Techniques for Understanding Legacy Software Systems

About the cover: A man unreels in front of him a portion of a reel of punched paper tape. A B220 computer installation can be seen in the background behind him. Picture from the Burroughs Corporation Collection (CBI 90), Charles Babbage Institute, University of Minnesota, Minneapolis.

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.

Techniques for Understanding Legacy 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 dinsdag 26 februari 2002, te 10.00 uur

door Tobias Kuipers
geboren te Vleuten, Nederland

Promotor:	Prof. dr. P. Klint
Copromotor:	Dr. A. van Deursen
Faculteit:	Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Contents

Preface **9**

1 Introduction **13**
1.1 Maintaining Legacy Systems 14
1.1.1 Software Engineering for Maintenance 15
1.2 Changing Legacy Systems 17
1.2.1 Minor Change 18
1.2.2 Structural Change 18
1.3 Research Questions 19
1.4 Reader's Roadmap 21

2 Rapid System Understanding **23**
2.1 Introduction 23
2.2 Tool Architecture 24
2.3 Cases Investigated 25
2.4 Collected Data 26
2.4.1 System inventory 26
2.4.2 Program call graph 27
2.4.3 Database usage 27
2.4.4 Field usage 29
2.4.5 Section call graph 31
2.4.6 Further experiments 32
2.5 Interpreting Analysis Results 32
2.5.1 Understanding copybooks 32
2.5.2 Call graph and reusability 33
2.5.3 Understanding data usage 34
2.5.4 Reusability assessment 34
2.6 Related Work 35
2.7 Conclusions 36
2.8 Future Work 37

3 Building Documentation Generators **39**
3.1 Introduction 39
3.2 Source Code Analysis 41
3.2.1 Lexical analysis 41
3.2.2 Syntactic Analysis 42
3.2.3 Island Grammars 43
3.2.4 Parse Tree Analysis 45
3.3 Extracting Documentation 45
3.3.1 Manual versus Automated Extraction 45
3.3.2 System Decomposition 46
3.3.3 Aggregation and Use Relations 46
3.3.4 System and Subsystem Partitioning 47
3.3.5 Program Descriptions 47
3.3.6 Section Descriptions 47
3.3.7 Batch Job Dependencies 48
3.4 Presenting Documentation 49
3.5 Business Case 50
3.5.1 Background 50
3.5.2 Documentation Wishes 51
3.5.3 Derived Documentation 51
3.5.4 Evaluation 53
3.6 Concluding Remarks 54

4 Identifying Objects using Cluster and Concept Analysis **57**
4.1 Introduction 57
4.2 Related Work 59
4.3 Field and Program Selection 60
4.4 Cluster analysis 61
4.4.1 Overview 61
4.4.2 Experimental Testbed 64
4.4.3 Experiments 65
4.4.4 Assessment 67
4.5 Concept Analysis 68
4.5.1 Basic Notions 68
4.5.2 Experimental Testbed 70
4.5.3 Experiments 70
4.6 Clustering and Concepts Compared 72
4.7 Object Identification 74
4.8 Concluding Remarks 76
4.8.1 Acknowledgments 76

5 Types and Concept Analysis for Legacy Systems **77**
5.1 Introduction 77
5.2 Type inference for COBOL 78
5.3 Concept Analysis 83
5.3.1 Basic Notions 84
5.4 Combine Types and Concepts 85
5.4.1 Data for Concept Analysis 86
5.4.2 Experiments Performed 86
5.5 Refinement of Concepts 90
5.5.1 Operations on concept lattices 90
5.5.2 Relation with source 91
5.6 Implementation 91
5.7 Related Work 94
5.8 Concluding remarks 94
5.8.1 Future work 95

6 Object-Oriented Tree Traversal with JJForester **97**
6.1 Introduction 97
6.2 JJForester 99
6.2.1 Overview 99
6.2.2 SDF 100
6.2.3 Code generation 102
6.2.4 Programming against the generated code 104
6.2.5 Assessment of expressiveness 107
6.3 Case study 108
6.3.1 The Problem 108
6.3.2 T-scripts explained 109
6.3.3 Analysis using JJForester 111
6.4 Concluding remarks 118
6.4.1 Contributions 118
6.4.2 Related Work 118
6.4.3 Future Work 119

7 Legacy to the Extreme **121**
7.1 Introduction 121
7.2 Tools Make It Possible 122
7.2.1 The Legacy Maintenance Toolbox 123
7.2.2 More Tools 125
7.3 Adopting XP Step by Step 126
7.4 XP on Legacy Code 128
7.5 Conclusions 131

8 Conclusions **133**
8.1 Reduce Search Space 133
8.2 Information for Structural Change 134
8.3 Altering the Software Engineering Process 134

Summary **145**

Samenvatting **149**

Preface

The work presented in this thesis was performed over a period of four years at the Center for Mathematics and Computer Science (CWI) in Amsterdam, The Netherlands. Over this period, more and more interest in our group's research was expressed by various companies. A number of people in the group had been toying with the idea of starting a company to aid in the technology transfer of the CWI research for some years, and this seemed like the ideal situation to actually start this company. I am very fortunate to have been part of both the research group, and the start-up company. In fact, I moved from CWI to the Software Improvement Group (the startup) after my four years at CWI, when all I had to do to finish this thesis was to "write the preface".

It took me over a year to actually finish it, and the fact that I did can hardly be attributed to me. I would like to thank the people who it *can* be attributed to profoundly: Arie van Deursen was the project leader of the project I first joined at CWI. He was my mentor, and taught me how to write research papers. He has become a good friend and (less important) a partner in the aforementioned company. Paul Klint taught me how to finish a thesis, mainly by *not* asking when I was going to finish it. Apart from that, he was one of the most accessible thesis advisors I know, and has been an inspiration. I fondly remember the seemingly random state-of-the-world talks we had (and still have).

Other people at CWI that need to be applauded are Joost Visser, who knows how to separate the important stuff from the less important stuff, thus making you go twice as fast on a good day. Especially our long discussions over dinner (after long discussions on where to eat) have been very educational (We need more coasters!). Leon Moonen has been there since we started, we shared a room as assistants at the UvA. Leon prefers rooms at least 5 degrees colder than I do, so we didn't share rooms at CWI. We did have a lot of very productive discussions, over even more coffee, however (Let's go skating when I'm done). Merijn de Jonge was an instrumental part of the team who always asked whether what you were doing couldn't be done better. He has written a number of very elegant software packages I use everyday. Furthermore, he got us home when we went sailing and the rudder fell off.

Dinesh was my first roommate, and the first to go. Dinesh showed me that it was okay to have non-conventional ideas (at least for him). He has become a good friend, at which point he packed his stuff and moved half way across the world. I did visit him in California, after which he moved to yet another continent. Let's see what happens when I visit him there. Jeroen Scheerder showed me lots of cool Mac stuff, which made me buy the one I'm typing this on now. Apart from that, he produced some of the software that was needed to do the research in this thesis. Jurgen Vinju took my place as the weekly talk organizer. He manages to buy all the nice gadgets at the right time, although he should be legally barred from buying cars. He, too, wrote some of the software I still use everyday. I hope he keeps calling me telling me he bought me a ticket to this cool show at the Paradiso (Hou het werkelijk!).

I enjoyed working at CWI and like to thank the rest of the group for providing such a nice environment: Mark van den Brand, Ralf Lämmel, Jan Heering, Pieter Olivier and Hayco de Jong, thanks!

I need to congratulate Eelco Visser on becoming a daddy, and need to thank him for being instrumental in my pursuing a research job in the first place. Let's do something together, now that I have a bit more time.

Over at the Software Improvement Group, I have to thank you all! Guys, we are doing good! Marjo Wildvank has taught me more about business then I ever thought I'd wanted to know. He is a master story teller, and I am starting to believe that his stories are true, too. Alex van den Bergh has rescued me on more than one occasion. Thank you for putting up with me, and for the great work you do, even though being cool is not a part-time job. Visit checkdef.com! Other SIG people: Steven Klusener, Gerard Kok, Indra Polak, Ernst Verhoeven, and more recently Albertine Frielinck, thank you very much for your sense of humor, and for your nice work (and the bedrijfsuitjes).

Fortunately, I still seem to have sort of a life, which is largely due to various friends calling and demanding to go have a beer somewhere, or come over for dinner. You know who you are.

Rap9 is such an institution that I almost feel I don't need to mention it. I am grateful that I can be part of a football team with such a history, and such brilliant players. We will become champions, although I fear who will tattoo what where, when it happens.

My Mom and Dad were there first, and apparently will support me, no matter what. What can I do in return, but to install and fix their computer? The fact that they moved to just a couple of blocks from our place make my visits more frequent and always fun, especially when dinner is involved (I still haven't had the worteltjesspaghetti!). I also thank my little big brother for being my brother. Let's hang out more.

Johan Prins and Maria Sindram, thank you having me visit you in the US, lastly in Charleston, S.C., where part of this thesis was written.

Finally, the one person who actually did all the work, supported me throughout my time at CWI and SIG, while graduating, starting her own career, getting a new job, taking me on beautiful holidays to Britanny and finding the time to actually marry me in between... Maartje, thank you for making me a better person.

Chapter 1

Introduction

Every software system that is being used needs to be maintained. Software is only finished when it is no longer in use. However, the rate of change may vary wildly from system to system. The number of releases for a new internet application that needs to stay ahead of the competition by adding features every week may be very high. It is typically low for embedded systems software, because of the high cost of retrofitting for instance your television with new software.

There are three main reasons for changing software, the so-called software maintenance categories [Swa76].

Corrective maintenance The first reason software needs to be changed is fault repair (or bug fixing). No software is delivered fault free, and during the lifetime of the software some of those faults will become apparent and need to be fixed.

Perfective maintenance A second reason software needs to be changed is of commercial nature. When a company brings out a software product in a competitive market, then the competition may want to change their product to stay competitive. Software may also need change to reflect the business process of a particular company. The software used inside large services organizations may be modeled upon the structure of the organization. If there is a division “Regional offices” then that division usually has its own systems, as has the division “Foreign offices”. Were the company to reorganize, and these two divisions to be merged into a new division “Offices”, then this has consequences for the two software systems.

Adaptive maintenance Finally, systems may need to change for legal or technical reasons. The system may need to run on different hardware or it may need to be upgraded to run on a different operating system. Legal changes could include the modification of currencies on which the system operates,

or a change in the time during which historical data is stored by the system. After the change the system will perform the same task as before, with hardly any visible change for the user.

An overview of different measurement studies presented in [Pig97] shows that on average about 80% of maintenance cost is in either perfective or adaptive maintenance (or non-corrective maintenance) and about 20% in corrective maintenance.

1.1 Maintaining Legacy Systems

When, for whatever reason, an existing system needs to be modified, it will invariably be seen as a *legacy* system (at least in part). That is, the system is the legacy of the people who have created it, or have modified it previously. In the ideal situation the original development team will stay on as maintainers of the system throughout its lifetime. This way, everyone who was ever involved in the design and building of the system is available to change it. Unfortunately, this is never the case. Traditional software engineering distinguishes strongly between developers or creators of a system, and maintainers of the same system. This follows the traditional manufacturing model, where a product, for instance a car, is developed and built by a team of people. Once the car is finished and delivered, the maintenance of the car will be performed by a different team.

The process of keeping a software system in tune with the needs of its users is referred to as *software evolution*. In order to be able to evolve a system, its maintainers need to know, in some detail, the inner workings of that system. Because they did not design nor develop the system, this knowledge does not come to them naturally. In an ideal case, there is ample knowledge about the system available outside of it in the form of design documents, both functional and technical, and actual technical documentation of the source code of the system. What's more, the original designers and developers of the system may be available to answer questions. Unfortunately, for the original development team there is no need to produce accurate, factual, and concise design documents, because they can build the system without them. Moreover, in a traditional software engineering environment, the designers of a system will produce a design document that cannot be implemented as is. All sorts of details in a design will be deemed "implementation details", and will be left to the developers to sort out. In order to sort out the implementation details, the developers may need to (and will) stretch the design a little to accommodate for their best solution. This is not a problem as long as the intention of the original design was clear to both designer and developer, because then the chosen solution will be a common one. Unfortunately, the slight change in the design is only documented in the head of the developer who made it. During the development of a non-trivial system there are hundreds, if not thousands of such implementation details to decide about, and each one may warrant their

own slight design change. These can add up to quite major changes which are not documented anywhere.

Software developers get away with this (as opposed to car developers) because there is no intrinsic need for these documents. Software developers in general build a single software system, which, once it is finished, can be copied an infinite number of times for negligible cost. This single copy is built by a single development team. This is as true for both mass distributed software as it is for single-purpose custom built software. Building a single car in the contemporary car market can never be cost effective. The volume production of cars is what keeps a car manufacturer going, so the need for documents that trace the exact steps to create a car is paramount. Not only do all cars need to be built from scratch, there is not a single development team. A single car model may be built on several sites distributed all over the world, and at each site they may be built by several teams.

1.1.1 Software Engineering for Maintenance

The previous paragraph should have started with the sentence "software developers get away with this temporarily", because in most of the cases software developers do *not* get away with it *over time*. Unlike cars, the finished software product can be changed to become a slightly changed, or even radically different software product. As described above, these changes are almost impossible to make without an understanding of the inner workings of the system, and may still be very hard with such an understanding.

A number of methods have been proposed and applied in the past to allow for people new to the system to understand it in the amount of detail they need to do their work. These people have been called "Software Immigrants" in the literature [SH98].

The source code of a software system can be seen as a document that describes exactly what the system does, by definition. Unfortunately, this is done in such enormous detail that it is hard for any individual to keep an overview of the whole system. In order to maintain an overview, and help introduce "software immigrants" to the system, some other, less detailed, source of information is needed. The methods described below all propose a different way to create, and maintain that other source of information about the system.

Traditional Software Engineering The traditional "waterfall" software engineering model starts from the premonition that software is only developed once, and that, once it is developed, it will enter into a maintenance state. When the software enters from the development stage into the maintenance stage, a number of documents should be delivered as well. Different methodologies here require different documents, but in general there should be a functional design document, which describes the overall functionality of the system, there should be a technical

design document, which describes the overall technical layout of the system, there should be in-depth technical documentation on a per-module, or per-program level, and the source code itself could be required to adhere to certain coding standards and contain information about the code.

Evidence suggests that the average software system designed and built following this method spends about 70% of it's time and effort and cost in the maintenance stage[Pig97], and in that stage the system may be changed quite drastically. However, after 5 years of maintenance, only the documents from the date of delivery are available, which tend to be horribly outdated, particularly the technical documentation.

More cyclical methodologies have been developed, to iterate over the design and development phase. Errors discovered during the development phase can be reset in the following design refinement phase. Unfortunately, these methodologies, for example Boehm's Spiral Model [Boe88], also end with the implementation phase, and do not consider themselves with maintenance at all.

What is apparent in these models is that a strict dichotomy exists between the actual software system that is running on a computer (the combined source code) and the documents available about the system. That is, the actual software system may or may not do what is in the documentation, and there is no way to guarantee that the two are in any way related.

Literate Programming Literate programming [Knu84] has been proposed as a solution for this problem: literate programming requires the developer to have a single document containing both the program text, as well as the documentation that describes the program text. In fact, literate programming in its purest form requires the developer to produce a book describing the problem the software system is going to solve, how this problem can be broken down into smaller problems, and how all these small problems can be solved individually by a very small piece of software, which is so small that it can be printed in the book and understood instantaneously. The advantage of having both documentation and code in the same document is that the engineer who has to modify the code has the documentation right there on his screen. When he needs to modify the system, he should first decide where the error is: Is it in the narrative of the book? Then he needs to fix the narrative, and then the source code. If the narrative is correct, then there somehow is a fault in the code, and he should fix it. This method requires a lot of discipline from developers to fix the narrative before the code. Even though the two are in the same document, the relation between the two is purely circumstantial: the narrative may tell a completely different story from the code.

Executable Specifications Another proposed solution comes in the form of executable specifications. Here the design of a system is laid down in a formal system specification document. The actual implementation of the system is then de-

rived from that specification automatically. Because the specification language is, in fact, a (very) high-level programming language, the design, or specification, cannot skip over the already mentioned implementation details. This means that formal specifications of non-trivial problems may contain much more detail than would be written down in a natural language design document. To introduce a new engineer to the system, natural language documents are needed, and the advantage of formal specifications over "regular" programming languages with respect to understandability and maintainability are lost.

Domain Specific Languages Domain Specific Languages are usually extremely high level languages that borrow parts of their syntax from the vocabulary of a certain problem domain. Having "programmers" who are well-versed in this particular problem domain solves the problem of having software immigrants. Being domain experts, these people are already familiar with the same vocabulary that the domain specific language is built from. Having, or creating a domain specific language is, by definition, only feasible in a situation where the problem domain is extremely well defined. Furthermore, having a domain specific language can only be profitable if a number of systems will be developed in the same problem domain, to warrant the initial investment in the development of the domain specific language itself [DK98].

Extreme Programming The most extreme solution is provided by the aptly named Extreme Programming (XP) methodology. XP does not distinguish at all between design, testing, development, or maintenance stages. Instead, a team of engineers produces a tiny, but working system in a couple of weeks, and keeps modifying and extending that system into the system that is actually requested by the customer. There are no design documents, or at least no persistent design documents; the design of the system is the source code itself. Because all engineers that are maintaining the system have helped develop and design it, their knowledge about the system comes naturally. Writing documentation or commenting code is actually discouraged, because it will be outdated the moment someone changes the code. Understandability should be achieved by creating readable code. Because all code is written in rotating pairs, there is no one engineer with exclusive knowledge of (part of) the system.

1.2 Changing Legacy Systems

Even though there is a lack of documentation and a lack of people with insight into the system, the legacy system still needs to be changed. The changes that can be made to a legacy system can fall into one of two categories. Both types of changes have their particular reasons, and both require a different type of knowledge about

the system to be available for the engineer. Of course the difference between the two types of change is not always clear; the two categories can be seen as the two ends of a change spectrum.

1.2.1 Minor Change

Minor changes in a software system are usually operational in nature: a system terminates on unexpected or illegal input, or it produces the wrong results for a particular input. An engineer is called in to repair the program as soon as possible, to get it up and running again. Large batch processes on mainframe computers typically run at night, with a single engineer standing by to fix things when they go wrong. It is impossible for this one engineer to know very much about all the systems that may be running at any given night. When the system breaks down, the engineer needs access to information about the system very fast, and will not be bothered by the amount of detail the information contains. The first goal of the information available should be to decrease the search space in which the problem can be found. If at a given night 25 systems are running, and each system consists of 200 modules, then having information available that within minutes pinpoints the error in one of the 5000 modules is invaluable. Further information that narrows the error down even further to a particular section of a module may be useful, but if the information takes half an hour to retrieve then its usefulness is rather limited, because an engineer can probably read the whole module in about half an hour.

Minor changes occur where an otherwise working system breaks down. Obviously, this is not a definition but rather a vague description: If a system breaks down on January 1st, 2002, because it cannot process the Euro currency, hardly any software engineer will tell you that fixing this problem would involve only a minor change.

1.2.2 Structural Change

Fixing an error such as being incapable of processing the Euro may not lead to structural change per se: The engineer could just change every occurrence of the guilder or the mark into an occurrence of the Euro. Although this may affect a large part of the system (and typically will, in legacy systems) the actual structure of the program may not have to be changed. If however the engineer decides to fix the problem once and for all, then he may decide to put all occurrences of money in a single part of the system, and have all other parts of the system reference this part if necessary. Then, at the introduction of the global unified currency he will only have to change his one money part, and the rest of the system will be unaffected, thus the problem will be reduced to a minor change.

