to document code by explicitly indicating what the expected results of a method should be for typical cases. Last but not least, tests are added upon receiving a bug report to check for the bug and to check the bug fix [Bec00].

A typical test for a particular method includes four components: (1) code to set up the fixture (the data used for testing), (2) the call of the method, (3) a comparison of the actual results with the expected values, and (4) code to tear down the fixture. Writing tests is usually supported by frameworks such as *JUnit* [BG98].

The test code / production code ratio may vary from project to project, but is ideally considered to approach a ratio of 1:1. In our project we currently have a 2:3 ratio, although others have reported a lower ratio.[1] One of the cornerstones of XP is that having many tests available helps the developers to overcome their fear for change: the tests will provide immediate feedback if the system gets broken at a critical place. The down-side of having many tests, however, is that changes in functionality will typically involve changes in the test code as well. The more test code we get, the more important it becomes that this test code is as easily modifiable as the production code.

The key XP practice to keep code flexible is "refactor mercilessly": transforming the code in order to bring it in the simplest possible state. To support this, a catalog of "code smells" and a wide range of refactorings is available, varying from simple modifications up to ways to introduce design patterns systematically in existing code [Fow99].

When trying to apply refactorings to the test code of our project we discovered that refactoring test code is different from refactoring production code. Test code has a distinct set of smells, dealing with the ways in which test cases are organized, how they are implemented, and how they interact with each other. Moreover, improving test code involves a mixture of refactorings from [Fow99] specialized to test code improvements, as well as a set of additional refactorings, involving the modification of test classes, ways of grouping test cases, and so on.

The goal of this chapter is to share our experience in improving our test code with other XP practitioners. To that end, we describe a set of *test smells* indicating trouble in test code, and a collection of *test refactorings* explaining how to overcome some of these problems through a simple program modification.

This chapter assumes some familiarity with the *x*Unit framework [BG98] and refactorings as described by Fowler [Fow99]. We will refer to refactorings described in Fowler's book using the format *Name (F:page#)* and to our test specific refactorings described in section 9.3 using the format *Name* (#).

9.2 Test Code Smells

This section gives a overview of bad code smells that are specific for test code.

Smell 1: *Mystery Guest*

When a test uses external resources, such as a file containing test data, the test is no longer self contained. Consequently, there is not enough information to understand the tested functionality, making it hard to use that test as documentation.

Moreover, using external resources introduces hidden dependencies: if some force changes or deletes such a resource, tests start failing. Chances for this increase when more tests use the same resource.

[1] This project started a year ago and involves the development of a product called DocGen [DKM01]. Development is done by a small team of five people using XP techniques. Code is written in Java and we use the JUnit framework for unit testing.

The use of external resources can be eliminated using the refactoring *Inline Resource* (1). If external resources are needed, you can apply *Setup External Resource* (2) to remove hidden dependencies.

Smell 2: *Resource Optimism*

Test code that makes optimistic assumptions about the existence (or absence) and state of external resources (such as particular directories or database tables) can cause non-deterministic behavior in test outcomes. The situation in which tests run fine at one time and fail miserably another time is not a situation you want to find yourself in. Use *Setup External Resource* (2) to allocate and/or initialize all resources that are used.

Smell 3: *Test Run War*

Such wars arise when the tests run fine as long as you are the only one testing but fail when more programmers run them. This is most likely caused by resource interference: some tests in your suite allocate resources, such as temporary files, that are also used by others. Apply *Make Resource Unique* (3) to overcome interference.

Smell 4: *General Fixture*

In the *JUnit* framework a programmer can write a setUp method that will be executed before each test method to create a fixture for the tests to run in.

Things start to smell when the setUp fixture is too general and different tests access only part of the fixture. Such setUps are harder to read and understand and may make tests run more slowly (because they do unnecessary work). The danger of having tests that take too much time to complete is that testing starts interfering with the rest of the programming process and programmers eventually may not run the tests at all.

The solution is to use setUp only for that part of the fixture that is shared by all tests using Fowler's *Extract Method (F:110)* and put the rest of the fixture in the method that uses it using *Inline Method (F:117)*. If, for example, two different groups of tests require different fixtures, consider setting these up in separate methods that are explicitly invoked for each test, or spin off two separate test classes using *Extract Class (F:149)*.

Smell 5: *Eager Test*

When a test method checks several methods of the object to be tested, it is hard to read and understand, and therefore more difficult to use as documentation. Moreover, it makes tests more dependent on each other and harder to maintain.

The solution is simple: separate the test code into test methods that test only one method using Fowler's *Extract Method (F:110)*, using a meaningful name highlighting the purpose of the test. Note that splitting into smaller methods can slow down the tests because of increased setup and teardown overhead.

Smell 6: *Lazy Test*

This occurs when several test methods check the same method *using the same fixture* (but for example check the values of different instance variables). Such tests often only have meaning when we consider them together, so they are easier to use when joined using *Inline Method (F:117)*.

Smell 7: *Assertion Roulette*

"Guess what's wrong?" This smell comes from having a number of assertions in a test method that have no explanation. If one of the assertions fails, you do not know which one it is. Use *Add Assertion Explanation* (5) to remove this smell.

Smell 8: *Indirect Testing*

A test class is supposed to test its counterpart in the production code. It starts to smell when a test class contains methods that actually perform tests on other objects (for example because there are references to them in the class that is to be tested). Such indirection can be moved to the appropriate test class by applying *Extract Method (F:110)* followed by *Move Method (F:142)* on that part of the test. The fact that this smell arises also indicates that there might be problems with data hiding in the production code.

Note that opinions differ on indirect testing. Some people do not consider it a smell but a way to guard tests against changes in the "lower" classes. We feel that there are more losses than gains to this approach: it is much harder to test anything that can break in an object from a higher level. Moreover, understanding and debugging indirect tests is much harder.

Smell 9: *For Testers Only*

When a production class contains methods that are only used by test methods, these methods either are not needed and can be removed or are only needed to set up a fixture for testing. Depending on the functionality of those methods, you may not want them in production code where others can use them. If this is the case, apply *Extract Subclass (F:330)* to move these methods from the class to a (new) subclass in the test code, and use that subclass to perform the tests on. You will often find that these methods have names or comments stressing that they should only be used for testing.

Fear of this smell may lead to another undesirable situation: a class without a corresponding test class. This happens when a developer does not know how to test the class without adding methods that are specifically needed for the test and does not want to pollute the production class with test code. Creating a separate subclass helps to deal with this problem.

Smell 10: *Sensitive Equality*

It is fast and easy to write equality checks using the toString method. A typical way is to compute an actual result and map it to a string, which is then compared to a string literal representing the expected value. Such tests, however may depend on many irrelevant details, such as commas, quotes, and spaces. Whenever the toString method for an object is changed, tests start failing. The solution is to replace toString equality checks by real equality checks using *Introduce Equality Method* (6).

Smell 11: *Test Code Duplication*

Test code may contain undesirable duplication. In particular the parts that set up test fixtures are susceptible to this problem. Solutions are similar to those for normal code duplication as described by Fowler [Fow99, p. 76]. The most common case for test code is duplication of code in the same test class. This can be removed using *Extract Method (F:110)*. For duplication across test classes, it may be helpful to mirror the class hierarchy of the production code into the test class hierarchy. A word of caution however: moving duplicated code from two separate classes to a common class can introduce (unwanted) dependencies between tests.

A special case of code duplication is *test implication*: test *A* and *B* cover the same production code, and *A* fails if and only if *B* fails. A typical example occurs when the production code gets refactored: before this refactoring, *A* and *B* covered different code, but afterwards they deal with the same code and it is not necessary anymore to maintain both tests.

9.3 Refactorings

Bad smells seem to arise more often in production code than in test code. The main reason for this is that production code is adapted and refactored more frequently, allowing these smells to escape.

One should not, however, underestimate the importance of having fresh test code. Especially when new programmers are added to the team or when complex refactorings need to be performed, clear test code is invaluable. To maintain this freshness, test code also needs to be refactored.

We define *test refactorings* as changes (transformations) of test code that: (1) do not add or remove test cases, and (2) make test code more readable, understandable, and maintainable.

The production code can be used as a (simple) test case for the refactoring: If a test for a piece of code succeeds before the test refactoring, it should also succeed after the refactoring (and no, replacing all test code by `assert(true)` is not considered a valid refactoring). This obviously also means that you should not modify production code while refactoring test code (similar to not changing tests when refactoring production code).

While working on our test code, we encountered the following refactorings:

Refactoring 1: *Inline Resource*

To remove the dependency between a test method and some external resource, we incorporate that resource into the test code. This is done by setting up a fixture in the test code that holds the same contents as the resource. This fixture is then used instead of the resource to run the test. A simple example of this refactoring is putting the contents of a file that is used into some string in the test code.

If the contents of the resource are large, chances are high that you are also suffering from *Eager Test* (5) smell. Consider conducting *Extract Method (F:110)* or *Reduce Data* (4) refactorings.

Refactoring 2: *Setup External Resource*

If it is necessary for a test to rely on external resources, such as directories, databases, or files, make sure the test that uses them explicitly creates or allocates these resources before testing, and releases them when done (take precautions to ensure the resource is also released when tests fail).