The process of finding and relocating all references to (in this case) parts of the system that relate to the processing of a particular currency is deemed structural

change. The information needed to perform such a change is much more detailed and harder to obtain than the information needed for minor changes. For example, in the previous section, it didn't matter very much if the information available only lead to the first occurrence of the problem: the aim of the engineer was to get the system running again. The second occurrence of the same problem may not appear that night, or only hours later: in the mean time, valuable computations might have been performed that may not have been performed if the engineer spent his time looking for other occurrences of the same problem.

When performing a structural change, the engineer would definitely have to find any and all references to currencies: if he would miss a single database retrieval operation, then guilder amounts may be subtracted from euro amounts or vice versa with all commercial and possibly legal consequences that may have.

When performing structural changes, time is usually less of a constraint than with minor changes. Accuracy is instead of extreme importance. Especially considering that the above example is one of the less complex forms of structural change.

More complex structural changes would be to change a system consisting of an online and a batch subsystem (for instance a checking account management system) into a client/server system. Here all the parts of the system that have to do with batch processing (typically the transferral of money from one account to the other) and the parts that have to do with online querying (balance retrieval, adding and removal of transfers, etcetera) are separated completely. At a certain moment each day (say 7 PM) the system moves from online mode to batch mode. At that time, balances can no longer be requested, and transfers can no longer be added or removed. Changing such a software system to (for instance) a client/server system that does not have harsh restrictions on which action can be performed at what time is a very complex structural change.

A final example of complex structural change of a software system is a change of implementation language. Legacy systems are typically implemented in languages that lack certain features that make software written in modern languages better understandable and maintainable. A form of modernization can be to re-implement the system in a different language.

1.3 Research Questions

This thesis does not deal with producing better software. It does not try to help people deliver fault-free software, or software that does not have to be changed. Instead it tries to help in making existing systems more flexible by supporting the engineers who need to change them with the appropriate technology. The concrete research questions that this thesis tries to answer are:

- How can we reduce the search space when searching for a particular artifact

in a legacy software system?

- How can we obtain sufficiently detailed information about a legacy software system to perform a structural change (semi-)automatically?
- How can we alter the software engineering process such that we no longer produce legacy systems?

Reduce Search Space The process of relating a certain externally perceivable feature of a software system to an internal artifact (the *traceability* of a feature) is one of the most frequently occurring activities in software maintenance. When a system contains millions of lines of code, automatically reducing the search space for a particular artifact can save hours if not days per search. Ideally, the search space is decreased to exactly those parts of the system that do something with the externally perceived feature.

Information for Structural Change Getting detailed information out of a legacy system *per se* is not necessarily hard. The problem is that the amount of detailed information may be (and usually is) so enormous, that it is useless unless processed (semi-)automatically. That is, the raw data extracted from a legacy system should be processed somehow to diminish in volume, but to increase in usefulness. An example could be that retrieving all database operations from a legacy system returns an enormous amount of data. Cross-referencing all the operations with the database schema, and filtering out only the operations on numbers gives back less data. If it would be possible somehow to filter the operation out even further to get only operations on numbers that represent a certain currency, then this would be the exact data needed for a currency conversion project.

Altering the Software Engineering Process As was described in Section 1.1.1 traditional software engineering hardly concerns itself with maintenance explicitly. If it does, it is seen as a completely separate activity from software development, to be performed by different people than the original developers. There are no practical steps in the software engineering process to prevent a system from becoming a legacy system. In fact, some people state that “programming leads to legacy systems” [Vis97b]. From that thesis follows that one way to prevent your system from becoming a legacy system is to never program it in the first place. However, there may be a slightly less drastic modification that can be made in the software engineering process to get rid of legacy systems.

1.4 Reader's Roadmap

As stated earlier, there is a sliding scale from performing a minor change on a software system to performing a structural change. This thesis concerns itself with these two types of changes and the information required to perform them.

The first chapters, "Rapid System Understanding" (Chapter 2) and "Building Documentation Generators" (Chapter 3) deal with retrieving and presenting information from a legacy software system in such a way that a maintenance engineer can retrieve a particular artifact in the system with maximum speed.

The next chapter, "Identifying Objects with Cluster and Concept Analysis" (Chapter 4) examines what data from a legacy system is required to derive a so-called object oriented design from that system. In an object-oriented design, procedures and the data they operate on are logically grouped. Chapter 4 examines two methods of automatically performing such a grouping and examines the pros and contras of each method.

One of the methods examined in Chapter 4 is used in the next chapter ("Types and Concept Analysis for Legacy Systems", Chapter 5) to group data in a different way. Here, data is grouped based on the way it is used to calculate new data. This way, we can get answers to questions like: what pieces of data in this system represent a monetary value, or what pieces of data represent a date, or an account number.

Where Chapters 4 and 5 mainly deal with the presentation and filtering of already extracted data from a system, Chapter 6 ("Object-Oriented Tree Traversal with JJForester") shows a technique for retrieving the actual detailed elements from the source code in an elegant and easy way.

Chapter 7 sketches a possible scenario of what could happen when a team of software maintainers tries to adopt the software engineering methodology called Extreme Programming.

The final chapter (Chapter 8) draws conclusions and examines how the research questions posed here have been answered.

Sources of the Chapters

Chapter 2, "Rapid System Understanding", was co-authored by Arie van Deursen. It was published earlier as:

> A. van Deursen and T. Kuipers. Rapid system understanding: Two COBOL case studies. In S. Tilley and G. Visaggio, editors, *Sixth International Workshop on Program Comprehension; IWPC'98*, pages 90–98. IEEE Computer Society, 1998.

Chapter 3, "Building Documentation Generators", was co-authored by Arie van Deursen. It was published earlier as:

> A. van Deursen and T. Kuipers. Building documentation generators. In *International Conference on Software Maintenance, ICSM'99*, pages 40–49. IEEE Computer Society, 1999.

Chapter 4, "Identifying Objects with Cluster and Concept Analysis", was co-authored by Arie van Deursen. It was published earlier as:

> A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *21st International Conference on Software Engineering, ICSE-99*, pages 246–255. ACM, 1999.

Chapter 5, "Types and Concept Analysis for Legacy Systems", was co-authored by Leon Moonen. It was published earlier as:

> T. Kuipers and L. Moonen. Types and concept analysis for legacy systems. In *Proceedings of the International Workshop on Programming Comprehension (IWPC 2000)*. IEEE Computer Society, June 2000.

Chapter 6, "Object-Oriented Tree Traversal with JJForester", was co-authored by Joost Visser. It was published earlier as:

> T. Kuipers and J. Visser. Object-oriented Tree Traversal with JJForester. In *Proceedings of the First Workshop on Language Descriptions, Tools and Applications 2001 (LDTA'01)*. Electronic Notes in Theoretical Computer Science 44(2). Elsevier Science Publishers, 2001.

Chapter 7 "Legacy to the Extreme", was co-authored by published earlier in "Extreme Programming Examined", published by Arie van Deursen and Leon Moonen. It was published earlier as:

> A. van Deursen, T. Kuipers, and L. Moonen. Legacy to the extreme. In M. Marchesi and G. Succi, editors, *eXtreme Programming Examined*. Addison-Wesley, Reading, Massachusetts, May 2001.

Chapter 2

Rapid System Understanding

This chapter describes the rapid extraction of facts from a legacy software system. When an engineer tries to relate a feature of a software system to an artifact inside that same system, he would like to know what parts of the system to look for, and what parts of the system to ignore. Rapid System Understanding investigates the techniques necessary to achieve that goal.[1]

2.1 Introduction

Rapid system understanding is the process of acquiring understanding of a legacy software system in a short period of time. Typical tasks that require rapid system understanding are:

- Assessing the costs involved in carrying out a European Single Currency or year 2000 conversion;
- Estimating the maintainability of a system, for example when deciding about accepting or issuing a maintenance outsourcing contract;
- Investigating the costs and benefits of migrating a system to an object-oriented language, in order to increase its flexibility and maintainability;
- Determining whether legacy code contains potentially reusable code or functionality.

[1] This chapter was published earlier as: A. van Deursen and T. Kuipers. Rapid system understanding: Two COBOL case studies. In S. Tilley and G. Visaggio, editors, *Sixth International Workshop on Program Comprehension; IWPC'98*, pages 90–98. IEEE Computer Society, 1998.

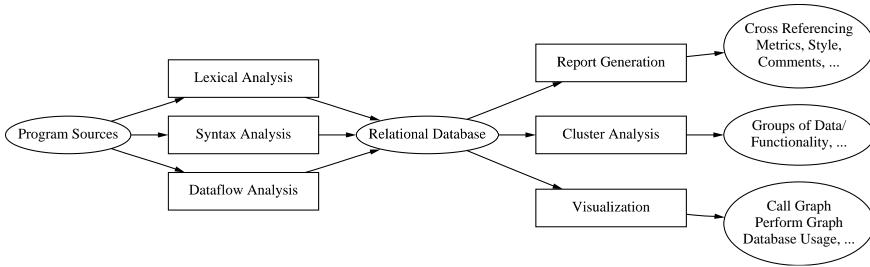


Figure 2.1: Architecture of tool set used

Performing these tasks should be cheap: one expects a cost estimate of, say, a year 2000 conversion to be significantly less expensive than carrying out that conversion. This is where rapid system understanding differs from more traditional system understanding. Accepting a less detailed understanding and slightly inaccurate results, a first assessment can be made quickly.

We assume that the engineer who needs to acquire understanding of a legacy system has negligible previous experience with it. He may be unfamiliar with some of the languages or dialects used in the legacy code. The systems involved are typically large, multi language, over 10 years old, and written by different programmers.

In this paper, we take two 100 KLOC COBOL systems from the banking area as our starting point. We address a number of related questions: What tools or techniques can be used in rapid system understanding? How well do they work for our case studies? What information can be extracted from legacy source code, and how should this information be interpreted?

The paper is organized as follows. In Section 2.2 we explain what tools and techniques can be used, and how these cooperate. In Section 2.3 we list the characteristics of the two COBOL systems under study. In Section 2.4 we describe the kind of information we extracted from the legacy code, while in Section 2.5 we discuss the possible interpretation of this data. In Sections 2.6, 2.7 and 2.8 we summarize related work, conclusions and future work.

2.2 Tool Architecture

Rapid system understanding implies summarizing of data. In order to understand a large legacy system, it is necessary to quickly find the "essence" of such a system. What constitutes this essence largely depends on the reasons for trying to understand the system.

Our approach is to analyze the code using generic tools that have no a-priori knowledge of the system. The results of this analysis are then fed into a central

repository. In turn, this repository can then be queried, printed, visualized, etc. The querying of the repository leads to a certain degree of understanding of the system. We can exploit this understanding by creating analysis techniques that do contain (a degree of) specific knowledge of the system. This will lead to data in the repository that is more suited for our specific goals. Again, this data can be visualized, queried, etc., to gain a greater understanding of the system. This process is repeated until the engineer who tries to understand the system has gained sufficient knowledge of it.

The general architecture of our tool set consists of three main parts, as shown in Figure 2.1. The first part is the code analysis part, the second the repository, and the third the tools that manipulate and present data from the repository.

For the code analysis part lexical, syntactic or other forms of analysis can be used. The figure distinguishes lexical, syntactic [dBSV97a], and data flow analysis. For the purpose of rapid system understanding, it will generally suffice to use lexical analysis. It can be performed faster than syntactic analysis, and is much more flexible [MN96].

To extract a number of relations from COBOL legacy systems, we have developed a simple Perl [WS91] script called `recover`. It knows about COBOL's comment conventions, keywords, sectioning, etc. It can be used to search the sources for certain regular expressions, and to fill tables with various relations, for example pertaining to the usage of databases, call structure, variable usage, etc. The data extracted for COBOL is discussed in full detail in Section 2.4.

We store the analysis results as comma-separated-value (CSV) files. Such files can be easily queried and manipulated by Unix tools such as `awk` [AKW88] and `join`, and can be read by arbitrary relational database packages enabling us to use SQL for querying the data extracted from the sources. These tools can also be used to generate reports, for example on the usage frequency of certain variables, or containing the fan-in/fan-out metric of sections of code.

Many relations stored in the repository are graphs. We use the graph drawing package `dot` [GKNV93] for visualizing these relations.

2.3 Cases Investigated

Central in our research are two COBOL systems from the banking area, which in this paper we will refer to as Mortgage and Share. Mortgage is a relation administration subsystem of a mortgage system. Share is the order-input (OI) subsystem of the ABN-AMRO stockbroking system. The respective owners of these systems are in general satisfied with their functionality, but less satisfied with their platform dependency. They are interested in extracting the essential functionality of these systems, in order to incorporate it into a more flexible, object-oriented, architecture. Thus, questions of interest include: Do these systems contain reusable

Mortgage	no	LOC	avg
copybooks	1103	49385	44
programs	184	58595	318
total	1288	107980	83

Share	no	LOC	avg
copybooks	391	16728	42
programs	87	104507	1201
total	479	121235	253

Figure 2.2: System inventory.

code? What fraction of the code is platform specific? Which data fields represent business entities? Which procedures or statements describe business rules?

The sizes of the two systems are summarized in Figure 2.2. Mortgage is a COBOL/CICS[2] application using VSAM[3] files. It is partly on-line (interactive), partly batch-oriented, and in fact only a subsystem of a larger (1 MLOC) system. Share is an IMS[4] application which uses both DL/I[5] (for accessing an IMS hierarchical database) and SQL (for accessing DB2 databases).

For Mortgage, we had system-specific documentation available, explaining the architecture and the main functionality of the programs. The documentation marked several programs as "obsolete": some of these were included in the version distributed to us, however. For Share, no specific documentation was available: we only had a general "style guide" explaining, for example, the naming conventions to be used for all software developed at the owner's site.

2.4 Collected Data

In this section, we discuss how we used the tool set of Section 2.2 to extract data from the Mortgage and Share sources, and how we displayed this data in a comprehensible manner. The results of the analysis will be discussed in Section 2.5.

2.4.1 System inventory

The system inventory table summarizes available files, sizes, types (copybook, program), and languages used (COBOL, CICS, SQL, DL/I, ...). The copybook table indicates how copybooks are included by programs (a simple lexical search for the arguments of the `COPY` command). If appropriate, for certain files (copybooks)

[2] CICS is Customer Information Control System, a user interface and communications layer
[3] VSAM is Virtual Storage Access Method, an access method for records
[4] IMS is Information Management System, a database and data communication system
[5] DL/I is Data Language 1, a database management language

it can be detected that they were generated, for example if they contain certain types of comment or keywords. The system inventory derived for Mortgage and Share was used to obtain Figure 2.2.

2.4.2 Program call graph

The call relation is a first step in understanding the dependencies between the programs in Mortgage and Share. Deriving the call graph for COBOL programs (see Figure 2.3 for the batch call graph of Mortgage) is not entirely trivial. First of all, the argument of a CALL statement can be a variable holding a string value, i.e., it can be dynamically computed. The most desirable solution to this problem is to have some form of constant propagation. In our case, for Mortgage it was sufficient to search for the values of certain variables or, in Share, for strings matching a certain lexical pattern.

In Share, we encountered further complications. Rather than a regular CALL statement, each call is in fact a call to some assembler utility. One of the arguments is a string encoding the name of the program to be called, as well as the way in which that program is to be loaded. The assembler routine subsequently takes care of loading the most recent version of this program. Once we understood this mechanism, it was relatively easy to derive the call graph using lexical pattern matching.

In Mortgage, the use of CICS provides further possibilities of calling programs. The first is the CICS LINK statement, which is similar to a CALL statement. The second is the CICS XCTL statement. This takes care of invoking a program just before or after an end-user has filled in a screen as presented in an on-line session. In Mortgage, the XCTL calls could be extracted by tracing the value of a specific variable.

Observe that these special call conventions imply that commercial reengineering tools should be sufficiently flexible to allow such organization-specific extensions. We have looked at two of the most advanced COBOL reengineering tools currenly available, Reasoning/COBOL [MNB+94] and MicroFocus/Revolve [Mic96]. Both support call graph extraction from abstract syntax trees, but neither is able to produce the on-line call graph of Mortgage or the batch call graph of Share. They can be adapted to produce these graphs, but that will be more time consuming than specifying a simple lexical search, making the latter option more attractive in a rapid system understanding setting.

2.4.3 Database usage

A viable starting point for locating data entities of interest is the data that the system reads from or stores in persistent databases. In Mortgage, VSAM files are used, and both COBOL as well as CICS constructs to access them. In Share,

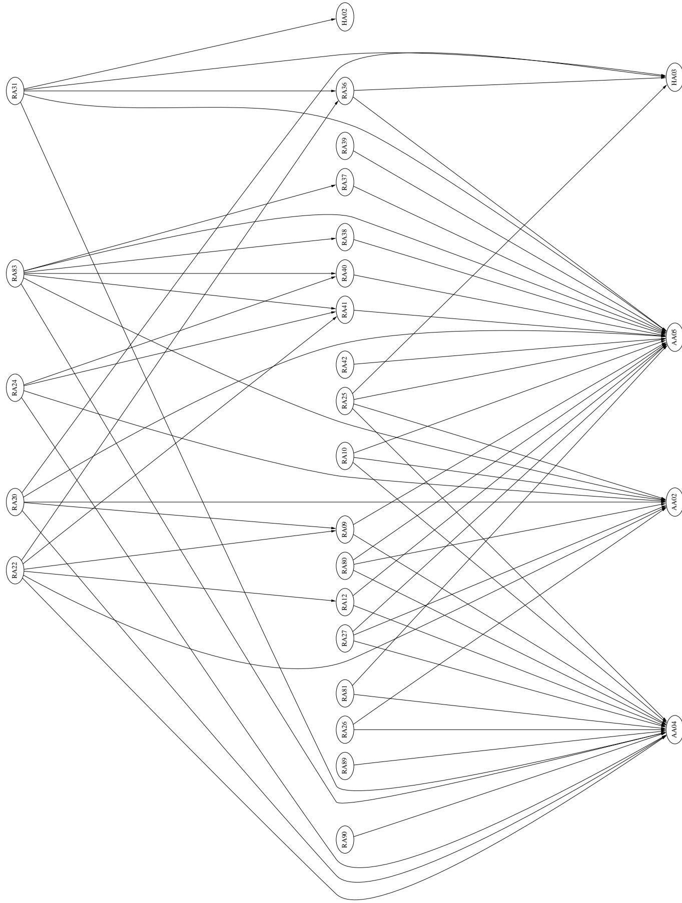


Figure 2.3: CALL graph for the batch part of Mortgage.

VSAM, hierarchical IMS and relational DB2 tables are used, and COBOL I/O statements, SQL and DL/I to access them.

In an SQL system, a datamodel listing all tables with their field names and types is explicitly available. The `recover` tool can be used to extract this model from the source. In a non-SQL application such as Mortgage, this datamodel is not available. What can be derived, though, is the COBOL record definition used when writing to or reading from files.

Share uses 36 different tables, with in all 146 different field names. The number of fields per table varies from 1 to 40 – suggesting that many tables share certain fields. To make this visible we generated a 60-page LaTeX document. For each table we have one section listing the fields and their types, as well as the programs in which the table was declared. We then used `makeindex` to generate an index, indicating at what pages the tables, fields, types, and programs were used.

The CRUD — create, read, update, delete — matrix indicates how databases are manipulated by programs. As viewing a CRUD matrix of a large system is cumbersome, we use the graphical representation of Figure 2.4. The left-hand column contains records read, the right-hand one records written, and the middle column lists the programs involved. An arrow from a record to a program indicates a read, and an arrow from a program to a record indicates a write.