Refactoring 3: *Make Resource Unique*

A lot of problems originate from the use of overlapping resource names, either between different tests runs done by the same user or between simultaneous test runs done by different users. Such problems can easily be prevented (or repaired) by using unique identifiers for all resources that are allocated, for example by including a timestamp. When you also include the name of the test responsible for allocating the resource in this identifier, you will have less problems finding tests that do not properly release their resources.

Refactoring 4: *Reduce Data*

Minimize the data that is setup in fixtures to the bare essentials. This offers two advantages: (1) it makes them more suitable as documentation, and (2) your tests will be less sensitive to changes.

Refactoring 5: *Add Assertion Explanation*

Assertions in the JUnit framework have an optional first argument to give an explanatory message to the user when the assertion fails. Testing becomes much easier when you use this message to distinguish between different assertions that occur in the same test. Maybe this argument should not have been optional...

Refactoring 6: *Introduce Equality Method*

If an object structure needs to be checked for equality in tests, add an implementation for the "equals" method for the object's class. You then can rewrite the tests that use string equality to use this method. If an expected test value is represented only as a string, explicitly construct an object containing the expected value, and use the new equals method to compare it to the actually computed object.

9.4 Related Work

Fowler [Fow99] presents a large set of bad smells and the refactorings that can be used to remove them. The difference between his work and ours is that we focus on smells and refactorings that are typical for test code whereas his book focuses more on production code. The role of unit tests in [Fow99] is also geared more toward proving that a refactoring didn't break anything than to being used as documentation of the production code.

Instead of focusing on cleaning test code that already has bad smells, Schneider [Sch00] describes how to prevent these smells right from the start by discussing a number of best practices for writing tests with JUnit.

The introduction of Mock Objects [MFC01] is another possibility for refactoring more complex tests. With this technique, one replaces parts of the production code with dummy implementations that both emulate real functionality and enforce assertions about the behavior of the code. This allows the tester to focus on the concrete code that has to be tested without having to deal with all surrounding code and the side effects that it may cause.

The C2 Wiki contains some discussion on the decay of unit test quality and practice as time proceeds,[2] and on the maintenance of broken unit tests.[3] Opinions vary between repairing broken unit tests, deleting them completely, and moving them to another class in order to make them less exposed to changes (which may lead to our *Indirect Testing* (8) smell).

9.5 Conclusions

In this chapter, we have looked at test code from the perspective of refactoring. While working on our XP project, we observed that the quality of the test code was not as high as the production code. Test code was not refactored as mercilessly as our production code, following Fowler's advice that it is okay to copy and edit test code, trusting our ability to refactor out truly common items later [Fow99, p. 102]. When at a later stage we started to refactor test code more intensively, we discovered that test code has its own set of problems (which we translated into smells) as well as its own repertoire of solutions (which we formulated as test refactorings).

The contributions of this chapter are the following:

- We have collected a series of *test smells* that help developers to identify weak spots in their test code;

[2] http://c2.com/cgi/wiki?TwoYearItch
[3] http://c2.com/cgi/wiki?RefactorBrokenUnitTests

- We have composed a set of specific *test refactorings* enabling developers to make improvements to their test code in a systematic way.
- For each smell we have given a solution, using either a potentially specialized variant of an existing refactoring from [Fow99] or one of the dedicated test refactorings.

The purpose of this chapter is to share our experience in refactoring test code of our ongoing XP project with other XP practitioners. We believe that the resulting smells and refactorings provide a valuable starting point for a larger collection based on a broader set of projects. Therefore, we would like to invite readers interested in further discussion on this topic to the C2 Wiki.[4]

An open question is how test code refactoring interacts with the other XP practices. For example, the presence of test code smells may indicate that your production code has some bad smells. So trying to refactor test code may indirectly lead to improvements in production code. Furthermore, refactoring test code may reveal missing test cases. Adding those to your framework will lead to a more completer test coverage of the production code. Another question is at what moments in the XP process test refactorings should be applied. In short, the precise interplay between test refactoring and the XP practices is a subject of further research.[5]

[4] http://c2.com/cgi/wiki?RefactoringTestCode

[5] In the next chapter (Chapter 10) we will revisit the video store example from Fowler's refactoring book [Fow99] and look in more detail into these issues.

CHAPTER 10

The Video Store Revisited — Thoughts On Refactoring and Testing

Testing and refactoring are core activities in extreme programming (XP). In principle, they are separate activities where the tests are used to verify that refactorings do not change behavior of the system. In practice however, they can become intertwined when refactorings invalidate tests. This chapter explores the precise relationship between the two. First, we identify which of the published refactorings affect the test code. Second, we observe that if test-first design is a way to arrive at well-designed code, **"test-first refactoring"** *is a way to arrive at a better design for existing code. Third, some refactorings improve testability, and should therefore be followed by improvements of the test code. To emphasize this, we propose the notion of* **"refactoring session"** *which includes changes to the code followed by changes to the tests. To guide the developer in the steps to take, we propose to extend the description of the mechanics of individual refactorings with consequences for the corresponding test code. The work presented in this chapter was published earlier as [DM02].*

10.1 Introduction

Two key activities in extreme programming (XP) are testing and refactoring. In this chapter, we explore the relationship between these two.

In XP, tests are fully automated, self-checking the validity of their outcome. Besides for checking correct behavior, tests are intended for *documentation* purposes. A test case is a simple scenario with a known outcome, and can be used to understand the code being tested. Since the tests are required to be run upon every change, their documentation value is guaranteed to remain up to date [Deu01]. Code development in XP is done through *test-first design*: Structuring the test cases guides the design of the production code.

Extreme programmers improve the design of the system through frequent refactoring. Refactorings improve the internal structure of the code without changing its external behavior. This is done by removing duplication, simplification, making code easier to understand, and adding flexibility. "Without refactoring, the design of software will decay. Regular refactoring helps code retain its shape." [Fow99, p.55].

One of the dangers of refactoring is that a programmer unintentionally changes the system's behavior. Ideally, it can be verified that this did not happen by checking that all the tests pass after refactoring. In practice however, there are refactorings that will invalidate tests (e.g., when a method is moved to another class and the test still expects it in the original class).

In this chapter, we explore the relationship between unit testing and refactoring. In Section 10.2, we provide a classification of the refactorings described by Fowler [Fow99], identifying exactly which of the refactorings affect class interfaces, and which therefore require changes in the test code as well. In Section 10.3 we take the video store example from [Fow99], and assess the implications of each refactoring on the test code. In Section 10.4, we propose *test-driven refactoring*, which analyzes the test code in order to arrive at code level refactorings. In Section 10.5, we discuss the relationship between code-level refactorings and test-level refactorings. In Section 10.6 we integrate these results via the notion of a *refactoring session* which is a coherent set of steps resulting in refactoring of both the code and the tests. In Section 10.7 we present a summary and draw our conclusions.

10.2 Types of Refactoring

Refactoring a system should not change its observable behavior. Ideally, this is verified by ensuring that all the tests pass before and after a refactoring [Bec00, Fow99].

In practice, it turns out that such verification is not always possible: some refactorings restructure the code in such a way that not all the tests will pass after the refactoring. For example, refactoring can move a method to a new class while some tests still expect it in the original class (in that case, the code will probably not even compile). Nevertheless, we do not want to change the tests together with a refactoring since that will make them less trustworthy for validating correct behavior afterwards.

In the remainder of this section, we will look in more detail at the refactorings described by Fowler [Fow99] to analyse in which cases problems might arise because the original tests cannot be used after refactoring.

10.2.1 Taxonomy

If we start with the assumption that refactoring does not change the behavior of the system, then there is only one reason why a refactoring can break a test: *because the refactoring changes the interface that the test expects.* Note that the interface extends to all visible aspects of a class (fields, methods, and exceptions). This implies that one has to be careful with tests that directly inspect the fields of a class since these will more easily change during a refactoring.[1]

So, initially, we distinguish two types of refactoring: refactorings that do not change any interface of the classes in the system and refactorings that do change an interface. The first type of refactorings have no consequences for the tests: Since the interfaces are kept the same, tests that succeeded before refactoring will also succeed after refactoring (if the refactoring indeed preserves the tested behavior).

The second type of refactorings can have consequences for the tests since there might be tests that expect the old interface. Again, we can distinguish two situations:

Incompatible: the refactoring destroys the original interface. In this case, tests that rely on the old interface will fail, and in most cases, the code will not even compile. We'll have to take measures to re-enable these broken tests.

Backwards Compatible: the refactoring extends the original interface. In this case the tests keep running via the original interface and will pass if the refactoring preserves tested behavior. Depending on the refactoring, we might need to add more tests covering the extensions.

A number of incompatible refactorings that normally would destroy the original interface can be made into backwards compatible refactorings. This is done by extending the refactoring so it will retain the old interface, for example, using the Adapter pattern or simply via delegation. As a side-effect, the new interface will already partly be tested. Note that this is common practice when refactoring a published interface to prevent breaking of dependent systems. A disadvantage is that a larger interface has to be maintained but when delegation or wrapping was used, that should not be too much work. Furthermore, language features like deprecation can be used to signal that this part of the interface is outdated.

10.2.2 Classification

We have analyzed the refactorings in [Fow99] and divided them into the following classes:

A. *Composite:* The four big refactorings *Convert Procedural Design to Objects*, *Separate Domain from Presentation*, *Tease Apart Inheritance*, and *Extract Hierarchy* will change the original interface, but we will not consider them in more detail since they are performed as series of smaller refactorings.

B. *Compatible:* Refactorings that do not change the original interface. Refactorings in this class are listed in Figure 10.1.

C. *Backwards Compatible:* Refactorings that change the original interface and are inherently backwards compatible since they extend the interface. Refactorings in this class are listed in Figure 10.2.

[1] In fact, directly inspection of fields of a class, is a test smell that could better be removed on forehand as discussed in Chapter 9.

Replace Conditional with Polymorphism
Replace Inheritance with Delegation
Replace Method with Method Object
Remove Assignments to Parameters
Replace Delegation with Inheritance
Replace Data Value with Object
Replace Exception with Test
Introduce Explaining Variable
Change Reference to Value

Split Temporary Variable
Decompose Conditional
Preserve Whole Object
Introduce Null Object
Substitute Algorithm
Remove Control Flag
Introduce Assertion
Extract Class
Inline Temp

Change Bidirectional Association to Unidirectional
Replace Nested Conditional with Guard Clauses
Replace Magic Number with Symbolic Constant
Consolidate Duplicate Conditional Fragments

Figure 10.1: Compatible refactorings (type B).

Replace Delegation with Inheritance
Replace Inheritance with Delegation
Consolidate Conditional Expression
Replace Record with Data Class
Introduce Foreign Method
Replace Temp with Query
Pull Up Constructor Body
Duplicate Observed Data
Form Template Method

Self Encapsulate Field
Push Down Method
Extract Superclass
Push Down Field
Extract Interface
Pull Up Method
Extract Method
Pull Up Field

Figure 10.2: Backwards compatible refactorings (type C).

Change Unidirectional Association to Bidirectional
Replace Parameter with Explicit Methods
Replace Parameter with Method
Separate Query from Modifier
Introduce Parameter Object
Parameterize Method

Remove Middle Man
Remove Parameter
Rename Method
Add Parameter
Move Method

Figure 10.3: Refactorings that can be made backwards compatible (type D).

Replace Constructor with Factory Method	*Encapsulate Collection*
Replace Type Code with State/Strategy	*Encapsulate Downcast*
Replace Type Code with Subclasses	*Collapse Hierarchy*
Replace Error Code with Exception	*Encapsulate Field*
Replace Type Code with Class	*Extract Subclass*
Replace Subclass with Fields	*Hide Delegate*
Change Value to Reference	*Inline Method*
Introduce Local Extension	*Hide Method*
Replace Array with Object	*Inline Class*
Remove Setting Method	*Move Field*

Figure 10.4: Incompatible refactorings (type E).

D. *Make Backwards Compatible:* Refactorings that change the original interface and can be made backwards compatible by adapting the new interface to old interface. Refactorings in this class are listed in Figure 10.3.

E. *Incompatible:* Refactorings that change the original interface and are not backwards compatible (for example, because they change the types of classes that are involved). Refactorings in this class are listed in Figure 10.4.

Note that the refactorings *Replace Inheritance with Delegation* and *Replace Delegation with Inheritance* appear both in the *Compatible* and *Backwards Compatible* category since they can be of either class, depending on the actual case.

10.3 Revisiting the Video Store

In this section, we study the relationship between testing and refactoring using a well-known example of refactoring. We revisit the video store code used by Fowler [Fow99, Chapter 1], extending it with an analysis of what should be going on in the accompanying video store test code.

The video store class structure before refactoring is shown in Figure 10.5. It consists of a *Customer*, who is associated with a series of *Rentals*, each consisting of a *Movie* and an integer indicating the number of days the movie was rented. The key functionality

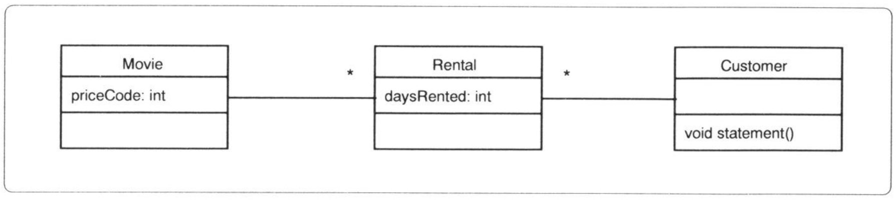


Figure 10.5: Classes before refactoring.

is in the Customer's *statement* method printing a customer's total rental cost. Before refactoring, this statement is printed by a single long method. After refactoring, the statement functionality is moved into appropriate classes, resulting in the structure of Figure 10.6 taken from [Fow99, p.51].

Fowler emphasizes the need to conduct refactorings as a sequence of small steps. At each step, you must run the tests in order to verify that nothing essential has changed. His testing approach is the following: "I create a few customers, give each customer a few rentals of various kinds of films, and generate the statement strings. I then do a string comparison between the new string and some reference strings that I have hand checked". Although Fowler doesn't list his test classes, this typically should look like the code in Figure 10.7.

Studying this string-based testing method, we can make the following observations:

- The setup is complicated, involving the creation of many different objects.
- The documentation value of the test is limited: it is hard to relate the computation of the charge of 4.5 for movie m1 to the way in which charges are computed for the actual movies rented (in this case a childrens and a regular movie, each with their own price computation).
- The tests are brittle. All test cases include a full statement string. When the format changes in just a very small way, all existing tests (!) must be adjusted, an error prone activity we would like to avoid.

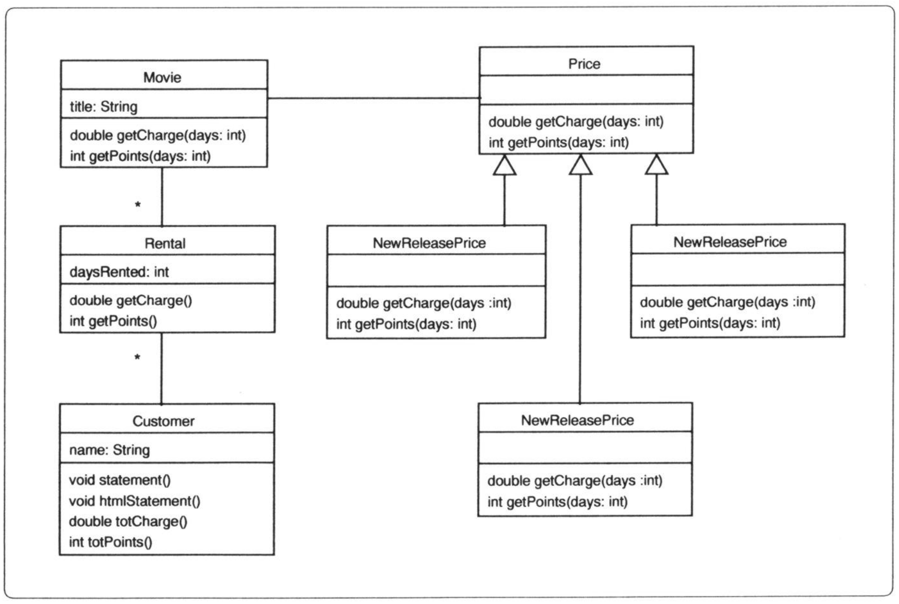


Figure 10.6: Class structure after refactoring.

In short, the poor structure of the long method necessarily leads to an equally poor structure of the test cases. From a testing perspective, we would like to be able to separate computations from report writing. The long statement method prohibits this: it needs to be refactored in order to be able to improve the testability of the code.

This way of reasoning naturally leads to the application of the *Extract Method* refactoring to the *statement* method. Fowler comes to the same conclusion, based on the need to write a new method printing a statement in HTML format. Methods to extract include *getCharge* for computing the charge of a rental, and *getPoints* for computing the "frequent renter points".

Extract Method is of type B, the *compatible* refactorings, so we can use our existing tests to check the refactoring. However, we have created new methods, for which we might like to add tests that document and verify their specific behavior. To write them, we can reuse the setup of movies, rentals, and customers used for testing the *statement* method. What we end up with is a number of smaller test cases specifically addressing either the charge or rental point computations.

Since the correspondence between test code and actual code is now much clearer

```
Movie m1 = new Movie("m1", Movie.CHILDRENS);
Movie m2 = new Movie("m2", Movie.REGULAR);
Movie m3 = new Movie("m3", Movie.NEW_RELEASE);
Rental r1 = new Rental(m1, 5);
Rental r2 = new Rental(m2, 7);
Rental r3 = new Rental(m3, 1);
Customer c1 = new Customer("c1");
Customer c2 = new Customer("c2");

public void setUp() {
  c1.addRental(r1);
  c1.addRental(r2);
  c2.addRental(r3);
}

public void testStatement1() {
  String expected =
    "Rental Record for c1\n" +
    "\tm1\t4.5\
    "\tm2\t9.5\n" +
    "Amount owed is 14.0\n" +
    "You earned 2 frequent renter points";

    assertEquals(expected, c1.statement());
}
```

Figure 10.7: Initial sample test code.

and better focused, we can apply white box testing, and use our knowledge of the structure of the code to determine the test cases needed. Thus, we see that the *getCharge* method to be tested distinguishes between 5 cases, and we make sure our tests cover these cases.