2.4.4 Field usage

The database usage and datamodel provide a rough overview of the database operations per program. In many cases, it is useful to look at the table field level as well.

In order to expose shared use of database fields, we collect all occurrences of database field identifiers per section per program. From this table, we derive a list as shown in Figure 2.5. In Mortgage essentially 35 datafields are contained in one large table. Figure 2.5 shows how many of these datafields are used in each section. Of particular interest are those sections dealing with a small number (less than, say, 10) of data fields only.

We extracted the field usage relation using lexical analysis only. From the database declarations we extracted the field names. We then matched on section declarations to find section boundaries, and identified a field usage if one of the lines in a section contains a field name as substring. Clearly, this is an approximative method, relying on systematic naming conventions: the more accurate way would be to parse the source and do dataflow analysis to follow field usage through the code. For the two cases studied, though, the results obtained by lexical analysis were sufficiently accurate, at least for rapid system understanding purposes.

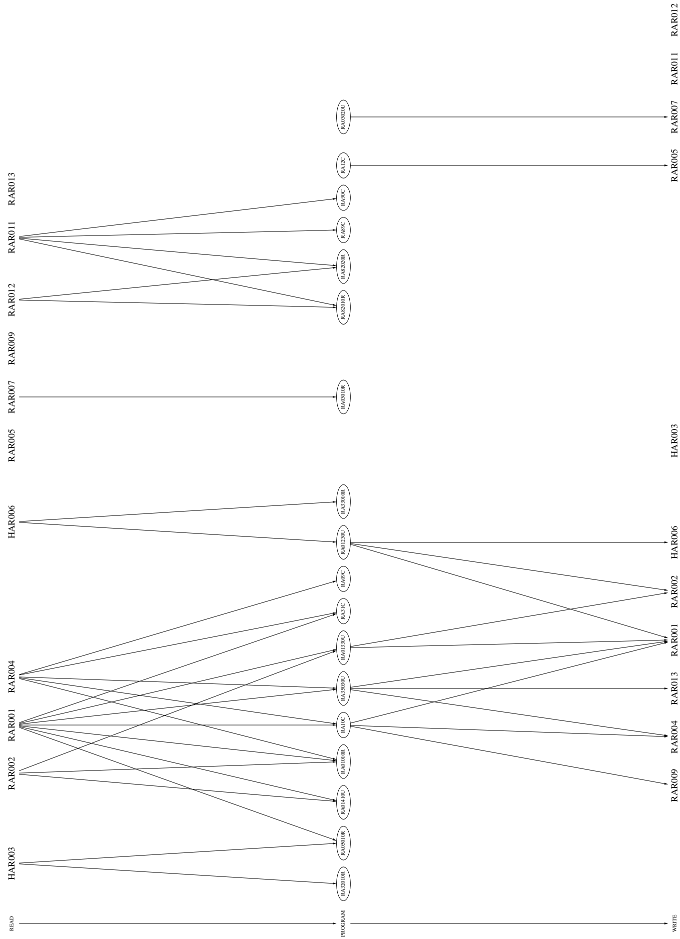


Figure 2.4: Graphical representation of the CRUD matrix of Mortgage.

Program	*Section*	*# vars*
RA01330U	B200-UPDATE-UIT-FIB	35
RA01230U	B200-UPDATE-UIT-FIB	35
RA01010R	C100-VUL-FIB	34
RA31C	R300-MBV-RELATIENR	32
RA31	R300-MBV-RELATIENR	32
RA20	R220-VERWERK-30-31	26
RA20	R210-VERWERK-20-21	25
RA20	R200-VERWERK10	25
RA35010R	B300-VALIDATIE	16
RA33010R	B300-VALIDATIE	16
RA20	R500-SPAARHYPOTHEEK	13
...	...	...

Figure 2.5: Number of record fields used per section.

Program	*Section*	*Fan-in*	*Fan-out*
RA20	R320-LEES-HAB006	23	1
RA20	R330-DATUM	16	2
RA83	R995-CLOSE-FILES	12	0
RA22	R995-CLOSE-FILES	12	0
RA80	R30-PRINT-REGEL	10	0
RA23	R995-CLOSE-FILES	9	0
RA20	R995-CLOSE-FILES	8	0
RA24	R995-CLOSE-FILES	7	0
RA80	R70-WRITE-RAB011S	5	0
RA38	R300-VERTAAL-GROOT-KLEIN	5	0
...	...	...	...
RA26	R995-CLOSE-FILES	3	0
RA23	R60-REDSEQ-RAB008	3	1
RA20	R212-VUL-OUD	3	3

Figure 2.6: Sections sorted by fan-in and fan-out.

2.4.5 Section call graph

A last relation we experimented with was the call graph at the section and paragraph level (the PERFORM graph). The perform relation can be used to visualize the calling structure of sections in individual programs. Since there are many programs, this is only useful if a selection of interesting programs is made beforehand (e.g. using the CALL graph).

Apart from visualizing the perform graph per program, all sections of the system can also be listed and sorted according to some metric. Figure 2.6 shows all sections with a fan-in of at least 3, and a fan-out of at most 3. It can be used to search for sections with an attractive fan-in/fan-out ratio.

At the system level, the sections included in copybooks are of particular interest. These sections were designed to be reusable, and therefore are natural candidates for further reuse. Figure 2.7 lists some of these sections for Mortgage, together with the number of other programs and sections containing a call to that section (fan-in).

Section	*Performed*	*Sections*	*Programs*
Y800-PRINT	145	17	12
Y998-HANDLE-ERROR	92	92	92
Y010-40-AIB	89	89	89
Y010-00-AIB	81	81	81
Y502-MASKER	80	25	25
Y020-00-FIB	79	79	79
Y020-40-FIB	66	66	66
...	...	...	...
Z610-82-RAB011	7	5	4
Z610-80-RAB011	7	6	5
Y675-STD-STRAAT	7	4	2
Y625-ELF-PROEF	7	6	6
Y415-INFO-SCHONEN-NIT	7	3	3
Z610-03-RAB011	6	4	4
Y750-STD-NAAM	6	5	3

Figure 2.7: Sections performed by different sections and programs.

2.4.6 Further experiments

While finding out how to extract the various relations from the sources, we also used `recover` as an enhanced COBOL lexer. The `recover` script contains several functions to ignore COBOL comment columns and lines, to split lines into strings, numbers, identifiers, keywords, and other tokens, to search for arguments of keywords, to expand copybooks, to record information encountered in earlier lines, and to store results into tables. These functions were fruitfully used to acquire an understanding of conventions used, relevant data, etc.

2.5 Interpreting Analysis Results

In this section, we discuss how the graphs and reports derived in the previous section helped us to actually understand the two COBOL systems under study.

2.5.1 Understanding copybooks

Rapid system understanding is a mixture between looking at global system information like the call graph and looking in more detail at a specific program in order to obtain a feeling of its particularities. One of the aims during rapid system understanding is to reduce the number of programs that need to be studied in detail. Having the copybook relation explicitly available will help to avoid looking at copybooks that are in fact never included.

For Share, 136 of the 391 (35%) copybooks were not used; for Mortgage 673 of the 1103 (61%) were not used. These large numbers can partly be explained as Mortgage is part of a larger system: for safety, all copybooks were included. Likewise, Share relies on general utilities used at the owner's site; to be safe many

of these were included. We have not yet looked at other forms of dead code, such as sections or programs never called. To detect the latter, one should also have all JCL[6] scripts available, which we did not have for our case studies.

Another use of the copybook relation is to identify patterns in the copybook inclusions. It turned out, for example, that the batch and the on-line part of Mortgage use two almost disjoint sets of copybooks.

2.5.2 Call graph and reusability

The batch call graph for a part of Mortgage is shown in Figure 2.3. This graph shows particularly well that we can identify:

- Programs with a high fan-out. From inspection we know that these are typically "control" modules. They invoke a number of other programs in the appropriate order. In Figure 2.3, they are mostly grouped in the left-hand column.

- Programs with a very high fan-in, i.e., called by most other programs. These typically deal with technical issues, such as error handling. From inspection it is clear that they are tightly connected to legacy architecture, and are not likely to contain reusable code. In Figure 2.3, they are grouped in the right-hand column.

- Programs with a fan-in higher than their fan-out, yet below a certain threshold. These programs can be expected to contain code that is reusable by different programs. In Figure 2.3, they are mostly in the middle column. These programs form the starting point when searching for candidate methods when trying to extract an object-oriented redesign from legacy code.

For the batch part of Mortgage, this categorization worked remarkably well.

The call graph based on CICS LINK commands (not shown) contains the remaining calls. Due to the presence of error handling modules, this call graph was difficult to understand. Removing all modules with fan-in higher than a certain threshold (say 10), we obtained a comprehensible layout.

For Mortgage, this analysis of the call graph led to the identification of 20 potentially reusable programs that performed a well-defined, relatively small task.

For Share, only 50% of the programs were contained in the call graph; the remaining programs are called by JCL scripts, which we did not have available. Therefore, for Share further ways of identifying reusable code will be required.

At a finer granularity, analysis of the PERFORM graph will be an option. In principle, the same categorization in control, technical, and potentially reusable code can be made. The derived table of Figure 2.6 can help to find sections of an

[6]JCL is Job Control Language, a shell-like system for MVS.

acceptable fan-in. Clearly, this list is rather large, and not all sections will be relevant: we decide to inspect the code of a section based on its name. For example, we could probably ignore CLOSE-FILES, but should take a look at VERTAAL-GROOT-KLEIN[7], especially since this section occurs in three different programs.

Analysis of the sections included in copybooks as shown in Figure 2.7 will proceed along the same lines: based on the number of perform statements and the name of the section (for example, STD-STRAAT, indicating processing of a STRAAT, i.e., street), we will inspect code of interest. Surprisingly, Share does not use any sections defined in copybooks.

2.5.3 Understanding data usage

The tools provide three ways to understand the data usage. The first is the index of tables, attributes, types and programs derived from SQL data definitions. This index can be used to detect, for example, sets of attributes that occur in several different tables and hence may be (foreign) keys.

The second aid is the CRUD matrix, which shows which programs read or write certain tables. We used Figure 2.4, which shows the database usage for Mortgage, to identify databases that are only read from (for example the "zip-code book", shown in the top-left corner of Figure 2.4) or only written to (logging, shown in in the top-right corner), databases used by many programs, or programs using many databases. We also used this figure to identify those databases that are used by most other programs. For Mortgage, there are only three such databases. The tools indicate which (level 01 COBOL) record definitions are used to access these, and the fields in these records we considered as the essential business data of Mortgage.

The third possibility aims at finding out how these data fields are used throughout the system, using Figure 2.5. It can help to group data fields based on their shared use, or to locate sections of interest, potentially containing core business functionality. Again, we will use this list to select sections of interest manually. Examples are sections called VALIDATIE, which contain code for checking the validity of fields entered via screens.

2.5.4 Reusability assessment

In the preceding sections, we have discussed how certain information can be extracted from COBOL sources using lexical analysis methods (Section 2.4) and how we can use this information to understand the legacy system at hand (Section 2.5). Does this acquired understanding help us to answer the questions posed in Section 2.3?

[7] Dutch for *map-upper-lower*

- *Do the systems contain reusable code?* Based on the call graph several programs and sections were identified which, after inspection, turned out to contain well-isolated reusable functionality.
- *Which data fields represent business entities?* The tools help to identify those data fields that are written to file and used by most programs: the indexed representation helps to browse these fields and filter out certain fields that are perceived as "non-business".
- *Which statements describe business rules?* An inventory of the data fields used is made per section: those dealing with several fields are likely to describe business-oriented procedures.
- *What fraction of code is platform-specific?* Of the 340 sections of Share 177 (approximately 50%), refer to at least one data field. Thus, an initial estimate is that the other 50% is likely to contain platform-specific code. For Mortgage, 510 of the 2841 sections (only 18%) refer to the data items stored on file. Thus, 82% appears to be platform-oriented rather than business oriented. Inspection of the functionality shows that this is the case: a large part of Mortgage deals with CICS-specific details (implementing a layer on top of CICS).

Evidently the answers to these questions are approximative. If a full reuse, reengineering, or re-implementation project is to be started, this project will require a more detailed answer to these questions. In order to decide to embark upon such a project, fast answers, such as those discussed in this section, obtained at minimal costs, are required.

2.6 Related Work

Lexical analysis of legacy systems Murphy and Notkin describe an approach for the fast extraction of source models using lexical analysis [MN96]. This approach can be used for the "analysis" phase (as showed in Figure 2.1), in stead of `recover`. Murphy and Notkin define an intermediate language to express lexical queries. Queries composed in this language are generally short and concise. Unfortunately, the tool was not available, so we were not able to use this tool for our COBOL experiments.

An interesting mixture between the lexical approach of AWK and matching in the abstract syntax tree is provided by the TAWK language [GAM96]. Since TAWK is not (yet) instantiated with a COBOL grammar, however, we could not use it for our experiments.

Reengineering tools There are a number of commercial reengineering tools that can analyze legacy systems, e.g. [Mic96, MNB+94]. They are either language-specific (mainly COBOL), or otherwise based on lexical analysis. Lexical analysis provides a level of language independence here, and makes the system easier to adapt to new languages and dialects.

The COBOL specific reengineering tools have built-in knowledge of COBOL: They work well if the application at hand conforms to the specific syntax supported by tool, usually the union of several COBOL dialects.

Examples of language-independent tools are Rigi [MOTU93] or Ciao [CFKW95].

Many papers report on tools and techniques for analyzing C code. We found it difficult to transfer these to the COBOL domain and to apply them to our case studies. COBOL lacks many C features, such as types, functions, and parameters for procedures. Moreover, approaches developed for C tend not to take advantage of typical COBOL issues, such as the database usage for business applications.

Finding reusable modules Part of our work is similar in aims to the RE^2 project, in which candidature criteria have been defined to search for functional abstractions, data abstractions, and control abstractions [CV95]. The RE^2 approach has been applied to COBOL systems by Burd *et al.* [BMW96, BM97].

Neighbors [Nei96] analyzes large Pascal, C, assembly, and Fortran systems consisting of more than a million lines of code. One of his conclusions is that in large systems, module names are not functional descriptions, but "architectural markers". This agrees with our observation that we could not use module names to locate reusable code, while section names proved helpful in many cases.

2.7 Conclusions

Rapid system understanding, in which fast comprehension is more important than highly accurate or detailed understanding, plays an important role in the planning, feasibility assessment and cost estimating phases of system renovation projects.

System understanding tools require an architecture in which it is easy to exploit a wide range of techniques. The architecture discussed in Section 2.2 distinguishes analysis of source code, a central relational database to store analysis results, and various forms of presenting these results such as report generation and graph visualization.

The datamodel used by the relational database, and the analysis and visualization techniques used, depend on the rapid system understanding problem at hand. The paper discusses an instantiation for identifying reusable business logic from legacy code.

Lexical analysis, using only superficial knowledge of the language used in the sources to be analyzed, is sufficiently powerful for *rapid* system understanding.

An important advantage is its flexibility, making it possible to adapt easily to particularities of the system under consideration. (See, for example, the derivation of the call graph of Share as discussed in Section 2.4).

In order to assess the validity of the architecture proposed, the emphasis on lexical analysis, and the instantiation used for identifying business logic from legacy code, we studied two COBOL case studies from the banking area. The two case studies show that (1) lexical methods are well-suited to extract the desired data from legacy code; (2) the presentation forms chosen help us to quickly identify business data fields and chunks of code manipulating these fields; (3) the proposed approach is capable of finding answers to the reusability questions posed in Section 2.3.

We consider the results of this case study to be encouraging, and believe the approach to be viable for a range of system understanding and reusability assessment problems. The limitations of our approach are:

- Lexical analysis cannot take advantage of the syntactic structure of the sources. In our cases, for example, it is difficult to extract those variables that are used in, say, conditions of if-then-else statements.
- Identification of business data is based on the assumption that this data is stored in databases.
- Identification of data field usage is based on textual searches for the field names. This works on the assumption of systematic variable naming. A more accurate, yet also much more involved, method would be to follow the database field usage through the dataflow.

The latter two assumptions are reasonable and will generally hold, but certainly not for all systems.

2.8 Future Work

While developing the rapid system understanding tools and techniques, and while applying them, several further research questions emerged. We are in the process of investigating the following topics.

Use of metrics Our work bears a close relationship with the area of metrics. A question of interest is what metrics are indicative for reusability in the two COBOL systems we studied. Another relevant question is which metrics can be computed sufficiently easily, in order to make them applicable in a rapid system understanding setting.

Code cloning While analyzing Mortgage we observed a high degree of duplicated code. We intend to investigate whether lexical methods are suitable for detecting the clones present in Mortgage.

Restructuring and modularization We are currently experimenting with applying cluster analysis methods to remodularization of legacy code [DK97].

Comprehension models Searching through code using lexical analysis can be viewed as browsing in order to answer questions and verify hypotheses. Recent studies in system and program understanding have identified code cognition models emphasizing this hypothesis verification aspect [Bro83, MV96, MV97]. From our experience with the two COBOL cases we observed that many of our actions were aimed at *reducing the search space*. Thus, rather than verifying hypothesis immediately, we started by organizing the set of programs such that the chance of looking at less relevant programs was minimized. It seems interesting to study how this search space reduction fits in some of the existing code cognition models.

As a last remark, the present year 2000 crisis may be an ideal opportunity to experimentally verify the validity of cognition models for rapid system understanding.[8] Many "year 2000 solution providers" start by performing a "quick scan" in order to determine the costs of the year 2000 conversion project, and almost all of these scans are based on lexical analysis. A successful cognition model should be able to describe most of the methods used by these solution providers, and might be able provide hints for improvements for methods not taking advantage of this model.

[8]This chapter was written and published before 2000. In the interim, the Y2K crisis has come and gone. The remark, however, still stands: Problems similar to Y2K keep cropping up and keep forming a fertile testbed for verification of cognition models. Large scale minor change problems in the foreseeable future include expansion of bank account numbers in The Netherlands, the expansion of internet protocol (IP) numbers, the standardization of measurement units (e.g. from cubic meters to kilowatts in the natural gas industry) and many more.

Chapter 3

Building Documentation Generators

This chapter integrates the analysis results presented in the previous chapter. It adds hypertext as a presentation form, which allows an engineer to *browse* through a system, moving from general overview to detailed information with a couple of mouse clicks. Retrieving facts from the legacy system is facilitated by the use of *island grammars*, an analysis technique which couples the flexibility of lexical analysis with the thoroughness of syntactic analysis.[1]

3.1 Introduction

The documentation of a system is needed to understand that system at a certain level of abstraction, in a limited amount of time. It is needed, for instance, if a system is migrated or re-engineered. It can be used to map functional modification requests as expressed by end users onto technical modification requests, and to estimate the cost of such modifications. Finally, documentation will help in the process of outsourcing maintenance or when engineers that are new to the system need to learn about the system.

The source code of a system can be viewed as its most detailed level of documentation: All information is there, but usually we do not have enough time to comprehend all the details. Luckily, we do not usually need to know *all* the details. Instead, we would like to have enough information so that we can build a *mental*

[1]This chapter was published earlier as: A. van Deursen and T. Kuipers. Building documentation generators. In *International Conference on Software Maintenance, ICSM'99*, pages 40–49. IEEE Computer Society, 1999.

model of the system, and *zoom in* to the specific details we are interested in. The level of detail (or abstraction) we are interested in depends very much on what we intend to do with the system.

This flexibility should be reflected in the documentation, which, therefore, should adhere to four criteria:

1. Documentation should be available on different levels of abstraction.

2. Documentation users must be able to move smoothly from one level of abstraction to another, without loosing their position in the documentation (zooming in or zooming out).