This has solved some of the problems. The tests are better understandable, more complete, much shorter, and less brittle. Unfortunately, we still have the complicated setup method. What we see is that the setup mostly involves rentals and movies, while the tests themselves are in the customer testing class. This is because the extracted method is in the wrong class: applying *Move Method* to *Rental* simplifies the set up for new test cases. Again we use our analysis of the test code to find refactorings in the production code.

The *Move Method* is of type D, refactorings that can be made backwards compatible by adding a wrapper method to retain the old interface. We add this wrapper so we can check the refactoring with our original tests. However, since the documentation of the method is in the test, and this documentation should be as close as possible to the method documented, we want to move the tests to the method's new location. Since there is no test class for Rental yet, we create it, and move the test methods for *getCharge* to it. Depending on whether the method was part of a published interface, we might want to keep the wrapper (for some time), or remove it together with the original test.

Fowler discusses several other refactorings, moving the charge and point calculations further down to the *Movie* class, replacing conditional logic by polymorphism in order to make it easier to add new movie types, and introducing the *state* design pattern in order to be able to change movie type during the life time of a movie.

When considering the impact on test cases of these remaining video store refactorings, we start to recognize a pattern:

- Studying the test code and the smells contained in it may help to identify refactorings to be applied at the production code;
- Many refactorings involve a change to the structure of the unit tests of well: in order to maintain the documenting value of these unit tests, they should be changed to reflect the structure of the code being tested.

In the next two sections, we take a closer look at these issues.

10.4 Test-Driven Refactoring

In *test-driven refactoring*, we try to use the existing test cases in order to determine the code-level refactorings. Thus, we study *test* code in order to find improvements to the *production* code.

This calls for a set of *code smells* that helps to find such refactorings. A first category is the set of existing code smells discussed in Fowler's book [Fow99]. Several of them, such as long method, duplicated code, long parameter list, and so on, apply to test code as well as they do to production code. In many cases solving them involves not just a change on the test code, but first of all a refactoring of the production code.

A second category of smells is the collection of *test smells* discussed in our earlier chapter on *refactoring test cases* (Chapter 9). In fact, in our movie example we encountered several of them already. Our uneasy feeling with the test case of Figure 10.7 is captured by the *Sensitive Equality* smell (Smell 10 of Chapter 9): comparing computed values to a string literal representing the expected value. Such tests depend on many irrelevant details, such as commas, quotes, tabs, and so on. This is exactly the reason the customer tests of Figure 10.7 become brittle.

Another *test smell* we encountered is called *Indirect Testing* (Smell 8 of Chapter 9): a test class contains methods that actually perform tests on other objects. Indirect tests make it harder to understand the relationship between test and production code. While moving the *getCharge* and *getPoints* methods in the class hierarchy (using *Move Method*), we also moved the corresponding test cases, in order to avoid *Indirect Testing*.

The test-driven perspective may lead to the formulation of additional test smells. For example, we observed that setting up the fixture for the CustomerTest was complicated. This indicates that the tests could be in the wrong class, or that the underlying business logic is not well isolated. Another smell could be that there are many test cases for a single method, indicating that the method is too complex.

Test-driven refactoring is a natural consequence of test-first design. Test-first design is a way to get a good design by thinking about test cases first when adding functionality. Test-driven refactoring is a way to improve your design by rethinking the way you structured your tests.

In fact, Beck's recent article on test-first design [Bec01] contains an interesting example that can be transferred to the refactoring domain. it involves testing the construction of a mortality table. His first attempt requires a complicated setup, involving separate "person" objects. He then rejects this solution as being overly complex for testing purposes, and proposes the construction of a mortality table with just an age as input. His example illustrates how test case construction guides design when building new code; Likewise, test case refactoring guides the improvement of design during refactoring.

10.5 Refactoring Test Code

In our study of the video store example, we saw that many refactorings on the code level can be completed by applying a corresponding refactoring on the test case level. For example, to avoid *Indirect Testing*, the refactoring *Move Method* should be followed by "*Move Test*". Likewise, in many cases *Extract Method* should be followed by "*Extract Test*". To retain the documentation value of the unit tests, their structure should be in sync with the structure of the source code.

In our opinion, it makes sense to extend the existing descriptions of refactorings with suggestions on what to do with the corresponding unit tests, for example in the "mechanics" part.

The topic of refactoring test code is discussed extensively in the previous chapter (Chapter 9). An issue of concern when changing test code is that we may "loose" test cases. When refactoring production code, the availability of tests safeguards us from accidentally loosing code, but this is not the case when modifying tests. One solution could be to apply *mutation testing* using a tool such as Jester [Moo01b]. Jester automatically makes changes to conditions and literals in Java source code. If the code is well-tested, such changes should lead to failing tests. Running Jester before and after test case refactorings should help to ensure that the changes did not harm the tests.

10.6 Refactoring Sessions

The meaningful unit of refactoring is a sequence of steps involving changes to both the code base and the test base. We propose the notion of a *refactoring session* to capture such a sequence. It consists of the following steps:

1. Detect *smells* in the code or test code that need to be fixed. In test-driven refactoring, the test set is the starting point for finding such smells.
2. Identify candidate refactorings addressing the smell.
3. Ensure that all existing tests run.
4. Apply the selected refactoring to the code. Provide a backwards compatible interface if the refactoring falls in category D. Only change the associated test classes when the refactoring falls in category E.
5. Ensure that all existing tests run. Consider applying mutation testing to assess the coverage of the test cases.
6. Apply the testing counterpart of the selected refactoring.
7. Ensure that the modified tests still run. Check that the coverage has not changed.
8. Extend the test cases now that the underlying code has become easier to test.
9. Ensure the new tests run.

The integrity of the code is ensured since (1) all tests are run between each step; (2) each step changes either code or tests, but never both at the same time (unless this is impossible).

10.7 Conclusions

In this chapter we have taken a close look at the interplay between testing and refactoring. We consider the following as our most important contributions:

- We have analyzed which of the documented refactorings necessarily affect the test code. It turns out that the majority of the refactorings are in category D (requiring explicit actions to keep the interface compatible) and E (necessarily requiring a change to the test code).

- We have studied Fowler's video store example from the point of view of unit tests included for documentation purposes. We have shown the test case implications of each refactoring conducted.
- We have proposed the notion of *test-driven refactoring*, which uses the existing test cases as the starting point for finding suitable code level refactorings.
- We have argued for the need to extend the descriptions of refactorings with a section on their implications on the corresponding test code. If the test are to maintain their documentation value, they should be kept in sync with the structure of the code.
- We have proposed the notion of a *refactoring session*, capturing a coherent series of separate steps involving changes to both the production code and the test code.

Part IV
Epilogue

Chapter 11

Conclusions

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 inferencing 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.

Bibliography

[AB01] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded Java programs. In D. Grant, editor, *Proceedings of the 13th Australian Software Engineering Conference (ASWEC 2001)*, pages 68–75. IEEE Computer Society Press, 2001.

[Abn96] S. Abney. Partial parsing via finite-state cascades. In *Proceedings of the ESSLLI '96 Robust Parsing Workshop*, 1996.

[AFFS98] A. Aiken, S. Fähndrich, S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Second International Workshop on Types in Compilation (TIC'98)*, volume 1473 of *Lecture Notes in Computer Science*, pages 78–96. Springer-Verlag, 1998.

[AKW88] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language.* Addison-Wesley, 1988.

[AP92] P. Arthur and R. Passini. *Wayfinding – People, Signs, and Architecture.* McGraw-Hill, 1992.

[Asv96] P. R. J. Asveld. A bibliography on fuzzy automata, grammars and languages. *BEATCS*, 58:187–196, 1996.

[BA96] S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis.* IEEE Computer Society Press, 1996.

[BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, 1998.

[BDH$^+$01] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of Compiler Construction 2001*, LNCS, 2001.

[BDK+96] M. G. J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 9–18. Springer-Verlag, 1996.

[Bec99] K. Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10):70–77, October 1999.

[Bec00] K. Beck. *Extreme Programming Explained. Embrace Change.* Addison-Wesley, 2000.

[Bec01] K. Beck. Aim, fire: Kent Beck on test-first design. *IEEE Software*, 18(5):87–89, 2001.

[Ben90] K. H. Bennett. An introduction to software maintenance. *Information and Software Technology*, 12(4):257–264, 1990.

[BG98] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *JAva Report*, 3(7):51–56, 1998.

[BHB99] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *Proceedings of the 21th International Conference on Software Engineering (ICSE-21)*, pages 555–563. ACM Press, 1999.

[BHK89] J. A. Bergstra, J. Heering, and P. Klint. The Algebraic Specification Formalism ASF. In *Algebraic Specification*, chapter 1, pages 1–66. ACM Press & Addison-Wesley, 1989.

[Bis92] W. Bischofberger. Sniff—a pragmatic approach to a C++ programming environment. In *Proceedings of the 1992 USENIX C++ Conference*, pages 67–82, August 1992.

[BKPS97] T. Ball, J. Kim, A. Porter, and H. Siy. If your version control system could talk. In *Proceedings of the Workshop on Process Modelling and Empirical Studies of Software Engineering*, May 1997.

[BKV97] M. G. J. van den Brand, P. Klint, and C. Verhoef. Re-engineering needs generic programming language technology. *Sigplan Notices*, 32(2):54–61, 1997.

[BKV01] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. Technical Report SEN-R0121, CWI, July 2001.

[BL76] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[BMIM98] W. J. Brown, R. C. Malveau, H. W. S. McCormick III, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, 1998.

[BMW93] T. Biggerstaff, B. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering (ICSE-15)*, pages 482–498. ACM Press, may 1993.

[Bra95] S. Brand. *How Buildings Learn: What Happens After They're Built.* Penguin, New York, 1995.

[Bro83] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[Bro91] P. Brown. Integrated hypertext and program understanding tools. *IBM Systems Journal*, 30(3):363–392, 1991.

[BS95] M. L. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, interfaces and the incremental approach.* Morgan Kaufman Publishers, 1995.

[BSL98] C. O. Braga, A. v. Staa, and J. C. S. P. Leite. Documentu: A flexible architecture for documentation production based on a reverse-engineering strategy. *Journal of Software Maintenance: Research and Practice*, 10(4):279–303, 1998.

[BSV97a] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE'97)*, pages 144–155. IEEE Computer Society Press, October 1997.

[BSV97b] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Obtaining a COBOL grammar from legacy code for reengineering purposes. In A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing, Amsterdam, September 1997. Springer-Verlag.

[BV00] M. G. J. van den Brand and J. J. Vinju. Rewriting with layout. In *Proceedings of the First International Workshop on Rule-Based Programming (RULE'2000)*, September 2000.

[Car83] J. Carroll. An island parsing interpreter for the full augmented transition network formalism. In *Proceedings of the 1st Conference of the European Chapter of the Association for Computational Linguistics*, pages 101–105, 1983.

[Car97] L. Cardelli. Type systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997.

[CC90] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[CC00] A. Cox and C. Clarke. A comparitive evaluation of techniques for syntactic level source code analysis. In *Proceedings of the 7th Asia-Pacific Software Engineering Conference*, December 2000.

[CCM96] G. Canfora, A. Cimitile, and M. Munro. An improved algorithm for identifying objects in code. *Software Practice and Experience*, 26(1):25–48, 1996.

[CDDF99] A. Cimitile, A. De Lucia, G. A. Di Lucca, and A. R. Fasolino. Identifying objects in legacy systems using design metrics. *Journal of Systems and Software*, 44(3):199–211, 1999.

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[CFKW95] Y.-F. Chen, G. S. Fowler, E. Koutsofios, and R. S. Wallach. Ciao: A graphical navigator for software and document repositories. In G. Caldiera and K. Bennett, editors, *Proceedings of the 1995 International Conference on Software Maintenance (ICSM'95)*, pages 66–75. IEEE Computer Society Press, 1995.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[Com00] Compuware. *UNIFACE: An Environment for Building Complex, Business-Critical Applications*, September 2000. White paper.

[Cor89] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[CR99] S. Chandra and T. Reps. Physical type checking for C. In *Proceedings of the 1999 ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'99)*, pages 66–75. ACM Press, September 1999. SIGSOFT Software Engineering Notes, 24(5).

[CTJ$^+$94] X. P. Chen, W. T. Tsai, J. K. Joiner, H. Gandamaneni, and J. Sun. Automatic variable classification for COBOL programs. In *Proceedings of the 18th Annual International Computer Software and Applications Conference (COMPSAC'94)*, pages 432–437, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[DCG+99] P. Devanbu, Y-F. Chen, E. Gansner, H. Müller, and J. Martin. CHIME: Customizable hyperlink insertion and maintenance engine for software engineering environments. In *Proceedings of the 21th International Conference on Software Engineering (ICSE-21)*, pages 473–482. ACM Press, 1999.

[DDL99] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering approach combining metrics and program visualization. In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE'99)*, pages 175–186, October 1999.

[Deu01] A. van Deursen. Program comprehension risks and benefits in extreme programming. In E. Burd, P. Aiken, and R. Koschke, editors, *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 176–185. IEEE Computer Society Press, October 2001.

[Dev99] P. Devanbu. Genoa - a customizable, front-end retargetable source code analysis framework. *ACM Transactions on Software Engineering and Methodology*, 8(2):177–212, April 1999.

[DHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[DK83] T. A. van Dijk and W. Kintsch. *Strategies of Discourse Comprehension.* Academic Press, 1983.

[DK98] A. van Deursen and T. Kuipers. Rapid system understanding: Two COBOL case studies. In S. Tilley and G. Visaggio, editors, *Proceedings of the 6th International Workshop on Program Comprehension (IWCP'98)*, pages 90–98. IEEE Computer Society Press, 1998.

[DK99a] A. van Deursen and T. Kuipers. Building documentation generators. In *Proceedings of the 1999 International Conference on Software Maintenance (ICSM'99)*, pages 40–49. IEEE Computer Society Press, 1999.

[DK99b] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 21th International Conference on Software Engineering (ICSE-21)*, pages 246–255. ACM Press, 1999.

[DKM00] A. van Deursen, T. Kuipers, and L. Moonen. Arrangement and method for a documentation generation system. U. S. Patent Application, August 2000.

[DKM01] A. van Deursen, T. Kuipers, and L. Moonen. Legacy to the extreme. In M. Marchesi and G. Succi, editors, *eXtreme Programming Examined*, pages 501–514. Addison-Wesley, May 2001.

[DM98] A. van Deursen and L. Moonen. Type inference for COBOL systems. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE'98)*, pages 220–230. IEEE Computer Society Press, October 1998. Included as Chapter 4 of this thesis.

[DM99] A. van Deursen and L. Moonen. Understanding COBOL systems using types. In *Proceedings of the 7th International Workshop on Program Comprehension (IWCP'99)*, pages 74–83. IEEE Computer Society Press, 1999. An extended version appeared as [DM01].

[DM00] 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 Press, November 2000. Included as Chapter 7 of this thesis.

[DM01] A. van Deursen and L. Moonen. An empirical study into cobol type inferencing. *Science of Computer Programming*, 40(2–3):189–211, July 2001. This is an extended version of [DM99]. It is included as Chapter 5 of this thesis.

[DM02] A. van Deursen and L. Moonen. The video store revisited - thoughts on refactoring and testing. In M. Marchesi and G. Succi, editors, *Proceedings of the 3rd International Conference on Extreme Programming and Agile Processes in Software Engineering (XP2002)*, May 2002. Included as Chapter 10 of this thesis.

[DMBK01] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In M. Marchesi and G. Succi, editors, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, May 2001. Included as Chapter 9 of this thesis.

[DWQ97] A. van Deursen, S. Woods, and A. Quilici. Program plan recognition for year 2000 tools. In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE'97)*, pages 124–133. IEEE Computer Society Press, October 1997.

[EGK+01] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[EHM+99] P. H. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. Sorensen, and M. Tofte. Anno Domini: From type theory to Year 2000 conversion tool. In *Proceedings of the 26th Symposium on Principles of Programming Languages (POPL'99)*. ACM Press, 1999.

[EM02] 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 Press, October 2002. Included as Chapter 8 of this thesis.

[Fag76] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[FB98] M. J. Fyson and C. Boldyreff. Using application understanding to support impact analysis. *Software Maintenance: Research and Practice*, 10:93–110, 1998.

[FKO98] L. Feijs, R. Krikhaar, and R. van Ommering. A relational approach to support software architecture analysis. *Software Practice and Experience*, 28(4):371–400, 1998.

[Flo02] G. Florijn. RevJava – Design critiques and architectural conformance checking for Java software. White Paper. SERC, the Netherlands, 2002.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FRS94] H. Fergen, P. Reichelt, and K. P. Schmidt. Bringing objects into COBOL: MOORE - a tool for migration from COBOL85 to object-oriented COBOL. In *Proceedings of the Conference on Technology of Object-Oriented Languages and Systems (TOOLS-14)*, pages 435–448. Prentice-Hall, 1994.

[GAM96] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proceedings of the 4th International Workshop on Program Comprehension (IWCP'96)*. IEEE Computer Society Press, 1996.

[GG93] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GKNV93] E. R. Gansner, E. Koutsofios, S. North, and K-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[GL91] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[GL96] J. M. Gravley and A. Lakhotia. Identifying enumeration types modeled with symbolic constants. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'96)*, pages 227–236. IEEE Computer Society Press, October 1996.

[Gru73] Jozef Gruska. Descriptional complexity of context-free languages. In *Proceedings of the Mathematical Foundations of Computer Science*, pages 71–83, 1973.

[GW99] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1999.

[Han97] J. Han. Supporting impact analysis and change propagation in software engineering environments. In *Proceedings of the 8th International Workshop on Software Technology and Engineering Practice (STEP'97)*, pages 172–182, July 1997.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF—Reference Manual. *Sigplan Notices*, 24(11):43–75, 1989.

[Hol98] R. Holt. Structural manipulations of software architecture using Tarski relational algebra. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE'98)*, pages 210–219. IEEE Computer Society Press, October 1998.

[HP96] J. Hart and A. Pizzarello. A scaleable, automated process for year 2000 system correction. In *Proceedings of the 18th International Conference on Software Engineering (ICSE-18)*, pages 475–484. ACM Press, 1996.

[HWS00] R. Holt, A. Winter, and A. Schürr. GXL: Towards a standard exchange format. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE 2000)*, pages 162–171, November 2000.