3. The different levels of abstraction must be meaningful for the intended documentation users.

4. The documentation needs to be consistent with the source code at all times.

Unfortunately, these criteria are not without problems. Criterion 4 implies that documentation is generated from the source code. In practice this is seldomly done. Consequently, it is violated by many legacy systems, which are modified continuously without updating the accompanying technical documentation.

Criterion 3 makes documentation generation hard. Meaningful abstractions can benefit immensely from design information which is usually not present in the source code itself. Such information needs to be added manually to the documentation.

For new systems, mechanisms like literate programming [Knu84] provide systematic ways of putting design information in the source code. For legacy systems this would involve a significant manual updating of program comments. Besides, design information is more often than not lost for legacy systems.

In this paper, we study ways in which we can update the documentation of legacy systems such that all four criteria are met. We propose a combination of manual and automatic (re)documentation. Whatever documentation can be generated from the sources is derived automatically. This then is combined with information provided by hand. Depending on the state of the system, and the knowledge about the system, either one of those activities can play the predominant role in the final documentation that is delivered. Figure 3.1 shows the architecture of the documentation generators that are built this way.

The remainder of this paper is organized as follows. In the next section, we introduce *island grammars*, the technology we use for extracting facts from a system's source code. In Section 3.3 we discuss what information should be contained in documentation, and how we can derive it from the legacy sources. In Section 3.4 we explain how the information extracted can be presented at what level of abstraction, using graph visualization and hypertext as primary tools. In Section 3.5 we describe a real-world Cobol legacy system, what its documentation

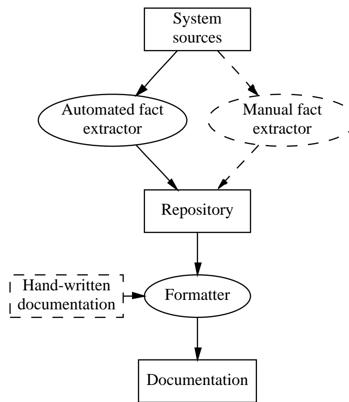


Figure 3.1: Deriving documentation from legacy sources. Solid lines indicate automatic processing, augmented with manually derived information indicated by dashed lines.

problems were, and how we applied the techniques described in this paper to build a documentation generator for that system. We end the paper with related work, a summary of the main contributions, and suggestions for future work.

3.2 Source Code Analysis

In order to generate documentation from a system, we need to analyze the source code of that system. We have tried several analysis approaches. In this section we will discuss these approaches in detail. In later sections we discuss how we have used these approaches.

3.2.1 Lexical analysis

When generating documentation for a system, only a few constructs in the source code are of interest. After all, the documentation should be a useful abstraction of the system. The constructs of a language that are of interest very much depend on the type of documentation that should be generated. If these constructs have an easily recognizable lexical form, lexical analysis is an efficient way to find them. If, for instance, we are looking for files that are opened for reading in a Cobol source, we simply look for the string "open input" and take the word directly following that string as the file handle that has been opened.

The advantage of this approach is that we do not need to know the full syntax of the language we want to analyze. Another advantage is that lexical analysis is very efficient. This allows us to analyze large numbers of files in a short time, and

also allows us to experiment with different lexical patterns: If a pattern does not yield the correct answer, the analysis can be easily changed and rerun.

The main disadvantage of lexical analysis is that it is (in general) not very precise, and that some language constructs are much harder to recognize lexically than others. For example, for the case study later discussed in this paper we need to find the files that were executed from a DCL program, the DEC job control language for VAX VMS. In DCL, we can look for the string "run", which is the DCL keyword for execution. If, on the other hand, we would want to know which files are executed from a Bourne shell script, we would need to specify all built-in functions of the Bourne shell language. There is no special keyword for execution in the shell, rather, it attempts to execute all words that are not built-in functions.

Strings such as "open input" and "run" obviously can occur in different contexts, and may mean completely different things in each context. These strings could occur in comment, for example, or inside a quoted string. Because we need to recognize different contexts in most cases, much of the original simplicity of the lexical pattern is gone. Furthermore, as long as we do not specify the full syntax of a language, there is the risk that we may have overlooked particular contexts in which a pattern can or cannot occur.

Most commonly used for lexical analysis are Unix tools such as `grep`, `awk`, and `perl`. Murphy and Notkin [MN96], describe LSME, a system which allows for the lexical specification of contexts of patterns, as well as the patterns themselves. For the analysis of Cobol, we have developed `recover` [DK98], which keeps track of the global structure of Cobol, and allows the user to specify patterns typically required in a program understanding context.

3.2.2 Syntactic Analysis

More precise analysis of source code can be achieved by taking the syntactic structure of the code into account, analyzing the abstract syntax tree instead of the individual source code lines. This makes the context in which a particular construct occurs explicitly available. Moreover, it abstracts from irrelevant details, such as layout and indentation.

Unfortunately, most legacy systems are written in languages for which parsers are not readily available. Developing a grammar from which to generate such a parser requires a significant investment. As an example, Van den Brand *et al.* [dBSV97b] report a period of four months needed to develop a fairly complete Cobol grammar.

For program understanding and documentation purposes, however, only a handful of language constructs are needed, so it seems too much work to have to specify the full grammar of a legacy language. Therefore, we propose the use of "island grammars", in which certain constructs are parsed in full detail, whereas others are essentially ignored.

syntax

$$
\begin{array}{rcll}
\text{Stat+} & \rightarrow & \text{Program} & (3.1) \\
\sim [\backslash.] + \text{"}\backslash.\text{"} & \rightarrow & \text{Stat} & (3.2) \\
\text{"}\texttt{SELECT}\text{" FileHandle "}\texttt{ASSIGN}\text{"} & & & \\
\text{"}\texttt{TO}\text{" FileName Option} * \text{"."} & \rightarrow & \text{Stat} & (3.3) \\
\text{Name} + (\text{"}\texttt{IS}\text{"})?\ \text{Value} & \rightarrow & \text{Option} & (3.4) \\
\text{Id} & \rightarrow & \text{FileName} & (3.5) \\
\text{Id} & \rightarrow & \text{Name} & (3.6) \\
\text{Id} & \rightarrow & \text{Value} & (3.7) \\
\text{Id} & \rightarrow & \text{FileHandle} & (3.8) \\
[\texttt{A} - \texttt{Z}][\texttt{A} - \texttt{Z0} - \texttt{9}\backslash_]* & \rightarrow & \text{Id} & (3.9)
\end{array}
$$

priorities

$$
\begin{array}{rcl}
\text{"}\texttt{SELECT}\text{" FileHandle "}\texttt{ASSIGN}\text{"} & & \\
\text{"}\texttt{TO}\text{" FileName Option} * \text{"."} & \rightarrow & \text{Stat} \quad > \\
\sim [\backslash.] + \text{"}\backslash.\text{"} & \rightarrow & \text{Stat}
\end{array}
$$

Figure 3.2: An example island grammar

3.2.3 Island Grammars

An island grammar consists of (1) detailed productions for the language constructs we are specifically interested in (2) liberal productions catching all remaining constructs; and (3) a minimal set of general definitions covering the overall structure of a program.

As an example, suppose we have a simple language L. Programs in L consist of a list of one or more statements. For documentation generation purposes we are only interested in one statement, the “SELECT” statement. The definition of the island grammar is in Figure 3.2. We use the grammar definition language SDF2 [Vis97b] for our definition.[2] We can distinguish the following groups of productions:

- The definition of the statement of interest is on line (3), defining a a statement to be produced by the keywords “SELECT”, a FileHandle, “ASSIGN”, “TO”, a FileName, a possibly empty list of Options, terminated with a “.” character. Productions (4–9) define the details of the other non-terminals.

[2] Please note that productions in SDF2 are reversed with respect to languages like BNF. On the right-hand side of the arrow is the non-terminal symbol that is produced by the symbols on the left-hand side of the arrow.

- The liberal production catching all remaining constructs is on line (2), defined as any character that is not a "." (the tilde negates the character class containing the period), followed by a period.

  Obviously, this grammar is ambiguous, because a "SELECT" statement can be produced by both productions (2) and (3). To resolve this, Figure 3.2 defines **priorities** preferring production (3) to (2).

- Line (1) defines the overall structure of a program, which is defined as a list of statements.

The reason for using this grammar development technique, is that we significantly reduce the grammar development time. Another advantage is that the parse tree that is returned by the parser only contains the relevant information. We do not have to weed through dozens of complicated structures to get to the information we look for.

By far the biggest advantage is the flexibility of the technique. Although some legacy languages have a proper language definition, we have yet to see a legacy system that does not use compiler specific extensions, or locally developed constructs. Furthermore, most parsers for legacy systems are quite liberal in checking their input, so although a program is not syntactically correct according to the language definition, it does parse, compile, and run. Using our grammar development technique, we can either ignore these specifics (by writing a *catch-all* production such as (2) above), or add a production particular to a certain extension of the legacy system at hand.

In principle, island grammars can be used in combination with any parser generator, the best known ones being Yacc and Bison. We benefited from the use of SDF2, which has a number of attractive characteristics.

First, SDF2 is based on *scannerless* parsing, in which the distinction between lexical scanning and parsing has disappeared. Hence, the user of SDF2 is not restricted to regular expressions for defining lexical tokens. Moreover, explicit lexical disambiguation is permitted in the formalism.

Second, parsers for SDF2 are implemented using *generalized* LR parsing [Vis97a], which accepts arbitrary context-free grammars, not just LALR grammars accepted by Yacc and Bison. This avoids the notorious shift reduce conflicts inherent to the use of LALR grammars. A priority mechanism can be used to deal with ambiguities that may arise due to the use of arbitrary context-free grammars.

Last but not least, because SDF2 is a *modular* syntax definition language, we can specify an island grammar in different modules. This way, for each analysis we can have a different grammar that is an extension of a common core language. This helps to keep the grammars as small and concise as possible. Consider the island grammar developed above. Here, productions (1), (2), and (5–9) can be viewed as being part of the core of language L. These can be put in a separate module. Then,

the only productions needed for our "SELECT" analysis are productions (3–4), and the priority rule, which should be defined in a different module.

3.2.4 Parse Tree Analysis

The parser generated from the grammar in the previous section will return parse trees that can be easily analysed. The parse trees are encoded in *aterm* format [dBKV96]. This parse tree can be read in by a Java framework we wrote, thus giving access to the parse tree as a Java object. The framework implements the *visitor* design pattern [GHJV94], via a visitor class that can be specialized to perform a particular analysis on the tree. This is simplified by the fact that the Java framework has full knowledge of the island grammar that has been specified, and contains methods for matching patterns of productions in the grammar to corresponding nodes in the tree.

The analysis results that are of interest can be written to a repository, and from there they can be combined, queried and used in the rest of the documentation generation process. All extractions described in Section 3.3 were performed using this Java parse tree analysis framework. The data extracted were put in a repository. The presentations described in Section 3.4 were then generated from that repository.

This way of analyzing source code is similar in concept to a number of other systems, e.g. CIAO [CFKW95], in the sense that there is a chain of analysis, filter, and presentation events. In our approach, however, we start filtering the data during the first (analysis) phase, because we only deal with those language constructs defined in the island grammar.

3.3 Extracting Documentation

In this section, we will discuss the sort of information that should be contained in software documentation, and how this information can be identified in the legacy sources.

3.3.1 Manual versus Automated Extraction

Given the choice between manual or automatic extraction of information from source code automatic extraction (for example using island grammars) is the preferred option: it is consistent with the actual source code, and can be easily maintained by automatic regeneration.

If generation is not feasible, the facts needed to construct the documentation can be provided by hand. This may take the form of a list of programs and a one or two line description of their functionality. Whenever documentation is generated, data from this list is included as well. Moreover, automatic checks as to whether *all*

Level	Documentation
system	overall purpose, list of subsystems
subsystem	purpose, list of modules, batch jobs, databases, screens, ...
batch job	programs started, databases accessed, frequency, ...
program	behavior, programs called, databases read or written, invoked by, parameters, ...
section	functionality, external calls, sections performed, conditions tested, variables used, ...

Figure 3.3: Cobol system hierarchy, with associated documentation requirements

programs are indeed contained in the lists can be made whenever documentation is regenerated, encouraging programmers to keep the descriptions up to date. The integration of manual and automated extraction is illustrated in Figure 3.1, which also shows how additional forms of externally available documentation can be included in the resulting documentation.

3.3.2 System Decomposition

We can decompose a large software system into several layers of abstraction, ranging from individual procedures up to the overall system. At each level, we need documentation, helping us to answer questions about the purpose (why?) of a component, the subcomponents it consists of (part-of relationships), the components it needs to perform its tasks (uses relationships), the way in which it performs its tasks (how?), the way in which the component can be activated (usage conditions), the system requirements the component corresponds to, etc.

The actual splitting in layers of abstraction, and the corresponding documentation requirements, will differ from system to system. The hierarchy with associated documentation requirements we use for Cobol systems is shown in Figure 3.3.

3.3.3 Aggregation and Use Relations

The parts-of and uses relationships discussed in the previous section can be easily derived from the source code. In general, it is relatively straightforward to extract facts about calls, database usage, screens used, etc.

A factor complicating this extraction is that many legacy systems use non-standard conventions for, e.g., calling or database access. We have seen calling conventions in which all calls were redirected via an assembly utility, and database access conventions hiding all SQL operations via a set of Cobol modules. The

flexibility of island parsing makes it particularly easy to tailor the extractors to such conventions.

3.3.4 System and Subsystem Partitioning

At the system level, the directory structure of program files or the naming conventions used usually provide a candidate partitioning into subsystems. If these are absent, or perceived as inadequate, we use automatic subsystem classification techniques to arrive at a better partitioning [Lak97, DK99b]. Such alternatives can then be added to the documentation, helping the user to see component relations that do not immediately follow from the actual partitioning.

In addition to the decomposition of the overall system, short descriptions of the individual subsystems as well as of the overall behavior are needed in the documentation. In many cases, such top level documentation may already be available, in which case it can be included in the documentation generation process. If it is not, a description of the various subsystems should be added by hand.

3.3.5 Program Descriptions

In many systems, coding standards are such that each program or batch job starts with a *comment prologue*, explaining the purpose of this component, and its interaction with other components. If available, such a comment prologue is a very useful documentation ingredient which can be automatically extracted from the source. Observe that it is generally *not* a good idea to extract *all* comment lines from a program's source into its documentation: many comment lines are tightly connected to specific statements, and meaningless in isolation. Moreover, in many cases obsolete pieces of code have been "commented out", which clearly should not appear in system documentation.

3.3.6 Section Descriptions

For the sections (local procedures) of a Cobol program, it is usually not as easy to extract a description as it is for Cobol programs starting with a comment prologue. On the positive side, however, section names are generally descriptive and meaningful, explaining the purpose of the section. This is unlike Cobol *program* names, which generally have a letter/number combination as name indicating which subsystem it is part of, not what its purpose is.

Since we encountered an actual need for the documentation of sections that consisted of more than just the name, but at the same time was more abstract than simply the complete source code, we decided to search for ways in which to select the essential statements from a section. In terms of the theory of program comprehension as proposed by Brooks [Bro83], we try to select those statements

Figure 3.4: An example of the visualization of data dependencies in a DCL job

that act as *beacons* for certain understanding tasks, such as finding out under what condition a certain piece of code is being executed.

Statements we include in such section descriptions are the *conditions* checked in control statements, calls to external programs, database access operations, calls to other sections, statements containing arithmetic computations, and comment lines. This explicitly excludes the frequently occurring MOVE statement, which in Cobol is an assignment from one variable to another. As it does nothing "interesting" (no computation) leaving it out of the documentation directs the reader's attention to those statements that do perform some task of interest.

Following this strategy, the length of a summarized section is about one third of the length of its full source code. To make the summary as comprehensible as possible, we reuse indentation of the original source code, giving the reader a sense of the nesting level.

3.3.7 Batch Job Dependencies

For a Cobol mainframe application, the batch jobs determine which programs are started every day or night, and how datafiles used for communication purposes are sorted, renamed, etc. In many cases, the batch jobs are the least understood components of a Cobol system, in spite of their importance for the daily data processing. Documentation should help in understanding such batch jobs. We have experimented with visualizing the data dependencies in such batch jobs.

Finding the data dependencies for a batch job is a three step process. First, we identify the Cobol programs executed in a batch job. Second, we analyze these Cobol programs, determining which data files are read and which ones are written.

Third, we return to the batch files, to see whether these data files occur in them, for example for sorting or renaming.

Recognizing these dependencies involves two island grammars: one for the job control language, finding the execution, sort and renaming statements, and one for Cobol, identifying the data file manipulation statements.

Once the data dependencies are found, they can be visualized. The visualization of an example batch job is shown in Figure 3.4. The resulting graph only shows the functional dependencies: Dynamic dependencies, such as the order of execution, are not explicitly visible. Also observe that in some cases, it will be impossible to determine the name of a data file, because it is determined at run time. Special nodes in the graph are used to mark such file names.

3.4 Presenting Documentation

Once we have decided which information to put into the documentation, we can decide how to *present* that information to the user. Hypertext has been proposed as a natural way of presenting software documentation [Bro91, Raj97] as the hyperlinks can be used to represent, for example, part of and uses relationships between between the documented components.

The most natural way of organizing all the information derived is to follow the system hierarchy, producing essentially one page per component. For Cobol this would result in pages corresponding to the full system, subsystems, programs, batch jobs, and sections, following the decomposition of Figure 3.3.

If a user knows what programs he wants to read about, finding an initial node to start browsing is simple. In many cases, however, there may not be such a straightforward starting point. Therefore, we provide various indexes with entry points to the hypertext nodes, such as:

- Alphabetic index of program names;
- Keyword search on documentation contents;
- Graphs representing use relationships. In particular, navigating through a call graph may help to find execution starting points or modules frequently used. We have used the graph drawing package `dot` [GKNV93] to integrate clickable image maps for various call graphs and data-dependency graphs into generated documentation. In order to prevent visual cluttering of graphs, we have applied *node concentration* on them, as can be seen in Figure 3.4.
- Hand-written index files, establishing links between requirements and source code elements.

Many presentation issues are not specific to software documentation. By using a standard format such as HTML, the generated documentation can benefit from various future developments of the Web, such as search engines, page clustering based on lexical affinity, link generation from textual documentation files, the use of XML to establish a better separation content from presentation, etc.

3.5 Business Case

We have used all the techniques and ideas discussed in this paper in a commercial project aiming at redocumenting a Cobol legacy system. In this section, we describe our findings.

3.5.1 Background

PensionFund is a system for keeping track of pension rights of a specific group of people in the Netherlands. It consists of approximately 500 Cobol programs, 500 copybooks, and 150 DEC DCL batch jobs, totaling over 600,000 lines of code. The main tasks of the system are processing pension contributions and pension claims.

Several years after the initial delivery, the organization responsible for *PensionFund* decided to outsource all maintenance activities to a division of Dutch software house ROCCADE, specializing in software management and maintenance. In order to make a realistic estimate of the anticipated maintenance costs involved before accepting maintenance commitments, ROCCADE performed a *system scan* in which a number of key factors affecting maintainability are estimated.

One of the outcomes of the scan was that the documentation for *PensionFund* was not adequate. In fact, documentation was not kept up to date: for example, although in 1998 a number of major *PensionFund* modifications were implemented, the documentation was never updated accordingly. Very little documentation maintenance had been performed, although the need for documentation grew as more and more programmers who had participated in the original design of *PensionFund* moved to other projects.