[ISO00] ISO. *Programming language COBOL: Proposed Revision of ISO 1989:1985*. International Standardization Organization, 2000. Committee Draft 1.8, ISO/IEC CD 1.8 1989 : yyyy(E).

[Joh78] S. C. Johnson. Lint, a C program checker. In *Unix Programmer's Manual*, volume 2A, chapter 15, pages 292–303. Bell Laboratories, 1978.

[Jon98] C. Jones. *Estimating Software Costs*. McGraw-Hill, 1998.

[KC99] R. Kazman and J. Carrière. Playing detective: Reconstructing software architecture from available evidence. *Journal of Automated Software Engineering*, 6:107–138, 1999.

[Kea91] S. Kearns. Tlex. *Software Practice and Experience*, 21(8):805–821, August 1991.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[KM00] T. Kuipers and L. Moonen. Types and concept analysis for legacy systems. In *Proceedings of the 8th International Workshop on Program Comprehension (IWCP 2000)*. IEEE Computer Society Press, June 2000. Included as Chapter 6 of this thesis.

[KMUO98] K. Kawabe, A. Matsuo, S. Uehara, and A. Ogawa. Variable classification technique for software maintenance and application to the year 2000 problem. In P. Nesi and F. Lehner, editors, *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98)*, pages 44–50. IEEE Computer Society Press, 1998.

[Kop97] R. Koppler. A systematic approach to fuzzy parsing. *Software Practice and Experience*, 27(6):637–649, 1997.

[KV01] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In *Proceedings of the Workshop on Language Descriptions, Tools and Applications (LDTA 2001)*, pages 28–52, 2001. Electronic Notes in Theoretical Computer Science, volume 44.

[KW99] B. Kullbach and A. Winter. Querying as an enabling technology in software reengineering. In *Proceedings of the 3th European Conference on Software Maintenance and Reengineering (CSMR'99)*, pages 42–50. IEEE Computer Society Press, 1999.

[Lak97] A. Lakhotia. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software*, pages 211–231, March 1997.

[Leh80a] 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.

[Leh80b] M. M. Lehman. Programs, life cycles and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980. Special Issue on Software Engineering.

[Leh97] M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the European Workshop on Software Process Technology EWSPT'96*, volume 1149 of *LNCS*, pages 108–124. Springer-Verlag, 1997.

[Let86] S. Letovsky. Cognitive processes in program comprehension. In E. Solloway and S. Iyengar, editors, *Proceedings of the First Workshop on Empirical Studies of Programmers*, pages 58–79. Ablex Publishing, 1986.

[LM93] J. P. Loyall and S. A. Mathisen. Using dependence analysis to support the software maintenance process. In *Proceedings of the 1993 International Conference on Software Maintenance (ICSM'93)*, pages 282–291. IEEE Computer Society Press, 1993.

[LPR98] M. M. Lehman, D. E. Perry, and J. F. Ramil. Implications of evolution metrics on software maintenance. In T. M. Koshgoftaar and K. Bennett, editors, *Proceedings of the 1998 International Conference on Software Maintenance (ICSM'98)*, pages 208–217. IEEE Computer Society Press, 1998.

[LR95] D. Ladd and J. Ramming. A*: A language for implementing language processors. *IEEE Transactions on Software Engineering*, 21(11):894–901, 1995.

[LS75] M. Lesk and E. Schmidt. Lex—a lexical analyser generator. Computer Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ, USA, October 1975.

[LS97] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering (ICSE-19)*, pages 349–359. ACM Press, 1997.

[LW90] S. S. Liu and N. Wilde. Identifying objects in a conventional procedural language: An example of data design recovery. In *Proceedings of the 1990 International Conference on Software Maintenance (ICSM'90)*, pages 266–271. IEEE Computer Society Press, 1990.

[Lyn60] K. Lynch. *The Image of The City*. MIT Press, 1960.

[LZ69] E. T. Lee and L. A. Zadeh. Note on fuzzy languages. *Information Sciences*, 1(4):421–434, 1969.

[McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[MFC01] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: Unit testing with mock objects. In M. Marchesi and G. Succi, editors, *eXtreme Programming Examined*, pages 287–301. Addison-Wesley, May 2001.

[MGHDM95] E. Merlo, J. F. Girard, L. Hendren, and R. De Mori. Multi-valued constant propagation analysis for user interface reengineering. *International Journal of Software Engineering and Knowledge Engineering*, 5(1), March 1995.

[Mil56] G. A. Miller. The magical number seven plus or minus two: some limits on our capacity for processing information. *Psychological Review*, 63:81–97, 1956.

[MN96] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, 1996.

[MNB$^+$94] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to reengineer legacy systems. *Communications of the ACM*, 37(5):58–70, 1994. Special issue on reverse engineering.

[Moo97] L. Moonen. A generic architecture for data flow analysis to support reverse engineering. In A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing, Amsterdam, September 1997. Springer-Verlag.

[Moo01a] L. Moonen. Generating robust parsers using island grammars. In E. Burd, P. Aiken, and R. Koschke, editors, *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 13–22. IEEE Computer Society Press, October 2001. Included as Chapter 2 of this thesis.

[Moo01b] I. Moore. Jester — a JUnit test tester. In M. Marchesi and G. Succi, editors, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, pages 84–87, May 2001.

[Moo02] L. Moonen. Lightweight impact analysis using island grammars. In *Proceedings of the 10th International Workshop on Program Comprehension (IWCP 2002)*. IEEE Computer Society Press, June 2002. Included as Chapter 3 of this thesis.

[MOTU93] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, 1993.

[MV95] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, pages 44–55, August 1995.

[MV96] A. von Mayrhauser and A. M. Vans. On the role of hypotheses during opportunistic understanding while porting large scale software. In *Proceedings of the 4th International Workshop on Program Comprehension (IWCP'96)*, pages 68–77. IEEE Computer Society Press, 1996.

[MV97] A. von Mayrhauser and A. M. Vans. Hypothesis-driven understanding processes during corrective maintenance of large scale software. In *Proceedings of the 1997 International Conference on Software Maintenance (ICSM'97)*, pages 12–20. IEEE Computer Society Press, 1997.

[NBOS99] M. G. Nanda, P. Bhaduri, S. Oberoi, and A. Sanyal. An application of compiler technology to the year 2000 problem. *Software Practice and Experience*, 29(4):359–377, 1999.

[NK95] P. Newcomb and G. Kottik. Reengineering procedural into object-oriented systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE'95)*, pages 237–249. IEEE Computer Society Press, October 1995.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[OJ97] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 19th International Conference on Software Engineering (ICSE-19)*. ACM Press, 1997.

[OT93] C. L. Ong and W. T. Tsai. Class and object extraction from imperative code. *Journal of Object-Oriented Programming*, pages 58–68, March–April 1993.

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[Pas84] R. Passini. *Wayfinding in Architecture*. Van Nostrand Reinhold, 1984.

[Pen87] N. Pennington. Comprehension strategies in programming. In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Proceedings of the Second Workshop on Empirical Studies of Programmers*, pages 100–112. Ablex Publishing, 1987.

[Pig97] T. M. Pigoski. *Practical Software Maintenance – Best Practices for Managing Your Software Investment*. John Wiley & Sons, 1997.

[PP94] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[RBJ97] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[RFT99] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Proceedings of the 26th Symposium on Principles of Programming Languages (POPL'99)*. ACM Press, 1999.

[RHN99] Zs. Ruttkay, P. ten Hagen, and H. Noot. Chartoon: a system to animate 2D cartoon faces. In *Short Papers and Demos Proc. of Eurographics'99*, 1999.

[RK91] J. Rekers and W. Koorn. Substring parsing for arbitrary context-free grammars. *Sigplan Notices*, 26(5):59–66, 1991.

[RPR93] H. Reubenstein, R. Piazza, and S. Roberts. Separating parsing and analysis in reverse engineering tools. In *Proceedings of the 1st Working Conference on Reverse Engineering (WCRE'93)*, pages 117–125, October 1993.

[Rus91] G. W. Russell. Experience with inspection in ultralarge-scale developments. *IEEE Software*, 8(1):25–31, 1991.

[RV99] V. Rajlich and S. Varadarajan. Using the web for software annotations. *International Journal of Software Engineering and Knowledge Engineering*, 9(1):55–72, 1999.

[Sch91] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering (ICSE-13)*, pages 83–92. ACM Press, 1991.

[Sch00] A. Schneider. JUnit best practices. *JAva World*, 12, 2000. http://www.javaworld.com/javaworld/jw-12-2000/jw-1221-junit.html.

[SCHC99] S. E. Sim, C. L. A. Clarke, R. C. Holt, and A. M. Cox. Browsing and searching software architectures. In *Proceedings of the 1999 International Conference on Software Maintenance (ICSM'99)*, pages 381–390. IEEE Computer Society Press, 1999.

[SE84] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, September 1984.

[SFI88] O. Stock, R. Falcone, and P. Insinnamo. Island parsing and bidirectional charts. In *Proceedings of the 12th Conference on Computational Linguistics*, pages 636–641, 1988.

[SH98] S. E. Sim and R. C. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *Proceedings of the 20th International Conference on Software Engineering (ICSE-20)*, pages 361–370. ACM Press, 1998.