The lack of proper documentation resulted in:

- A growing backlog of major and urgent modification requests, which by early 1999 had risen to 12.
- Difficulty in carrying out adequate year 2000 tests, since the documentation did not help to identify the sources of errors encountered during testing.
- Difficulty in mapping modification requests, phrased in terms of desired *functionality* modifications, onto changes to be made in actual programs.

- Difficulty in splitting the large number of daily batch jobs into clusters that could be run independently and in parallel, which was becoming necessary as the increasing number and size of the batch jobs caused the required daily compute time to grow towards the upper limit of 24 hours.

3.5.2 Documentation Wishes

To remedy these *PensionFund* problems, a redocumentation project was planned. The plan was to compose a number of MS-Word documents, one per program, containing:

- A short description
- Calls made (from other Cobol programs or batch jobs) to this program, and calls made from this program;
- Database entities as well as flat files read and written;
- Dataflow diagram;
- Description of functionality in pseudo-code.

Apart from the per program documentation, per batch file one dataflow chart was planned for. Management was willing to make a significant investment to realize this documentation.

Initially, the idea was to write this documentation by hand. This has the advantage that documentation writers can take advantage of their domain or system knowledge in order to provide the most meaningful documentation. Unfortunately, hand-written documentation is very costly and error prone. Because it is not a job many people like to do, it is difficult to find skilled documentation writers.

Therefore, it was decided to try to *generate* the documentation automatically. This has the advantages that it is cheap (the tools do the job), accurate, complete, and repeatable. If necessary, it was argued, it could be extended with manually derived additional information.

3.5.3 Derived Documentation

The contents requirements of the *PensionFund* documentation corresponds to the wishes discussed the previous section. The specific information derived per program is shown in Figure 3.5. Arriving at this list and determining the most desirable way of presentation was an interactive process, in which a group of five *PensionFund* maintenance programmers was involved.

The fact extraction phase mainly involved finding the structure of PERFORM, CALL, and database access statements, and was implemented using island parsing. For those extraction steps for which a line by line scan was sufficient (for example,

Header	Content
Summary	Name, lines of code, two-line description
Activation	Batch jobs or Cobol programs called by
Parameters	List of formal parameters
Data	Databases and flat files read or written
Screens	List of screens sent or received
Calls	Modules and utilities called
Overview	Clickable conditional perform graph
Sections	Clickable outline for each section

Figure 3.5: Contents of the HTML document derived for each *PensionFund* program.

Cobol comment extraction), or for the ones which required the original layout and indentation (summarizing sections) lexical analysis was implemented using Perl.

The result of the fact extraction was a set of relations, which were combined into the required relations per program using Unix utilities such as join and AWK. The final production of HTML code from the resulting relation files was written using Perl.

All the documentation per program could be generated automatically. Even the the two-line description per program could be generated, as this was an easily recognizable part of the prologue comment. Had this not been the case, this would have required a manual step. As top level indices we generated alphabetic lists, lists per subsystem, and clickable call graphs. Moreover, we composed one index manually, grouping the programs based on their functionality.

As a separate top level view, we used the data dependency visualization we derived from the batch files. For each DCL file, we used the techniques described in Section 3.2 to find all Cobol programs that are executed. We then analyzed these Cobol programs to find the data files they read and write to. Using static analysis it is impossible to find all the data file names, because, in this system, some file names were obtained dynamically. This occurs especially in error conditions, where the name of the file to write the error data to is somehow related to the kind of error. The files we could not find names for are only a small fraction of all data files. In order to visualize these unnamed files at a later stage, we introduced special filenames for these files. In Figure 3.4 these unresolved filenames can be seen on the left side, and are clearly marked: “unresolved”.

The list of data files was then matched against the DCL files again, to see whether the data was manipulated there. In the *PensionFund* system, we looked at the `sort` statement, which takes one file and a number of sort parameters, and writes to a different file. They are visualized as diamonds in the figure.

An example browsing session trough the generated documentation is shown in Figure 3.6. A typical session would be a maintenance programmer trying to

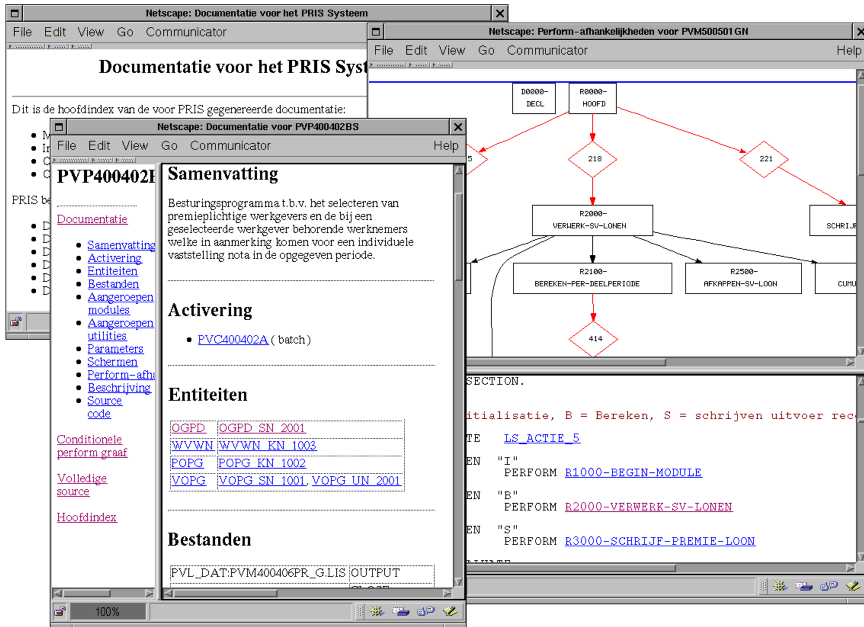


Figure 3.6: Browsing through the documentation generated for *PensionFund*.

find out why a particular batch job did not work as expected. He starts browsing the visualization of the data dependencies in the batch job, follows the links to a specific program, reads the purpose of that program, searches the perform graph for relevant sections, ending in the section responsible for the incorrect system behavior.

3.5.4 Evaluation

As we have demonstrated in the previous section, the documentation generator we have built for *PensionFund* exactly fulfills the wishes the *PensionFund* owners had. Furthermore, the documentation that is generated adheres to the four criteria mentioned in first section of this paper. Compared to the initial plan of manually deriving all documentation, significant cost savings were achieved by employing documentation generators, even if the time needed for configuring the documentation generators is taken into account.

A question of interest is whether this approach is applicable to other legacy systems as well. Our approach takes the *good* properties of a system in to account. For *PensionFund*, these are the systematic coding style which meant that certain

properties (such as program descriptions) were automatically derivable from the source. Furthermore, the programs were relatively short, which made them a natural starting point for documentation generation. Another result is that the (conditional) perform graphs are not too big, making them easily comprehensible. Finally, the fact that the sections were relatively short made the section summaries feasible.

Although other systems may not share the desirable properties of *PensionFund*, they usually have some of these and possibly other strong points. Apart from the program description, all other documentation can be generated for any (Cobol) system. Program descriptions could then be added by hand once, such that subsequent generation steps have this information available. It is part of our future work to see how the generation of the documentation as described here is useful in other systems. We may decide for other systems, that certain levels of documentation are of no use, and new ones are more natural.

We believe the techniques described in this paper are flexible enough to enable us to build different types of documentation generators for different types of systems rather easily.

3.6 Concluding Remarks

Related Work Chikofski and Cross define *redocumentation* as the creation of a semantically equivalent representation of a software system within the same level of abstraction. Common tools include pretty printers, diagram generators, and cross-reference listing generators [CC90]. Landis *et al.* discuss various documentation methodologies, such as Nassi Schneiderman charts, flow charts and Jackson diagrams [LHGF88].

Wong *et al.* emphasize *structural* redocumentation, which, as opposed to documentation *in-the-small*, deals with understanding architectural aspects of software. They use Rigi for the extraction, querying, and presentation, using a *graph editor* for manipulating program representations. Several *views* of the legacy system can be browsed using the editor. Our approach also focuses on the structural aspects of documentation. Rather than using a dedicated graph editor, we use standard HTML browsers for viewing the documentation. We determine the required views in advance, via discussion with the team of maintenance programmers.

The software bookshelf [FHK+97] is an IBM initiative building upon the Rigi experience. In the bookshelf metaphor, three roles are distinguished: the *builder* constructs (extraction) tools; the *librarian* populates repository with meaningful information using the building tools or other (manual) ways, and the *patron* is the end user of the bookshelf. For the building phase, the parsing is like our island approach, in that only constructs of interest are recognized. The parsers are written in Emacs macros, without using an explicit grammar. The parsing code directly emits the HTML code.

Several papers report on the use of hypertext for the purpose of documenting software [Bro91, Raj97, dOBvSdPL98]. Of these, [dOBvSdPL98] follows the *literate programming* [Knu84] approach as also used in, for example, Javadoc, enabling the programmer to control the generation of HTML by manually adding dedicated comment tags.

The need for flexible source code extraction tools was also recognized by, for example, LSME [MN96], as discussed in Section 3.2.1. Another approach of interest is TAWK [GAM96], which uses an AWK like language to match abstract syntax trees.

Contributions In this paper, we have described our contributions to the field of documentation generation. Specific to our approach are:

- The systematic integration of manual documentation writing with automated documentation generation in a redocumentation setting.
- The integration of different levels of documentation abstractness, and the smooth transition between the different levels of documentation.
- The island grammar approach to software fact extraction
- A method for building documentation generators for systems in the Cobol domain.
- The automatic visualization of data-dependencies in mainframe batch jobs.
- The application of the contributions listed above in a commercial environment.

We have shown how we can build documentation generators which adhere to at least the first three criteria from the introduction. The fourth criterion (documentation needs to be consistent with the source code) can only be achieved by eliminating the need for manual documentation. By combining automatically derived documentation and manually derived documentation, and by keeping the input for the two well separated, our documentation generators only need little human input to adhere to all four criteria.

Future Work At the time of writing, we are finalizing the *PensionFund* case study. Moreover, we are in the process of initiating other commercial redocumentation projects, which will help us to identify additional documentation needs and new ways of presenting the data extracted from the sources.

On the extraction side, we plan to elaborate the ideas underlying island grammars. In particular, we will take a close look at the best way of expressing the required analysis of the abstract syntax tree.

Chapter 4

Identifying Objects using Cluster and Concept Analysis

The amount of facts retrieved from a legacy system by performing the analyses described in the previous two chapters is enormous. Particularly when the goal of the analysis is to perform a structural change on the system, the retrieved facts need to be filtered, preferably in an automated way. A possible change may be to move from a procedural system to an object-oriented one. This chapter explores two ways of filtering and interpreting the facts as obtained in the previous two chapters in such a way that they become the starting point for an object-oriented redesign of the system.[1]

4.1 Introduction

In 1976, Belady and Lehman formulated their *Laws of Program Evolution Dynamics* [BL76]. First, a software system that is used will undergo continuous modification. Second, the unstructuredness (entropy) of a system increases with time, unless specific work is done to improve the system's structure. One possible way of doing this is to migrate software systems to object technology. Object orientation is advocated as a way to enhance a system's correctness, robustness, extendibility, and reusability, the key factors affecting software quality [Mey97].

The migration of legacy systems to object orientation, however, is no mean task. A first, less involved, step includes merely the identification of candidate objects in a given legacy system. The literature reports several systematic approaches

[1]This chapter was published earlier as: A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *21st International Conference on Software Engineering, ICSE-99*, pages 246–255. ACM, 1999.

to object identification, some of which can be partially automated. (In Section 4.2 we provide a summary). There are several problems, however, with the application of these approaches to actual systems.

1. Legacy systems greatly vary in source language, application domain, database system used, etc. It is not easy to select the identification approach best-suited for the legacy system at hand.
2. It is impossible to select a *single* object identification approach, since legacy systems typically are heterogeneous, using various languages, database systems, transaction monitors, and so on.
3. There is limited experience with actual object identification projects, making it likely that new migration projects will reveal problems not encountered before.

Thus, when embarking upon an object identification project, one will have to select and compose one's own blend of object identification techniques. Moreover, during the project, new problems will have to be solved. This is exactly what happened to us when we tried to construct an object-oriented redesign of Mortgage, a real life legacy Cobol system.

For many business appliations written in Cobol, the data stored and processed represent the core of the system. For that reason, the data records used in Cobol programs are the starting point for many object identification approaches (such as [CDDF99, NK95, FRS94]).

Object identification typically consists of several steps: (1) identify legacy records as candidate classes; (2) identify legacy procedures or programs as candidate methods; (3) determine the best class for each method via some form of cluster analysis [Lak97]. This approach gives good results in as far as the legacy record structure is adequate. In our case study, however, records consisted of up to 40 fields. An inspection of the source code revealed that in the actual use of these records, many of the fields were entirely unrelated. Making this record into a single class would lead to classes with too many unrelated attributes.

In this paper, we report on our experience with the application of some of the techniques proposed for object identification, most notably cluster and concept analysis, to Mortgage. Moreover, we discuss in full detail how the unrelated-record-fields problem – not covered by any of the existing object identification approaches – can be addressed in general. Our approach consists of clustering record fields into coherent groups, based on the actual *usage* of these fields in the procedural code. We not only use traditional *cluster analysis* [KR90, Lak97] for this, but also the recently proposed *concept analysis* [SR97, LS97].

The principal new results of this paper include:

- A proposal for usage-based record structuring for the purpose of object identification;

- Significant practical experience with the use of cluster and concept analysis for object identification;
- A discussion of a number of problems (and solutions) involving the use of cluster and concept analysis in general;
- A comparison of the use of cluster and concept analysis for the purpose of object identification.

4.2 Related Work

A typical approach to finding classes in legacy code 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. [OT93, LW90, CCM96, Sch91]. A unifying framework discussing such *subsystem classification techniques* is provided by Lakhotia [Lak97].

Unfortunately, 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, as was also observed by Cimitile *et al.* [CDDF99].

Other class extraction techniques have been developed specifically with languages like Cobol in mind. They take specific characteristics into account, such as the close connection with databases.

Newcomb and Kotik [NK95] take all level 01 records as a starting point for classes. They then proceed to map similar records to single classes, and find sections that can be associated as methods to these records. Their approach exhibits a high level of automation, and, as a consequence, results in an object-oriented program that stays close to the original Cobol sources.

Fergen *et al.* [FRS94] describe the MOORE tool, which analyses Cobol-85 code, and provides the engineer with a set of *class proposals*. All records are given a *weight*, which indicates the number of references made to that record. No attempt is made at splitting up large records into smaller structures. Proposals for methods consist of Cobol paragraphs which use or modify one of the record fields, again ranked by the weight of the fields in that paragraph. To reduce the total number of classes, every time a new candidate class is found, a numeric *similarity* measure is used to see whether already existing classes can be used to build this new candidate class.

De Lucia *et al.* [DDF$^+$97, CDDF99] describe the ERCOLE paradigm for migrating RPG programs to object-oriented platforms. It consists of several steps, one of which is “abstracting an object-oriented model.” This step is centered

around the persistent data stores. Batch programs, subroutines, or groups of call-related subroutines are candidate methods. Data stores and methods are combined in such a way that certain object-oriented design metrics get optimal values.

Sneed and Nyáry [SN95] present a tool, OBJECT-REDOC, that can be used to derive documentation automatically from legacy sources. The documentation itself is "object-oriented", in that it takes an object-oriented view on the legacy system. Sneed also provides a systematic method, REORG, to transform Cobol programs to object-oriented frames in a largely manual manner [Sne92].

Tan and Ling [TL95] present a domain-specific approach to reengineering data-intensive business programs. They propose the use of an *augmented object model*, which is an extension of the object modeling technique OMT. Their model recovery procedure takes constants, user inputs, retrieved and updated database records, and user outputs as its starting point. However, they make no attempt at splitting up records in smaller structures.

Wiggerts *et al.* [WBF97] describe three different *scenarios* for object identification. Their *function*-driven scenario takes legacy *functionality* (subsystems performing a certain task) as starting point for class extraction. The *data*-driven approach starts by searching for (persistent) data elements, which are likely to describe business entities. The *object*-driven approach, finally, does not start from the legacy system itself, but starts by building an object model of the application domain.

4.3 Field and Program Selection

Legacy systems contain data and functionality that are useful in a given application domain. Unfortunately, the legacy system also contains a significant amount of code of a technical nature, closely tied to the implementation language, operating system, database management system, etc. When migrating legacy systems to object technology, such technical code is of significantly less interest than the domain-related code, for example because the object-oriented platform is likely provide facilities for dealing with the technicalities in an entirely different manner.

Therefore, a first important step in any object identification activity must be to filter the large number of programs, procedures, records, variables, databases, etc., present in the legacy system.

One of the main selection criteria will be whether a legacy element is domain-related or implementation-specific. This is a criterion that is not easy to derive from structural code properties alone. Consequently, this step may require human interaction, in order to take advantage of domain knowledge, application knowledge, systematic naming conventions, meaningful identifiers, comments, etc.

In many cases, though, structural code properties will be able to provide a meaningful selection of legacy data elements and procedures. Selection criteria to be used may include the use of metrics, such as requiring a McCabe complexity

metric between a given minimum and maximum as discussed in [CB91]. Others may include the classification of variables, for example according to the *type* they belong to [DM98] or according to whether a variable is used to represent data obtained from persistent data stores [CDDF99].

Our own experience with selecting domain-related data and functionality is described in [DK98]. In this paper, we will use two guidelines, one for selecting data elements and one for selecting programs. These helped to find objects in our Mortgage case study, and we expect them to work well for other systems too.

First, in Cobol systems the *persistent data stores* (following the terminology of [CDDF99]) contain the essential business data. Hence, the selection to be made on all records in a Cobol program is to restrict them to those written to or read from file. This selection can be further improved by taking the CRUD (Create, Read, Update, Delete) matrix for the system into account. Threshold values can be given to select those databases that are read, updated, deleted, or written by a minimal or maximal number of different programs.

Second, it is important to select the programs or procedures containing domain-related functionality. An analysis of the *program call graph* can help to identify such programs. First, programs with a high *fan-out*, i.e., programs calling many different programs, are likely to be control modules, starting up a sequence of activities. Second, programs with a high *fan-in*, being called by many different programs, are likely to contain functionality of a technical nature, such as error handling or logging. Eliminating these two categories reduces the number of programs to deal with. In many cases, the remaining programs are those containing a limited, well described functionality.

4.4 Cluster analysis

The goal of this paper is to identify groups of record fields that are related functionally. Cluster analysis is a technique for finding related items in a data-set. We apply cluster analysis to the usage of record fields throughout a Cobol system, based on the hypothesis that record fields that are related in the implementation (are used in the same program) are also related in the application domain.

In this section we will first give a general overview of the cluster analysis techniques we used. Then we give an overview of the cluster analysis experiments we performed. We end the section with an assessment of our cluster experiments and the usage of cluster analysis for object identification in general.

4.4.1 Overview

We will explain the clustering techniques we have used by going through the clustering of an imaginary Cobol system. This system consists of four programs, and uses one record containing nine fields. The names of these fields are put into the

	P_1	P_2	P_3	P_4
NAME	1	0	0	0
TITLE	1	0	0	0
INITIAL	1	0	0	0
PREFIX	1	0	0	0
NUMBER	0	0	0	1
NUMBER-EXT	0	0	0	1
ZIPCD	0	0	0	1
STREET	0	0	1	1
CITY	0	1	0	1

Table 4.1: The usage matrix that is used as input for the cluster analysis

	N	T	I	P	N	NE	Z	S	C
N	0								
T	0	0							
I	0	0	0						
P	0	0	0	0					
N	$\sqrt{2}$	$\sqrt{2}$	$\sqrt{2}$	$\sqrt{2}$	0				
NE	$\sqrt{2}$	$\sqrt{2}$	$\sqrt{2}$	$\sqrt{2}$	0	0			
Z	$\sqrt{2}$	$\sqrt{2}$	$\sqrt{2}$	$\sqrt{2}$	0	0	0		
S	$\sqrt{3}$	$\sqrt{3}$	$\sqrt{3}$	$\sqrt{3}$	1	1	1	0	
C	$\sqrt{3}$	$\sqrt{3}$	$\sqrt{3}$	$\sqrt{3}$	1	1	1	$\sqrt{2}$	0

Table 4.2: The distance matrix from Table 4.1

set of cluster items. For each of the variables in the set, we determine whether or not it is used in a particular program. The result of this operation is the matrix of Table 4.1. Each entry in the matrix shows whether a variable is used in a program (1) or not (0).

Distance Measures

Because we want to perform cluster analysis on these data, we need to calculate a distance between the variables. If we see the rows of the matrix as vectors, then each variable occupies a position in a four dimensional space. We can now calculate the Euclidean distance between any two variables.

If we put the distances between any two variables in a matrix, we get a so-called *distance* (or *dissimilarity*) matrix. Such a distance matrix can be used as input to a clustering algorithm. The distance matrix for Table 4.1 is shown in Table 4.2. Note that any relation the variables had with the programs $P_1, \ldots, P_4$ has become invisible in this matrix.

An overview of different distance calculations for clustering can be found in [Wig97].

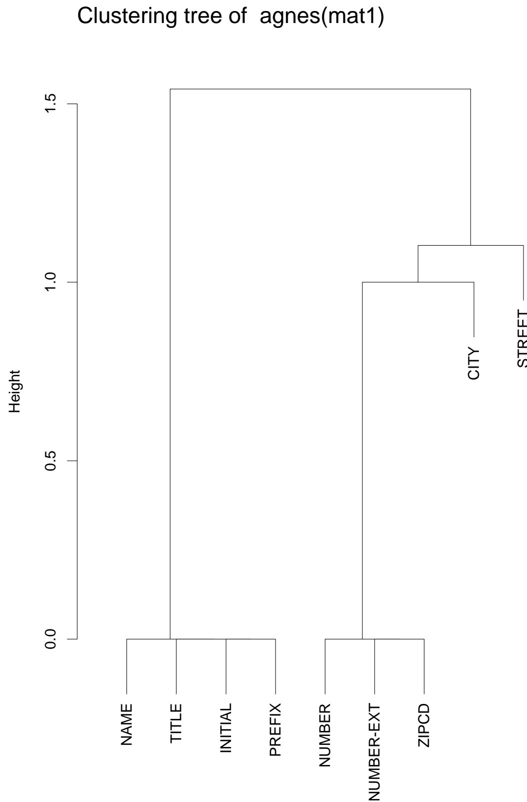


Figure 4.1: The resulting clustering from Table 4.2

Agglomerative Clustering

We use an agglomerative hierarchical clustering algorithm (AGNES, from [KR90]). This algorithm starts by putting each element in its own cluster, and then proceeds by creating new clusters that contain two (or more) clusters that are closest to one another. Finally, only one cluster remains, and the algorithm terminates. All intermediate clusterings can be seen as branches on a tree, in a dendrogram. Figure 4.1 shows the dendrogram that results from clustering the data in Table 4.1.

The actual clusters found by this algorithm are identified by drawing a horizontal line through the dendrogram, at a user defined height. In our example here, that line would typically be drawn at height 1.3, thus producing two clusters. The first cluster contains NAME, TITLE, INITIAL, and PREFIX. The second contains NUMBER, NUMBER-EXT, ZIPCD, CITY, and STREET. These clusters are

likely candidates to become classes, containing the named fields as their member variables.

Explanation of Dendrogram

In Figure 4.1, the axis labelled "height" shows the relative distance the clusters have from each other. The variables NAME, TITLE, INITIAL, and PREFIX have a relative distance of zero (see Table 4.2), and thus form one cluster. We will call this cluster c_1. NUMBER, NUMBER-EXT and ZIPCD also have distance zero. We will call this cluster c_2. No other clusters with members that have distance 0 exist.

The clustering algorithm uses "average linkage" to measure the distance between two clusters. This means that the distance between two clusters is the average of the distances between all nodes of the one cluster, and all nodes of the other cluster. (See [Wig97] for a discussion of this and other linkage methods.) Using this linkage method, the closest element to cluster c_2 is either CITY, or STREET. They both have a distance of 1 to c_2. The clustering algorithm nondeterministically chooses one of CITY or STREET. In our case it chooses CITY. c_2 and CITY together form cluster c_3.

The element closest to c_3 is STREET. It has a distance of $\sqrt{2}$ to CITY, and a distance of 1 to all elements of c_2. So, on average, the distance between STREET and c_3 is $\frac{3+\sqrt{2}}{4} \approx 1.1$. This new cluster we will call c_4.

Now, only two clusters remain: c_1 and c_4. The distance between these two clusters is $\frac{4\times(3\times\sqrt{2}+2\times\sqrt{3})}{4\times5} \approx 1.54$.

4.4.2 Experimental Testbed

The input data for our cluster experiments was generated from Cobol source code, using lexical analysis tools. The data from these tools was fed into a relational database. We wrote a tool to retrieve the data from the database, and to format it for our cluster tools. The source code was from Mortgage, a 100.000 LOC Cobol system from the banking area. It uses VSAM files for storing data. The toolset used for the generation of data, and the architecture of those tools is described in more detail in [DK98]. The Mortgage system is described in more detail in [DK98, WBF97].

For our cluster experiments we used S-PLUS, a statistical analysis package from MathSoft. The cluster algorithms described in [KR90] are implemented as part of S-PLUS.[2]

All experiments were performed on a SGI O2 workstation.

[2] The implementation is available from http://win-www.uia.ac.be/u/statis/programs/clusplus_readme.html

4.4.3 Experiments

As already described in Section 4.3, we selected a number of variables and programs from Mortgage to perform our cluster experiments on. In this section we will describe our main experiment, which was executed in three steps. The results of the clustering experiments are shown in Figure 4.2. As stated before, we are looking for clusters of functionally related record fields. In order to validate the use of cluster analysis for this purpose, we need to validate the clusters found. We have asked engineers with an in-depth knowledge of the system to validate the clusters for us.

The (variable) names mentioned in the dendrograms of Figure 4.2 are in Dutch. We will translate the names as we explain the three dendrograms of that figure.

1. We restricted the variables to be clustered to only those occurring in the three main records of Mortgage. This led to the dendrogram of Figure 4.2(a). There are a number of groups that seem meaningful, such as STRAAT, POSTKD, WOONPL and HUISNR (street, zip code, city and street number), or the cluster containing STREEK, LANDKD, and GEMKD (region, country code, county code). In short, this dendrogram does illustrate which variables are used together frequently, and which could therefore be grouped together.

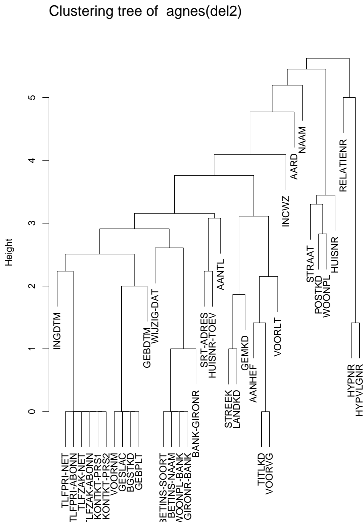
   Unfortunately, there are also a number of fields with a position that is not so easy to explain. These are in particular the ones with a "higher" position, such as INCWZ, AARD, NAAM or AANTL (payment, kind, name, and occurrence). Also, the grouping of contact persons (KONTKT-PERS) with telephone numbers (everything starting with TLF) is unclear.

2. The next step is to restrict the number of programs involved. Figure 4.2(b) shows the clustering results when only programs from the group of "relevant programs" (as described in Section 4.3) were taken into account.

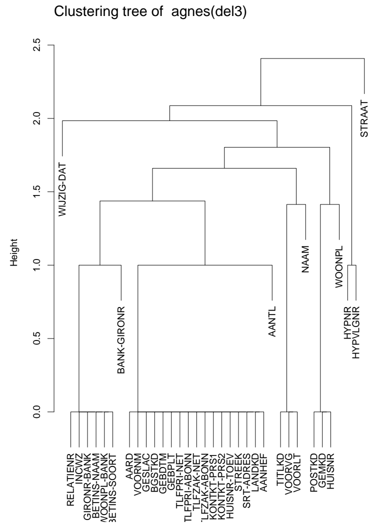
   The result is promising, and has a simpler tree structure. However, there is an unattractively large group of fields that are lumped together, which does not look very meaningful. The reason for this is that there are two programs in the group of relevant programs which use *all* variables. Therefore their discriminating capabilities in the clustering are very low.

3. We repeated the same experiment, but now without the programs which use all variables. The result is the dendrogram of Figure 4.2(c). This is a very satisfying cluster result.

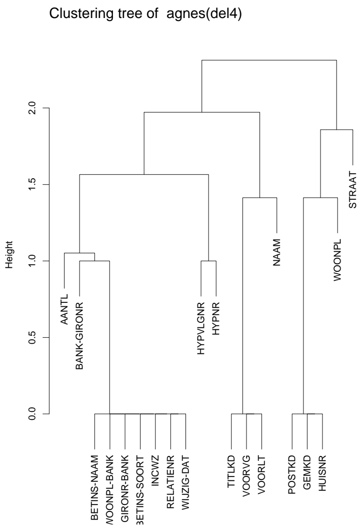
Note that the last dendrogram contains significantly less field names than the first. This makes it easier to comprehend the clusters, but also means that we have to inspect all removed variables manually for inclusion in one (or none) of the generated clusters.



(a) Clustering using variables from three main records of Mortgage.



(b) As Figure 4.2(a), but restricted to the relevant programs (with low fan-in and fan-out).



(c) As Figure 4.2(b), but without the programs which use all variables from the three records.

Figure 4.2: Sequence of more and more refined clustering

4.4.4 Assessment

We have identified two fundamental problems when using cluster analysis this way:

1. When clustering, all items end up in exactly one cluster. However, sometimes one item (one variable) is equally likely to end up in more than one cluster. For instance, two records may use the same key field. If all other fields of the records are disjoint, and are used disjointly, we end up with three clusters: one containing the fields of the first record, without the key field, one with the fields of the second record without the key field, and one with only the key field. It is unclear whether this is the most desirable result. Perhaps we would rather have two clusters, corresponding exactly to the two records. Unfortunately, as items can only occur in exactly one cluster, this is not possible using cluster analysis.

2. As we have demonstrated in our example, when we are building the cluster hierarchy, sometimes there is more than one closest cluster. Assume we have a cluster A, which has the same distance to both clusters B and C (e.g., in our example, both CITY and STREET had a distance of 1 to cluster c_2). The algorithm at that point chooses one, arbitrarily. Say the algorithm chooses cluster B, thus forming cluster A'. Now cluster A' has a particular distance to cluster D which may be very different from the distance it had had if the algorithm had chosen C and A to form A'. If this happens near the leaves of the dendrogram, the results of an arbitrary decision can be drastic.

We have partly solved these problems as follows:

1. The fields most likely to end up in more than one cluster are fields that are used together with a lot of other fields. Or, in short, the fields that are used most often. The system we experimented with demonstrated this property. The above mentioned key field is obviously used quite often, because it uniquely identifies a record. We have overcome the restrictions of the cluster algorithm by removing these variables from our cluster set before starting the cluster analysis. This proved to be a satisfactory method.

   Automatic variable selection procedures in cluster algorithms have been proposed in the literature [FGK88]. It is a topic of future research to incorporate these procedures in our clustering experiments.

2. We have tried to resolve the second problem by changing the distance metrics and the linkage methods between clusters. We experimented with all metrics and methods described in [Wig97]. However, although changing these parameters indeed resulted in different clusters, it did not necessarily result in *better* clusters. The problem here is that it often is unclear which of the choices is the better choice, and indeed the choice *is* arbitrary. What

sometimes is clear is that a particular sequence of choices is to be preferred above another sequence. We have not tried to incorporate this notion into our cluster algorithm. This would probably require some type of backtracking mechanism, or a multiple pass algorithm, and is a topic of further research.

In conclusion we can say that cluster analysis can be used for restructuring records, given a number of restrictions. First, the number of fields to be clustered cannot be too large. Second, the fields to be clustered should be occurring selectively in the system (i.e., they should not be omnipresent fields, for these generate noise). Finally, there needs to be some external way to validate the clustering.

4.5 Concept Analysis

Recently, the use of mathematical *concept analysis* has been proposed as a technique for analyzing the modular structure of legacy software [LS97, SR97, Sne98, ST98]. As with cluster analysis, we use concept analysis to find groups of record fields that are related in the application domain.

Concept analysis and cluster analysis both start with a table indicating the *features* of a given set of *items*. Cluster analysis then partitions the set of items in a series of disjoint clusters, by means of a numeric distance measure between items indicating how many features they share.

Concept analysis differs in two respects. First, it does not group items, but rather builds up so-called *concepts* which are maximal sets of items sharing certain features. Second, it does not try to find a single optimal grouping based on numeric distances. Instead it constructs *all* possible concepts, via a concise lattice representation.

As we will see in the next paragraphs, these two differences can help to solve the two problems with clustering discussed in the previous section. In this section, we will first explain the basics of concept analysis. Then we will discuss its application to our Mortgage case study in full detail, followed by a comparison with the clustering results.

4.5.1 Basic Notions

We start with a set $\mathcal{M}$ of *items*, a set $\mathcal{F}$ of *features*,[3] and a *feature table* (relation) $T \subseteq \mathcal{M} \times \mathcal{F}$ indicating the features possessed by each item. If we reuse the data of Table 4.1 as running example, the items are the field names, the features are usage in a given program, and the feature table corresponds to the matrix entries having value 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.

name	extent	intent
top	{NAME, TITLE, INITIAL, PREFIX, NUMBER, NUMBER-EXT, ZIPCD, STREET, CITY}	$\emptyset$
c1	{NAME, TITLE, INITIAL, PREFIX}	$\{P_1\}$
c2	{NUMBER, NUMBER-EXT, ZIPCD, STREET, CITY }	$\{P_4\}$
c3	{STREET}	$\{P_3, P_4\}$
c4	{CITY}	$\{P_2, P_4\}$
bot	$\emptyset$	$\{P_1, P_2, P_3, P_4\}$

Table 4.3: All concepts in the example of Table 4.1

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,

$$(\{\texttt{NAME}, \texttt{TITLE}, \texttt{INITIAL}, \texttt{PREFIX}\}, \{P_1\})$$

is the concept of those items having feature P_1, i.e., the fields used in program P_1. All concepts that can be identified from Table 4.1 are summarized in Table 4.3. The items of a concept are called its *extent*, and the features its *intent*.

The concepts of a given table form a partial order via:

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

As an example, for the concepts listed in Table 4.3, we see that bot $\leq c3 \leq c2 \leq$ top.

The subconcept relationship 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 subconcept relationships between them. For our example, the lattice is shown in Figure 4.3. In such visualizations, the nodes only show the "new" items and features

per concept. More formally, a node is labelled with an item i if that node is the *smallest* concept with i in its extent, and it is labelled with a feature f if it is the *largest* concept with f in its intent.

The concept lattice can be efficiently computed from the feature table; we refer to [LS97, SR97, Sne98, ST98] for more details.

4.5.2 Experimental Testbed

To perform our concept analysis experiments, we reused the Cobol analysis architecture explained in Section 4.4. The analysis results could be easily fed into the `concept` tool developed by C. Lindig from the University of Braunschweig.[4] We particularly used the option of this tool to generate input for the graph drawing package `graphplace` in order to visualize concept lattices.

4.5.3 Experiments

We have performed several experiments with the use of concept analysis in our Mortgage case study. As with clustering, the choice of items and features is a crucial step in concept analysis. The most interesting results were obtained by using exactly the same selection criteria as discussed in Section 4.3: the items are the fields of the relevant data records, and the programs are those with a low fan-in and fan-out. The results of this are shown in Figure 4.4, which shows the concept lattice for the same data as those of the dendrogram of Figure 4.2(b). In order to validate the use of concept analysis, we need to validate the results of the concept analysis. Again, these results were validated by systems experts.

In Figure 4.4 each node represents a concept. The items (field names) are names written below the concept, the features (programs using the fields) are written as numbers above the concept. The lattice provides insight in the organization of the Mortgage legacy system, and gives suggestions for grouping programs and fields into classes.

The row just above the bottom element consists of five separate concepts, each containing a single field. As an example, the leftmost concept deals with *mortgage numbers* stored in the field `MORTGNR`. With it is associated program `19C`, which according to the comment lines at the beginning of this program performs certain checks on the validity of mortgage numbers. This program *only* uses the field `MORTGNR`, and no other ones.

As another example, the concept `STREET` (at the bottom right) has three different programs directly associated with it. Of these, `40` and `40C` compute a certain standardized extract from a street, while program `38` takes care of standardizing street names.

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

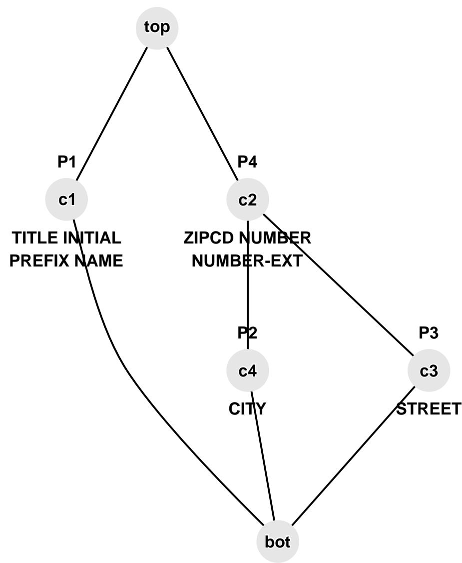


Figure 4.3: Lattice for the concepts of Table 4.3

If we move up in the lattice, the concepts become larger, i.e., contain more items. The leftmost concept at the second row contains *three* different fields: the *mortgage sequence number* MORTSEQNR written directly at the node, as well as the two fields from the lower concepts connected to it, MORTGNR and RELNR. Program 09 uses all three fields to search for full mortgage and relation records.

Another concept of interest is the last one of the second row. It represents the combination of the fields ZIPCD (zip code), HOUSE (house number), and CITYCD (city code), together with STREET and CITY. This combination of five is a separate concept, because it actually occurs in four different programs (89C, 89, 31C, 31). However, there are no programs that *only* use these variables, and hence this concept has no program associated with it.

The largest concepts reside in the top of the lattice, as these collect all fields of the connected concepts lower in the lattice. For example, the concept with programs 31 and 31C consists of a range of fields directly attached with it (FIRSTNM, ...), as well as of all those in the three downward links below it. It corresponds to almost all fields of one particularly large record, holding the data of so-called *relations* (people and companies that play a role when a mortgage is set up). These fields are then processed by programs 31 and 31C. Only one field, MOD-DAT (modification date), is part of that *relations* record but not used in 31 and 31C.

Another large concept of interest is the one with programs 89C and 89. The fields in this concept all come from the Dutch *zip code book*, holding data for all Dutch addresses and their zip codes. As can be seen from Figure 4.4, the fields of this concept are largely disjoint with those of the *relations* concept (with programs 31 and 31C). However, these two concepts also share five fields, namely those of the ZIPCD concept. These fields can be used (in various combinations) as the lookup key for the zip code book.