[SM79] B. Shneiderman and R. Mayer. Syntactic/semantic interactions in programmer behaviour: a model and experimental results. *International Journal of Computer and Information Sciences*, 8(3):219–238, 1979.

[Sne96] G. Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–189, April 1996.

[Sne98a] H. Sneed. Architecture and functions of a commercial reengineering workbench. In P. Nesi and F. Lehner, editors, *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98)*, pages 2–10. IEEE Computer Society Press, 1998.

[Sne98b] G. Snelting. Concept analysis — a new framework for program understanding. In *Proceedings of the 1998 ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, 1998. Sigplan Notices, 33(7).

[Sne00] G. Snelting. Software reengineering based on concept lattices. In *Proceedings of the 4th European Conference on Software Maintenance and Reengineering (CSMR'00)*. IEEE Computer Society Press, 2000.

[SR96] M. Siff and T. Reps. Program generalization for software reuse. In *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering, (FSE-4)*, pages 135–146, 1996. SIGSOFT Software Engineering Notes, 21(6).

[SR97] M. Siff and T. Reps. Identifying modules via concept analysis. In *Proceedings of the 1997 International Conference on Software Maintenance (ICSM'97)*. IEEE Computer Society Press, 1997.

[ST98] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *Proceedings of the 6th ACM SIGSOFT Symposium on the Foundations of Software Engineering, (FSE-6)*, pages 99–110. ACM Press, 1998. SIGSOFT Software Engineering Notes, 23(6).

[Sto98] M. Storey. *A Cognitive Framework For Describing and Evaluating Software Exploration Tools*. PhD thesis, Simon Fraser University, Canada, 1998.

[Sud88] T. A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison-Wesley, 1988.

[SV98] M. P. A. Sellink and C. Verhoef. Native patterns. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE'98)*. IEEE Computer Society Press, October 1998.

[SV99] M. P. A. Sellink and C. Verhoef. Generation of software renovation factories from compilers. In *Proceedings of the 1999 International Conference on Software Maintenance (ICSM'99)*, 1999.

[SYM00] T. Systä, P. Yu, and H. Müller. Analyzing Java software by combining metrics and program visualization. In *Proceedings of the 4th European Conference on Software Maintenance and Reengineering (CSMR'00)*, 2000.

[Tar41] A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6:73–89, 1941.

[Tic01] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, December 2001.

[Tom86] M. Tomita. *Efficient Parsing for Natural Languages: A Fast Algorithm for Practical Systems*. Kluwer, 1986.

[TWSM94] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, pages 501–520, December 1994.

[Ver00] E. Verhoeven. COBOL island grammars in SDF. Master's thesis, Informatics Institute, University of Amsterdam, 2000.

[Vis97] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[WBF97] T. Wiggerts, H. Bosma, and E. Fielt. Scenarios for the identification of objects in legacy systems. In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE'97)*, pages 24–32. IEEE Computer Society Press, October 1997.

[WM96] A. Watson and T. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. NIST Special Publication, 500–235. U. S. National Institute of Standards and Technology, Washington, D. C., September 1996.

[WP99] N. Williams-Preston. New type signatures for legacy Fortran subroutines. In *Proceedings of the 1999 ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'99)*, pages 76–85. ACM Press, September 1999. SIGSOFT Software Engineering Notes, 24(5).

[WS91] L. Wall and R. L. Schwarz. *Programming Perl*. O'Reilly & Associates, Inc., 1991.

[WTMS95] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Structural redocumentation: a case study. *IEEE Software*, 12(1):46–54, 1995.

[WZ91] M. Wegman and K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):18–210, 1991.

[YRK99] R. J. Yarger, G. Reese, and T. King. *MySQL & mSQL*. O'Reilly, 1999. http://www.mysql.org/.

Summary in Dutch / Samenvatting

explorer /ɪksplɔ:rəʳ/. Ontdekkingsreiziger, onderzoeker.
Van Dale Groot woordenboek Engels–Nederlands

ontdekkingsreiziger Iemand die onbekende gebieden zoekt en onderzoekt.
Kramers nieuw woordenboek Nederlands

Dit proefschrift gaat over ontdekkingsreizen in softwaresystemen, of beter gezegd, over de verkenning van onbekende gebieden in softwaresystemen. Dit roept meteeen een tweetal vragen op: (1) *waardoor* zijn er onbekende gebieden in softwaresystemen, en (2) *waarvoor* is verkenning van die onbekende gebieden noodzakelijk.

Het antwoord op beide vragen is *software-evolutie*: ieder softwaresysteem dat gedurende langere tijd in gebruik is zal tijdens die periode een aantal malen aangepast en uitgebreid moeten worden om operationeel te blijven. Deze modificaties kunnen variëren van technische aanpassingen vanwege de overgang naar een nieuw besturingssysteem tot functionele aanpassingen als gevolg van nieuwe of veranderde wensen van de gebruikers. Dit proces waarbij de toestand van een softwaresysteem in overeenstemming gebracht wordt met de veranderende omstandigheden en wensen noemen we software-evolutie (of software-onderhoud).

Het is een bekend gegeven dat als gevolg van software-evolutie de complexiteit van het softwaresysteem zal toenemen en de kennis over het systeem zal afnemen, tenzij er specifieke maatregelen genomen worden om dit tegen te gaan. De reden voor dit verval is dat door de opeenstapeling van veranderingen de originele structuur van het systeem verstoord raakt. Stukje bij beetje zal de relatie tussen het systeem en de ontwerpdocumentatie verdwijnen waardoor het systeem steeds moeilijker te onderhouden is. Als gevolg van deze kwaliteistsvermindering van informatie zullen opvolgende veranderingen een nog desastreuzer effect op de structuur en onderhoudbaarheid van het systeem hebben.

In dit proefschrift bestuderen we verschillende technieken en gereedschappen (*tools*) die software-ontwikkelaars kunnen helpen bij het software-evolutieproces. De nadruk ligt hierbij op het ondersteunen van de mentale beeldvorming over een softwaresysteem (ook wel aangeduid als *program comprehension* of *program understan-*

ding). Voorbeelden hiervan zijn het begrip van de werking van een systeem en het inzicht in de structuur van het systeem en in de samenhang tussen de individuele onderdelen (software-architectuur).

Verkenning van Softwaresystemen

In Hoofdstuk 1 onderzoeken we de analogie tussen het verkennen van software en het verkennen van steden en gebouwen. Als mensen een hen onbekende stad of gebouw bezoeken gebruiken ze een aantal basale verkenningstechnieken om zich een mentaal beeld (*mental model*) van de structuur van de stad of het gebouw te vormen. Ze creeren als het ware een soort landkaart of plattegrond in hun hoofd die hen helpt bij het vinden van de juiste weg en het bezoeken van speciale plekken. Geïnspireerd door cognitieve studies naar de mentale beeldvorming over steden en gebouwen, en de manier waarop stedenbouwkundigen en architecten deze kennis gebruiken om de inzichtelijkheid van een stad of gebouw te vergroten, gaan we op zoek naar de tegenhangers op softwaregebied. We definiëren het begrip *inzichtelijkheid van een softwaresysteem* (*legibility of a software system*) als *het gemak waarmee de verschillende delen van een softwaresysteem herkend en in een coherente structuur geordend kunnen worden* en beschrijven de vijf bouwstenen die gebruikt worden voor beeldvorming over een softwaresysteem:

bakens (landmarks) kunnen als herkennings- en referentiepunten gebruikt worden en geven een gevoel van positie en richting. Voorbeelden zijn specifieke datatypes (bijvoorbeeld data, rekeningnummers, bedragen) of broncode die aan bepaalde karakteristieken voldoet.

knooppunten (nodes) zijn de structurele entiteiten in het systeem, zoals programma's, modules, functies, datatypes, klassen, methoden, variabelen, enz.

paden (paths) vormen relaties tussen twee knooppunten die gevolgd kunnen worden om door het systeem te navigeren. Bijvoorbeeld: programma- of functieaanroepen, overerving, enz.

wijken (districts) groeperen de knooppunten op basis van gemeenschappelijke eigenschappen. Voorbeelden zijn de componenten in een software-architectuur, de variabelen van hetzelfde datatype, enz.

grenzen (edges) geven de overgang van de ene naar de andere wijk aan en bemoeilijken navigatie. Voorbeelden zijn de grenzen tussen de programma- en systeembibliotheken en de applicatiecode of de grens tussen client- en servercode.

De aanwezigheid en herkenbaarheid van bovenstaande elementen is bepalend voor de inzichtelijkheid van een softwaresysteem. Het onderzoek dat beschreven is in de rest van het proefschrift richt zich dan ook op het detecteren en beter zichtbaar maken van deze elementen in een softwaresysteem.