4.6 Clustering and Concepts Compared

The application of both concept and cluster analysis to Mortgage highlights the differences listed below. From them, we conclude that concept analysis is more suitable for object identification than cluster analysis.

Multiple partitionings Having a hierarchy of clusterings rather than a single partitioning result, is attractive as it allows one to select the most suitable clustering.

At first sight, a dendrogram seems to provide exactly such a hierarchy. Unfortunately, as we have seen in Section 4.4, the actual clusters built in the final iterations of an agglomerative analysis strongly depend on clustering decisions made earlier in the analysis. It is certainly not the case that a dendrogram shows all possible clusterings.

Concept analysis, by contrast, shows *all* possible groupings that are meaningful given the feature table. In our experience, this is more helpful for the engineer

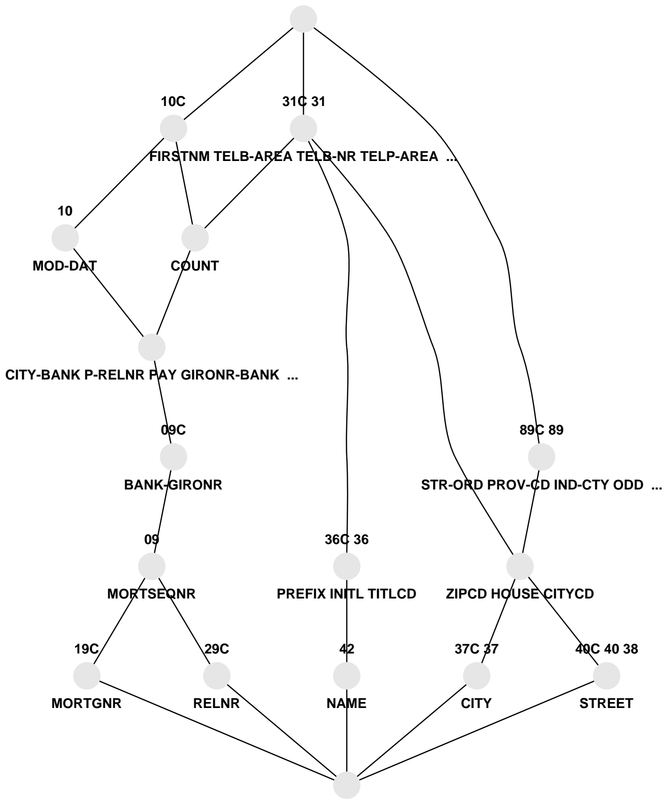


Figure 4.4: Concept lattice showing how persistent fields are used in programs in the Mortgage case study.

trying to understand a legacy system.

Items in multiple groups With cluster analysis, the result is a *partitioning*, i.e., each item is placed in exactly one cluster. In some cases, however, it is important to group items in multiple clusters. For our type of experiments, for example, database *key* fields may occur in multiple records: once as *primary* key, and in potentially multiple other records as *foreign* key.

With concept analysis, unlike clustering, this is possible. In our experiments, key fields occur as separate concepts, with separate upward links to those concepts using them as either primary or foreign key. In Figure 4.4, the zip code concept is an example of such a key concept.

Moreover, if concept analysis is used, it still is possible to obtain a partitioning, following an algorithm discussed in [SR97].

Features and Clusters For class extraction purposes, it is important to understand which features were responsible for the creation of certain clusters. With cluster analysis, this is virtually impossible, as the analysis is solely based on the distance matrix (see Table 4.2), in which no feature appears.

The absence of features also makes dendrograms more difficult to interpret than concept lattices. For example, in Figure 4.4 it is clear that program `10` is responsible for the special status of `MOD-DAT`, but in Figure 4.2(b) it is not at all obvious why `STRAAT` (street) appears at the top of the dendrogram.

Selection of input data The appropriate selection of input data stronly affects the results of both cluster and concept analysis. Cluster analysis turns out to be very sensitive to items that possess *all* features. As a result, we have derived two extra selection steps for cluster analysis: Remove programs that use all fields from the input data, and remove record fields that are used in all programs from the input data.

Concept anlaysis is also sensitive to the selection of input data, but less so: therefore, we were able to derive the concept lattice of Figure 4.4 from the data used for the dendrogram in Figure 4.2(b), rather than from the more restricted dataset used in Figure 4.2(c).

4.7 Object Identification

The final object identification step is to use the cluster and concept analysis results to build object-oriented classes. Although some degree of automation may be possible for this step, meaningful classes can be expected only if it is done interactively by a software engineer equipped with experience in object-oriented design as well as some knowledge of the application domain and the legacy system. The role of cluster and concept analysis, then, is to reduce the overwhelming number of 100,000 lines of code to a number of high-level design decisions.

When using cluster analysis, the engineer will have to decide at which height the clusters are to be chosen in a given dendrogram. This determines how many

clusters exist, how large they are, and what is contained in them. Each cluster represents a candidate class, having the fields in the cluster as its attributes. The cluster hierarchy present in a dendrogram also gives pointers for relations between the classes. If a large cluster c is obtained by merging clusters $c_1, ..., c_n$, the corresponding class c will typically be composed from the classes for $c_1, ..., c_n$ via aggregation (c will have n attributes for fields of type $c_1, ..., c_n$). In some cases, inheritance or association may be more appropriate, but the dendrogram itself provides no clues for making this decision. Cluster analysis provides no information on which methods to attach to each of the classes identified.

When using concept analysis, the engineer can take advantage of the presence of the programs (as features) in the lattice. An important use of the lattice is as a starting point for acquiring understanding of the legacy system. As illustrated by the discussion of the Mortgage experiment in Section 4.5, the engineer can browse through the lattice, and use it to select programs at which to look in more detail.

Each concept is a candidate class. The smallest concept introducing a field corresponds to the class having that field as attribute. The largest concept with a given program as feature corresponds to the class with that program attached as method to it. This is reflected in the way the concepts are labeled with items and features in the concept lattice. Classes close to the bottom are the smallest classes (containing few attributes).

The subconcept relationship corresponds to class relations. Typically, a class for a concept c is composed via aggregation from the classes of the subconcepts of c. Alternatively, if a concept c has a subconcept c', c may be composed from c' via inheritance. As an example, the concept with field `NAME` (and program `42`) in Figure 4.4 deals with names of persons. A natural refinement of this class is the concept above it, which extends a person's name with his prefixes, initials, and title code. Independent "columns" in the concept lattice correspond to separate class hierarchies.

A final question of interest is whether the classes found this way are "good" classes. For Mortgage, an independent, manually developed, object-oriented redesign exists (which is partly described by [WBF97]). A good semi-automatic approach should get as close as possible to this redesign. The lattice of Figure 4.4 does not yield the complete redesign, but the concepts in the lattice constitute the core classes of the independent redesign. One difference is that certain large "container" classes are not present in the lattice. A second difference is that in the redesign domain knowledge was used to further refine certain classes (for example, a separate "bank address" class was included). However, this separation was not explicitly present in the legacy system. For that reason, it was not included in the concept lattice, as this only serves to show how fields are actually being used in the legacy system.

4.8 Concluding Remarks

In this paper we have studied the object identification step of combining legacy data structures with legacy functionality. We have used both cluster and concept analysis for this step. Concept analysis solves a number of problems encountered when using cluster analysis.

Of utmost importance with both concept and cluster analysis is the appropriate selection of the items and features used as a starting point, in order to separate the technical, platform-specific legacy code from the more relevant domain-related code. The selection criteria we used are discussed in Section 4.3.

When searching for objects in data-intensive systems (which is the typically the case with Cobol systems), records are a natural starting point. We have argued that it is first necessary to decompose the records into smaller ones, and we have proposed a method of doing so by grouping record fields based on their actual usage in legacy programs.

We have used this grouping problem to contrast cluster analysis with concept analysis. We identified the following problems with cluster analysis (see Section 4.6): (1) cluster analysis only constructs *partitionings*, while it is often necessary to place items in multiple groups; (2) a dendrogram only shows a subset (a hierarchy) of the possible partionings, potentially leaving out useful ones; (3) a dendrogram is difficult to explain, as it is based on numeric distances rather than actual features; (4) cluster analysis tends to be sensitive to items possessing *all* features.

These limitations are inherent to clustering, and independent of the distance measures chosen, or the sort of items used to cluster on.

These problems are dealt with in a better way by concept analysis, making it therefore more suitable for the purpose of object identification. Concept analysis finds all possible combinations, and is not just restricted to partitionings. Moreover, the features are explicitly available, making it easier to understand why the given concepts emerge.

4.8.1 Acknowledgments

We thank the members of the *Object and Component Discovery* Resolver task group: Hans Bosma, Erwin Fielt, Jan-Willem Hubbers, and Theo Wiggerts. Finally, we thank Andrea De Lucia, Jan Heering, Paul Klint, Christian Lindig, and the anonymous referees for commenting on earlier versions of this document.

Chapter 5

Types and Concept Analysis for Legacy Systems

In the previous chapter a starting point for an object-oriented re-design was obtained by looking at the different data elements used in a system. In this chapter, this notion is refined. Not the individual data elements are used to perform concept analysis on, but rather the inferred types of the data elements. Type inference tries to identify a group of data elements which are of the same type, i.e., represent the same notion in a system. Using type inference, a large group of data elements can be reduced to a smaller group of data types.

As this chapter has been published as a separate article earlier, we ask the reader to bear with us as some paragraphs in this chapter overlap with paragraphs in the previous chapter.[1]

5.1 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,

[1] This chapter was published earlier as: T. Kuipers and L. Moonen. Types and concept analysis for legacy systems. In *Proceedings of the International Workshop on Programming Comprehension (IWPC 2000)*. IEEE Computer Society, June 2000.

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) avaliable 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 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 [DM98]. This results in types for variables, program parameters, database records, literal values, and so on, which are used during analysis.

In this paper, 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, DM99].

5.2 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 5.1. 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.[2] 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 [DM98]. 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.

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

[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.

```
1     DATA DIVISION.
2     / variables containing business data.
3      01 PERSON.
4       03 INITIALS     PIC  X(05).
5       03 NAME         PIC  X(27).
6       03 STREET       PIC  X(18).
7       ...
8     / variables containing char array of length 40,
9     / as well as several counters.
10     01 TAB000.
11      03 A00-NAME-PART.
12       05 A00-POS     PIC  X(01)  OCCURS 40.
13      03 A00-MAX      PIC S9(03)  COMP-3 VALUE 40.
14      03 A00-FILLED   PIC S9(03)  COMP-3 VALUE 0.
15      ...
16    / other counters declared elsewhere.
17     01 N000.
18      03 N100         PIC S9(03)  COMP-3 VALUE 0.
19      03 N200         PIC S9(03)  COMP-3 VALUE 0.
20
21    PROCEDURE DIVISION.
22    / procedure dealing with initials.
23     R210-INITIAL SECTION.
24      MOVE INITIALS TO A00-NAME-PART.
25      PERFORM R300-COMPOSE-NAME.
26
27    / procedure dealing with last names.
28     R230-NAME SECTION.
29      MOVE NAME TO A00-NAME-PART.
30      PERFORM R300-COMPOSE-NAME.
31
32    / procedure for computing a result based
33    / on the value of the A00-NAME-PART.
34    / Uses A00-FILLED, A00-MAX, and N100
35    / for array indexing.
36     R300-COMPOSE-NAME SECTION.
37      ...
38      PERFORM UNTIL N100 > A00-MAX
39       ...
40       IF A00-FILLED = N100
41        ...
```

Figure 5.1: Excerpt from one of the Cobol programs analyzed (with some explanatory comments added).

level. In the remainder we will use T_A to denote the primitive type of variable A.

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.

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.

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.

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.[3]

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*.

[3] 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.

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 [DM98], 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 types can be propagated through record fields even if the records do not have the same structure.

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 [DM99], we have described a range of experimental data showing the effectiveness of subtyping for dealing with pollution.

Example Figure 5.1 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.

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

Table 5.1: The list of items and their features

	items	features
0	zipcd number-ext number street city prefix initial title name	
1	zipcd number-ext number street city	p4
2	street	p4 p3
3	city	p4 p2
4	prefix initial title name	p1
5		p4 p3 p2 p1

Table 5.2: All concepts identified for Table 5.1.

5.3 Concept Analysis

Concept analysis[4] 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 [Sne98]. 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.

[4] This section overlaps with Section 4.5, because both chapters were published as separate articles.

5.3.1 Basic Notions

We start with a set $\mathcal{M}$ of *items*, a set $\mathcal{F}$ of *features*,[5] 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 Table 5.1 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, $\sigma(\{\mathtt{ZIPCD}, \mathtt{STREET}\}) = \{P_4\}$.

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, $\tau(\{P_3, P_4\}) = \{\mathtt{STREET}\}$.

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,

$$(\{\mathtt{PREFIX}, \mathtt{INITIAL}, \mathtt{TITLE}, \mathtt{NAME}\}, \{P_1\})$$

is the concept of those items having feature P_1, i.e., the fields used in program P_1. All concepts that can be identified from Table 5.1 are summarized in Table 5.2. The items of a concept are called its *extent*, and the features its *intent*.

The concepts of a given table are partially ordered via:

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

As an example, for the concepts shown in Table 5.2, 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

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

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 5.2.

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*

[5] 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.

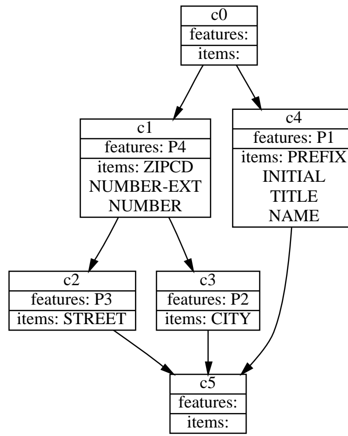


Figure 5.2: Lattice for the concepts of Table 5.2.

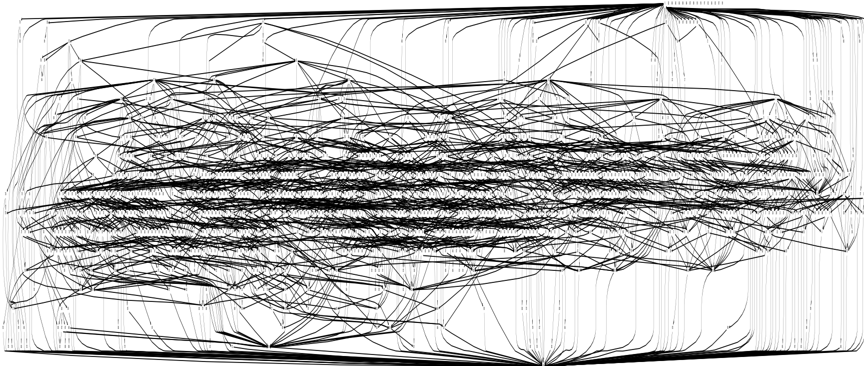


Figure 5.3: Types as items, program using type as feature

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].

5.4 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. Table 5.1 is an example of such a

relation, and Figure 5.2 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 paper this problem is solved by taking the *types* (as described in Section 5.2) of these variables, and relating them to programs in various ways.

5.4.1 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 Table 5.3.

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.

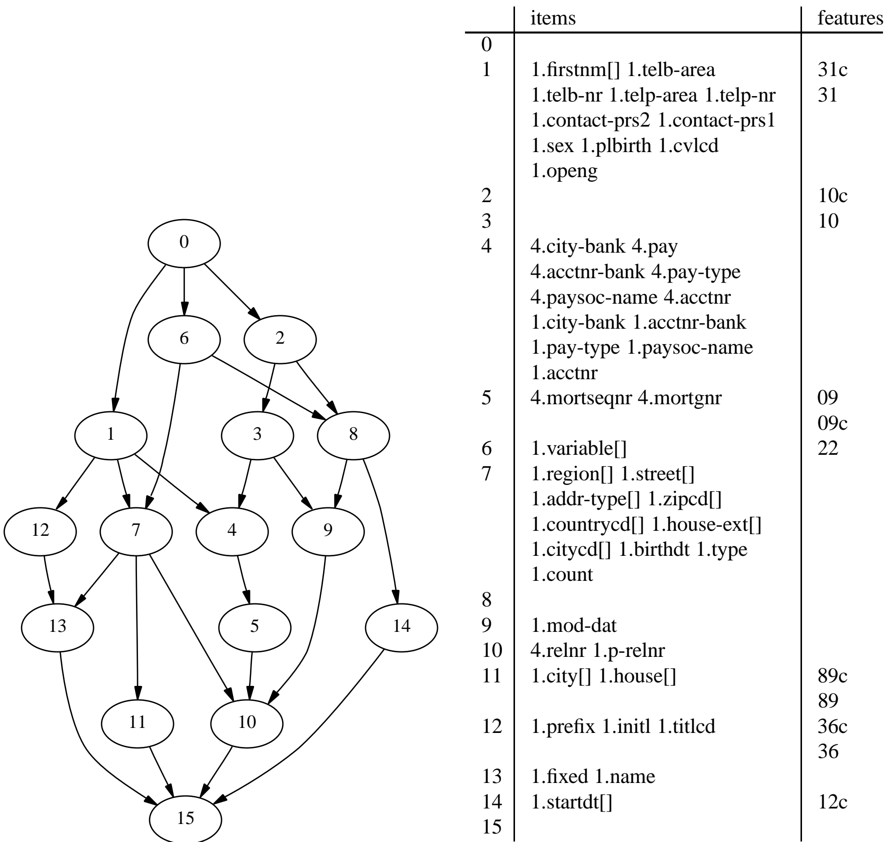
5.4.2 Experiments Performed

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

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

Table 5.3: Derived and inferred relations



	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 5.4: Concepts involving relevant programs

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 5.3. The list of items and features is not shown for (obvious) lack of space.

Filtering This picture 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.

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: $\{t \in T | (t, v) \in \mathsf{typeEquiv}, v \in V_{rel}\}$ Then the type equivalent variables that are used in the selected relevant programs are selected: $\{(v, p) \in V_{rel} \times P_{rel} | (v', p) \in \mathsf{varUsage}, (t, v') \in \mathsf{typeEquiv}, t \in T_{rel}\}$

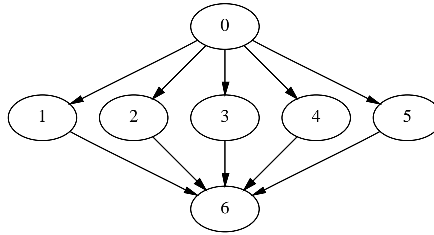
The result of the experiments with filtered data are much more comprehensible than those without filtering, basically because there are less concepts to try to understand. Figure 5.4 shows the concept lattice for the same system as in Figure 5.3, but with irrelevant programs filtered out according to [DK98]. The relevant data are the fields of the two records describing the persistent data in the system.

The lattice in Figure 5.4 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.

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



	items	features
0		
1	41	1.1.city[]
2	40	1.1.street[]
3	36c 35030u 31c 10c 05010r 01410u 01330u 01230u	1.1.record
4	36	1.1.fixed
5	31 10 09	1.1.p-relnr
6		

Figure 5.5: Programs as items, parameters as features

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 5.2, 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 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 5.4 is shown in Figure 5.5.

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.

5.5 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 5.6 shows the application of ConceptRefinery on the data of Table 5.1.

5.5.1 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.

5.5.2 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.