Eilandgrammatica's

In het *Island Grammars* deel van dit proefschrift onderzoeken we het afleiden van modellen uit de broncode van softwaresystemen en het gebruik van die modellen voor het analyseren van de gevolgen van bepaalde veranderingen (*impact analysis*). Een van de grootste uitdagingen bij het afleiden van modellen uit broncodes is het omgaan met de onregelmatigheden die typerend zijn voor het *reverse engineering* gebied. Voorbeelden van deze onregelmatigheden zijn syntactische fouten, incomplete broncode, verschillende dialecten van programmeertalen en programmeertalen die ingebed zijn in andere programmeertalen. In Hoofdstuk 2 presenteren we een oplossing voor dit probleem in de vorm van *eilandgrammatica's* die gebruikt kunnen worden voor de generatie van robuuste parsers die de gedetailleerdheid en accuraatheid van syntactische analyse combineren met de flexibiliteit en ontwikkelsnelheid van lexicale oplossingen. In Hoofdstuk 3 motiveren we waarom een *lichtgewicht* vorm van impactanalyse een vereiste is voor de planning en inschatting van projecten voor software-onderhoud. We presenteren een techniek voor de generatie van gereedschappen voor lichtgewicht impactanalyse uit eilandgrammatica's en demonstreren onze techniek aan de hand van een praktijkvoorbeeld. In dit voorbeeld bepalen we de gevolgen van de wens om alle 9-cijferige bankrekeningnummers op te rekken naar 10-cijferige nummers in de software van een grote Nederlandse bank.

Type-inferentie

In het *Type Inference* deel van dit proefschrift kijken we naar zogenaamde impliciete of afgeleide datatypes (*inferred types*) als een abstractie om de variabelen in een softwaresysteem te groeperen. Datatypes zijn een gebruikelijke abstractie in programmeertalen en ze vormen een goed vertrekpunt voor de verkenning van softwaresystemen en taken op het gebied van reverse engineering. Helaas zijn de softwaresystemen die deze activiteiten het hardst nodig hebben vaak geschreven in programmeertalen zonder adequaat type-systeem (bijvoorbeeld COBOL). Verder komt het in getypeerde programmeertalen (zoals C) vaak voor dat de ontwikkelaars gebruik maken van hetzelfde ingebouwde datatype (bijvoorbeeld `char`, `int` of `float`) voor de representatie van verschillende logische datatypes (bijvoorbeeld aantal en leeftijd). Als gevolg daarvan kunnen de datatypes niet meer gebruikt worden als abstractie omdat ze variabelen groeperen die eigenlijk in verschillende klassen ingedeeld zouden moeten worden. Om deze kwestie op te lossen presenteren we in Hoofdstuk 4 een methode voor het automatisch afleiden van datatypes voor variabelen. Deze methode groepeert de variabelen in een systeem op basis van de manier waarop deze variabelen daadwerkelijk gebruikt worden in dat systeem. Bijvoorbeeld, alle variabelen die met elkaar vergeleken worden krijgen hetzelfde datatype. We geven het formele typesysteem en de afleidingsregels voor deze aanpak en beschrijven hoe deze ideeën geïmplementeerd zijn in een prototype gereedschap voor de analyse van softwaresystemen die geschreven zijn in COBOL.

We vervolgen onze studie in Hoofdstuk 5 met de analyse van *type-vervuiling* (*type pollution*), het fenomeen dat afgeleide datatypes te groot worden en variabelen bevatten die intuïtief niet in dit datatype thuishoren. We presenteren een verbeterde afleidingsmethode die gebruik maakt van subtypering en laten zien dat dit een effectieve manier is om de type-vervuiling aan te pakken. In Hoofdstuk 6 combineren we afgeleide datatypes met conceptanalyse om de procedures in een softwaresysteem te groeperen met de datatypes die ze gebruiken. Het resultaat zijn abstracties die erg lijken op zogenaamde abstracte datatypes. Deze abstracties kunnen gebruikt worden voor verkenning van softwaresystemen en kunnen als basis dienen voor een object-georienteerd herontwerp van het systeem. In Hoofdstuk 7 onderzoeken we hoe een kunstmatige abstractie, zoals datatypes die op zichzelf geen onderdeel uitmaken van COBOL, op een bruikbare manier gepresenteerd kan worden aan COBOL onderhoudsprogrammeurs. We beschrijven de constructie van TYPEEXPLORER: een gereedschap voor het verkennen van COBOL softwaresystemen op basis van afgeleide datatypes. Verder laten we zien hoe dit gereedschap gebruikt kan worden voor de verkenning van een industrieel COBOL *legacy* systeem dat bestaat uit 100.000 regels broncode.

Refactoring & Testen

In het laatste deel van dit proefschrift onderzoeken we de kwaliteitsaspecten van een softwaresysteem, bezien vanuit een *refactoring*- en testperspectief. Refactoring (letterlijk herfactoriseren) is het herschrijven van een stuk programmatekst, zodanig dat het extern waarneembare (functionele) gedrag van die broncode gelijk blijft maar de interne structuur verbetert waardoor de werking eenvoudiger te begrijpen is.

In Hoofdstuk 8, presenteren we een methode voor de automatische detectie en visualisatie van zogenaamde *luchtjes* (*code smells*) die aan JAVA broncode kleven. Deze luchtjes zijn een metafoor voor die aspecten van een programmatekst die weliswaar niet zonder meer fout zijn (de betreffende programmatuur kan zonder problemen vertaald en uitgevoerd worden), maar wel de begrijpelijkheid en onderhoudbaarheid van het systeem sterk verminderen. Deze aspecten zijn typisch gerelateerd aan slecht programma-ontwerp en slordig programmeren. Voorbeelden van luchtjes die aan broncode kunnen kleven zijn codeduplicatie (als eenzelfde stuk broncode letterlijk op verschillende plaatsen in het programma herhaald wordt), methoden die te lang zijn, klassen die te veel functionaliteit bevatten, klassen die encapsulatieprincipes overtreden, enz. De resultaten van deze detectie kunnen op twee manieren gebruikt worden: (1) ter ondersteuning van automatische kwaliteitsinspectie van de broncode van een systeem waarbij de luchtjes gebruikt worden om het inspectieproces te leiden, en (2) voor het creëren van intelligente gereedschappen voor refactoring die niet alleen op aanvraag een stuk broncode op een gegeven manier herschrijven (hetgeen momenteel *state-of-the-art* is) maar die ook suggereren welk fragment van het programma op welke manier herschreven kan worden om het te verbeteren.

Ontwikkelaars gebruiken zogenaamde deeltests (*unit tests*) om het extern waarneembare gedrag bij refactoring te bewaken. Het idee is dat als een test voor en na refactoring correct werkt, de refactoring geen nadelige gevolgen heeft gehad. De broncode van de deeltests (de "testcode") wordt door de ontwikkelaars geschreven en onderhouden, in dezelfde programmeertaal als het echte programma (de "productiecode"). Naarmate het aantal tests bij een systeem groeit zal de behoefte aan onderhoud en refactoring van die testcode ook toenemen. In Hoofdstuk 9 beargumenteren we dat refactoring van testcode anders is dan refactoring van productiecode. We presenteren een verzameling specifieke luchtjes die aan testcode kunnen kleven en beschrijven hoe deze ook weer verwijderd kunnen worden. In Hoofdstuk 10 onderzoeken we de relatie tussen testen en refactoring en bekijken hoe ze in elkaar verstrikt kunnen raken wanneer door refactoring de testcode niet meer correct werkt (bijvoorbeeld doordat een methode naar een andere klasse verplaatst is terwijl de testcode nog naar de oorspronkelijke klasse verwijst). We beschrijven de voorwaarden waaronder zulke problemen op kunnen treden en geven een overzicht bij welke van de standaard refactorings uit [Fow99] dit gebeurt. Tot slot introduceren we het begrip *test-first refactoring*, een methode voor het verbeteren van de kwaliteit van een softwaresysteem waarbij luchtjes die aan de testcode kleven als bakens gebruikt worden voor het verkennen van de productiecode. Het achterliggende idee hiervan is dat kwalitatief goede broncode ook goed te testen is en dat suboptimale ("onwelriekende") testcode een indicatie geeft van plaatsen waar de productiecode verbeterd kan worden.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1996-1

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms.* Faculty of Mathematics and Computer Science, KUN. 1996-2

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation.* Faculty of Mathematics and Computer Science, KUN. 1996-3

M.G.A. Verhoeven. *Parallel Local Search.* Faculty of Mathematics and Computing Science, TUE. 1996-4

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory.* Faculty of Mathematics and Computer Science, KUN. 1996-5

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems.* Faculty of Mathematics and Computing Science, TUE. 1996-6

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance.* Faculty of Mathematics and Computer Science, UvA. 1996-7

H. Doornbos. *Reductivity Arguments and Program Construction.* Faculty of Mathematics and Computing Science, TUE. 1996-8

D. Turi. *Functorial Operational Semantics and its Denotational Dual.* Faculty of Mathematics and Computer Science, VUA. 1996-9

A.M.G. Peeters. *Single-Rail Handshake Circuits.* Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism.* Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference.* Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics.* Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth.* Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context.* Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types.* Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics.* Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution.* Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing.* Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing.* Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering.* Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation.* Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller.* Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model.* Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing.* Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes.* Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search.* Faculty of Mathematics and Natural Sciences, Univ. Leiden. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection.* Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Schedulere Optimization in Real-Time Distributed Databases.* Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics.* Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems.* Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods.* Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems.* Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems.* Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules.* Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System.* Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars.* Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Progam Construction.* Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic.* Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms.* Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes.* Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols.* Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor.* Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant.* Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs.* Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language.* Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure.* Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication.* Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle.* Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection.* Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences.* Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes.* Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes.* Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking.* Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols.* Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents.* Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ.* Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking.* Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects.* Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity.* Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing.* Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing.* Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13