5.6 Implementation

We have developed a prototype toolset to perform concept analysis experiments. An overview of this toolset is shown in Figure 5.7. 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 [dBSV97a]. 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 [DM99]. 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 5.4. The

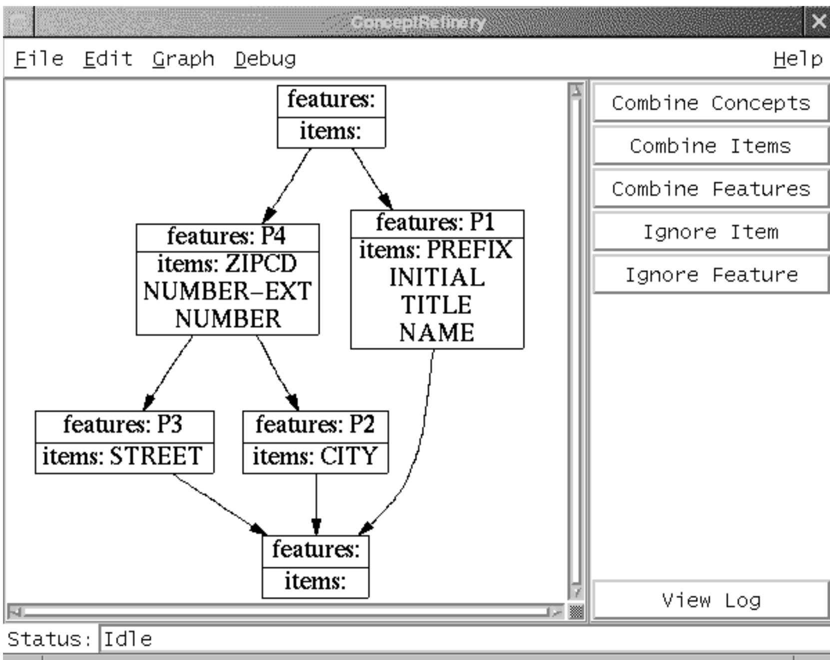


Figure 5.6: Screendump of ConceptRefinery.

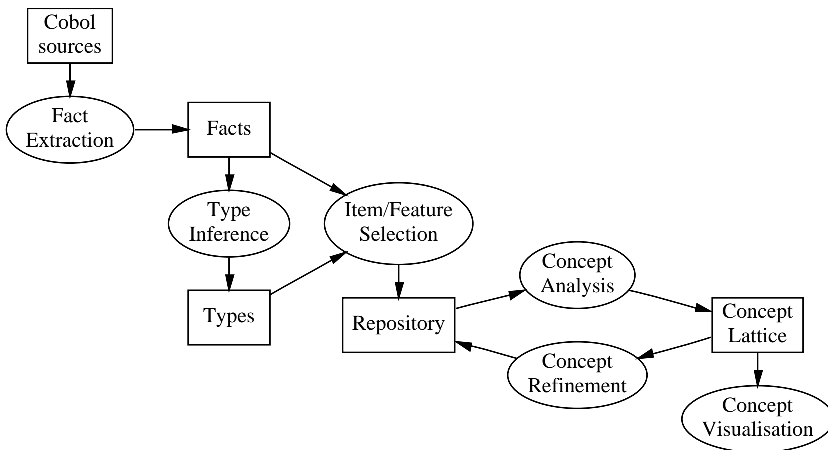


Figure 5.7: Overview of the toolset.

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.[6] 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 5.2, 5.5, 5.3 and 5.4 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 first concept to the items of the second and vice versa. The user interface of ConceptRefinery is shown in Figure 5.6. 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] The tool "`concepts`" is available from http://www.cs.tu-bs.de/softech/people/lindig/.

5.7 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 [Sne98, 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 [DM98], where we present the basic theory for Cobol type inferencing, and propose the use of subtyping to deal with pollution. In [DM99], we covered 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 [DM98, DM99] for more pointers to related work on type inferencing.

5.8 Concluding remarks

In this paper 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 paper 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.

5.8.1 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 will to experiment with applying cluster analysis algorithms to the concept lattice.

We have discussed two particular concept analysis contexts in this paper. 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 paper with Arie van Deursen are greatly appreciated. We thank Joost Visser for his comments on earlier drafts of this paper.

Chapter 6

Object-Oriented Tree Traversal with JJForester

The results presented in the previous chapter can only be achieved with data obtained from highly detailed analyses. In Chapter 3, island grammars were introduced to facilitate these analyses. However, when a system is parsed using a parser generated from a (island) grammar, the resulting parse tree needs to be analyzed. In this chapter, a technique for traversing and analyzing parse trees is developed. Furthermore, a case study of how to use the technique for the analysis of a software system is presented.[1]

6.1 Introduction

JJForester is a parser and visitor generator for Java that takes language definitions in the syntax definition formalism SDF [HHKR89, Vis97b] as input. It generates Java code that facilitates the construction, representation, and manipulation of syntax trees in an object-oriented style. To support *generalized LR parsing* [Tom85, Rek92], JJForester reuses the parsing components of the ASF+SDF Meta-Environment [Kli93].

The ASF+SDF Meta-Environment is an interactive environment for the development of language definitions and tools. It combines the syntax definition formalism SDF with the term rewriting language ASF [BHK89]. SDF is supported with generalized LR parsing technology. For language-centered software engineering

[1]This chapter was published earlier as: T. Kuipers and J. Visser. Object-oriented Tree Traversal with JJForester. In *Proceedings of the First Workshop on Language Descriptions, Tools and Applications 2001 (LDTA'01)*. Electronic Notes in Theoretical Computer Science 44(2). Elsevier Science Publishers, 2001.

applications, generalized parsing offers many benefits over conventional parsing technology [dBSV98]. ASF is a rather pure executable specification language that allows rewrite rules to be written in concrete syntax.

In spite of its many qualities, a number of drawbacks of the ASF+SDF Meta-Environment have been identified over the years. One of these is its unconditional bias towards ASF as programming language. Though ASF was well suited for the *prototyping* of language processing systems, it lacked some features to build mature *implementations*. For instance, ASF does not come with a strong library mechanism, I/O capabilities, or support for generic term traversal. Also, the closed nature of the meta-environment obstructed interoperation with external tools. As a result, for a mature implementation one was forced to abandon the prototype and fall back to conventional parsing technology. Examples are the ToolBus [BK98], a software interconnection architecture and accompanying language, that has been simulated extensively using the ASF+SDF Meta-Environment, but has been implemented using traditional Lex and Yacc parser technology and a manually coded C program. For Stratego [VBT99], a system for term rewriting with strategies, a simulator has been defined using the ASF+SDF Meta-Environment, but the parser has been hand coded using ML-Yacc and Bison. A compiler for RISLA, an industrially successful domain-specific language for financial products, has been prototyped in the ASF+SDF Meta-Environment and afterwards re-implemented in C [dB+96].

To relieve these drawbacks, the Meta-Environment has recently been re-implemented in a component-based fashion [B+00]. Its components, including the parsing tools, can now be used separately. This paves the way to adding support for alternative programming languages to the Meta-Environment.

As a major step into this direction, we have designed and implemented JJ-Forester. This tool combines SDF with the main stream general purpose programming language Java. Apart from the obvious advantages of object-oriented programming (e.g. data hiding, intuitive modularization, coupling of data and accompanying computation), it also provides language tool builders with the massive library of classes and design patterns that are available for Java. Furthermore, it facilitates a myriad of interconnections with other tools, ranging from database servers to remote procedure calls. Apart from Java code for constructing and representing syntax trees, JJForester generates visitor classes that facilitate generic traversal of these trees.

The paper is structured as follows. Section 6.2 explains JJForester. We discuss what code it generates, and how this code can be used to construct various kinds of tree traversals. Section 6.3 provides a case study that demonstrates in depth how a program analyzer (for the Toolbus language) can be constructed using JJForester.

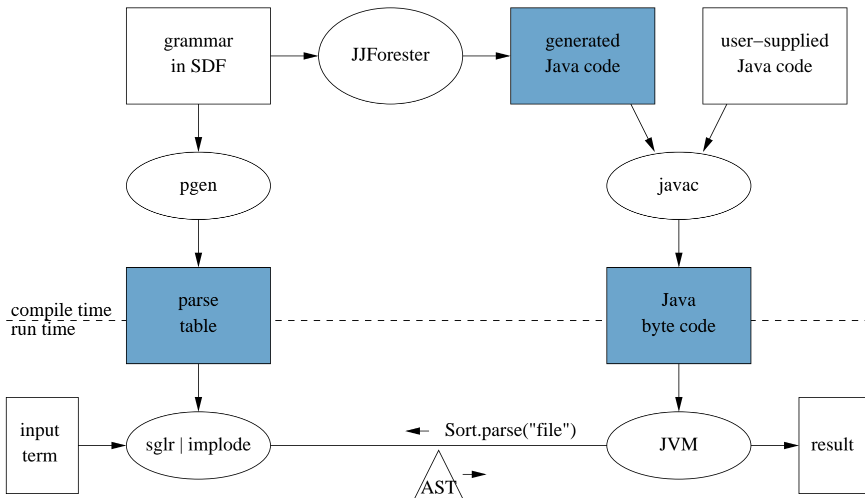


Figure 6.1: Global architecture of JJForester. Ellipses are tools. Shaded boxes are generated code.

6.2 JJForester

JJForester is a parser and visitor generator for Java. Its distinction with respect to existing parser and visitor generators, e.g. Java Tree Builder, is twofold. Firstly, it deploys generalized LR parsing, and allows *unrestricted*, *modular*, and *declarative* syntax definition in SDF (see Section 6.2.2). These properties are essential in the context of component-based language tool development where grammars are used as *contracts* [JV00]. Secondly, to cater for a number of reoccuring tree traversal scenarios, it generates variants on the Visitor pattern that allow different traversal strategies. In this section we will give an overview of JJForester. We will give a brief introduction to SDF which is used as its input language. By means of a running example, we will explain what code is generated by JJForester and how to program against the generated code.

6.2.1 Overview

The global architecture of JJForester is shown in Figure 6.1. Tools are shown as ellipses. Shaded boxes are generated code. Arrows in the bottom row depict run time events, the other arrows depict compile time events. JJForester takes a grammar defined in SDF as input, and generates Java code. In parallel, the parse table generator PGEN is called to generate a parse table from the grammar. The generated code is compiled together with code supplied by the user. When the resulting byte code is run on a Java Virtual Machine, invocations of *parse* methods

will result in calls to the parser SGLR. From a given input term, SGLR produces a parse tree as output. These parse trees are passed through the parse tree implosion tool *implode* to obtain abstract syntax trees.

6.2.2 SDF

The language definition that JJForester takes as input is written in SDF. In order to explain JJForester, we will give a short introduction to SDF. A complete account of SDF can be found in [HHKR89, Vis97b].

SDF stands for Syntax Definition Formalism, and it is just that: a formalism to define syntax. SDF allows the definition of lexical and context-free syntax in the same formalism. SDF is a modular formalism; it allows productions to be distributed at will over modules. For instance, mutually dependent productions can appear in different modules, as can different productions for the same non-terminal. This implies, for instance, that a kernel language and its extensions can be defined in different modules. Like extended BNF, SDF offers constructs to define optional symbols and iteration of symbols, but also for separated iteration, alternatives, and more.

Figure 6.2 shows an example of an SDF grammar. This example grammar gives a modular definition of a tiny lambda calculus-like language with typed lambda functions. Note that the orientation of SDF productions is reversed with respect to BNF notation. The grammar contains two context-free non-terminals, Expr and Type, and two lexical non-terminals, Identifier and LAYOUT. The latter non-terminal is used *implicitly* between all symbols in context-free productions. As the example details, expressions can be variables, applications, or typed lambda abstractions, while types can be type variables or function types.

SDF's expressiveness allows for defining syntax concisely and naturally. SDF's modularity facilitates reuse. SDF's declarativeness makes it easy and retargetable. But the most important strength of SDF is that it is supported by *Generalized LR Parsing*. Generalized parsing removes the restriction to a non-ambiguous subclass of the context-free grammars, such as the LR(k) class. This allows a maximally natural expression of the intended syntax; no more need for 'bending over backwards' to encode the intended grammar in a restricted subclass. Furthermore, generalized parsing leads to better modularity and allows 'as-is' syntax reuse.

As SDF removes any restriction on the class of context-free grammars, the grammars defined with it potentially contain ambiguities. For most applications, these ambiguities need to be resolved. To this end, SDF offers a number of disambiguation constructs. The example of Figure 6.2 shows four such constructs. The *left* and *right* attributes indicate associativity. The *bracket* attribute indicates that parentheses can be used to disambiguate Exprs and Types. For the lexical non-terminals the longest match rule is explicitly specified by means of *follow restrictions*. Not shown in the example is SDF's notation for relative priorities.

```
module Expr
exports
  context-free syntax
    Identifier         -> Expr {cons("Var")}
    Expr Expr           -> Expr {cons("Apply"), left}
    "\\" Identifier ":" Type "." Expr
                        -> Expr {cons("Lambda")}
    "(" Expr ")"        -> Expr {bracket}
```

```
module Type
exports
  context-free syntax
    Identifier          -> Type {cons("TVar")}
    Type "->" Type      -> Type {cons("Arrow"),right}
    "(" Type ")"        -> Type {bracket}
```

```
module Identifier
exports
  lexical syntax
    [A-Za-z0-9]+ -> Identifier
  lexical restrictions
    Identifier -/- [A-Za-z0-9]
```

```
module Layout
exports
  lexical syntax
    [\ \t\n] -> LAYOUT
  context-free restrictions
    LAYOUT? -/- [\ \t\n]
```

Figure 6.2: Example SDF grammar.

In the example grammar, each context-free production is attributed with a *constructor name*, using the *cons(..)* attribute. Such a grammar with constructor names amounts to a simultaneous definition of concrete and abstract syntax of the language at hand. The *implode* back-end turns concrete parse trees emanated by the parser into more concise abstract syntax trees (ASTs) for further processing. The constructor names defined in the grammar are used to build nodes in the AST. As will become apparent below, JJForester operates on these abstract syntax trees, and thus requires grammars with constructor names. A utility, called *sdf-cons* is available to automatically synthesize these attributes when absent.

SDF is supported by two tools: the parse table generator PGEN, and the scannerless generalized parser SGLR. These tools were originally developed as components of the ASF+SDF Meta-Environment and are now separately available as stand-alone, reusable tools.

6.2.3 Code generation

From an SDF grammar, JJForester generates the following Java code:

Class structure For each non-terminal symbol in the grammar, an *abstract* class is generated. For each production in the grammar, a *concrete* class is generated that extends the abstract class corresponding to the result non-terminal of the production. For example, Figure 6.3 shows a UML diagram of the code that JJForester generates for the grammar in Figure 6.2. The relationships between the abstract classes *Expr* and *Type*, and their concrete subclasses are known as the Composite pattern.

Lexical non-terminals and productions are treated slightly differently: for each lexical non-terminal a class can be supplied by the user. Otherwise, this lexical non-terminal is replaced by the pre-defined non-terminal `Identifier`, for which a single concrete class is provided by JJForester. This is the case in our example.

When the input grammar, unlike our example, contains complex symbols such as optionals or iterated symbols, additional classes are generated for them as well. The case study will illustrate this.

Parsers Also, for every non-terminal in the grammar, a parse method is generated for parsing a term (plain text) and constructing a tree (object structure). The actual parsing is done externally by SGLR. The parse method implements the Abstract Factory design pattern; each non-terminal class has a parse method that returns an object of the type of one of the constructors for that non-terminal. Which object gets returned depends on the string that is parsed.

Constructor methods In the generated classes, constructor methods are generated that build *language-specific* tree nodes from the generic tree that results from

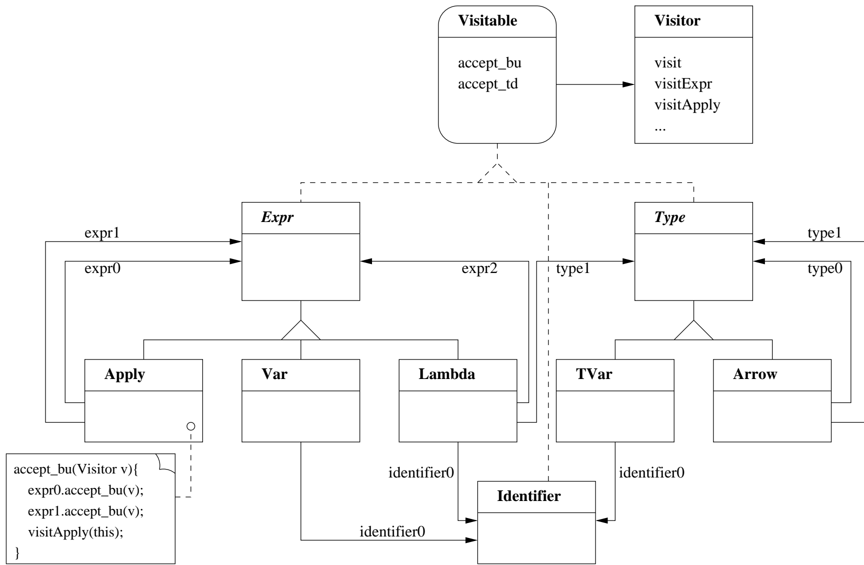


Figure 6.3: The UML diagram of the code generated from the grammar in Figure 6.2.

the call to the external parser.

Set and get methods In the generated concrete classes, set and get methods are generated to inspect and modify the fields that represent the subtrees. For example, the Apply class will have `getExpr0` and `setExpr0` methods for its first child.

Accept methods In the generated concrete classes, several accept methods are generated that take a Visitor object as argument, and apply it to a tree node. Currently, two *iterating* accept methods are generated: `accept_td` and `accept_bu`, for top-down and bottom-up traversal, respectively. For the Apply class, the bottom-up accept method is shown in the Figure 6.3.

Visitor classes A Visitor class is generated which contains a visit method for each production and each non-terminal in the grammar. Furthermore, it contains one unqualified visit method which is useful for *generic* refinements (see below). These visit methods are *non-iterating*: they make no calls to accept methods of children to obtain recursion. The default behavior offered by these generated visit methods is simply to do nothing.

Together, the Visitor class and the accept methods in the various concrete classes implement a variant of the Visitor pattern [GHJV94], where the respon-

sibility for iteration lies with the accept methods, not with the visit methods. We have chosen this variant for several reasons. First of all, it relieves the programmer who specializes a visitor from reconstructing the iteration behavior in the visit methods he redefines. This makes specializing visitors less involved and less error-prone. In the second place, it allows the iteration behavior (top-down or bottom-up) to be varied. In Section 6.4.3 we will comment on the possibilities of offering even more control over iteration behavior.

Apart from generating Java code, JJForester calls PGEN to generate a parse table from its input grammar. This table is used by SGLR which is called by the generated parse methods.

6.2.4 Programming against the generated code

The generated code can be used by a tool builder to construct tree traversals through the following steps:

1. Refine a visitor class by redefining one or more of its visit methods. As will be explained below, such refinement can be done at various levels of genericity, and in a step-wise fashion.
2. Start a traversal with the refined visitor by feeding it to the accept method of a tree node. Different accept methods are available to realize top-down or bottom-up traversals.

This method of programming traversals by refining (generated) visitors provides interesting possibilities for reuse. Firstly, many traversals only need to do something 'interesting' at a limited number of nodes. For these nodes, the programmer needs to supply code, while for all others the behavior of the generated visitor is inherited. Secondly, different traversals often share behavior for a number of nodes. Such common behavior can be captured in an initial refinement, which is then further refined in diverging directions. Unfortunately, Java's lack of multiple inheritance prohibits the converse: construction of a visitor by inheritance from two others (but see Section 6.4.3 for further discussion). Thirdly, some traversal actions may be specific to nodes with a certain constructor, while other actions are the same for all nodes of the same type (non-terminal), or even for all nodes of any type. As the visitors generated by JJForester allow refinement at each of these levels of specificity, there is no need to repeat the same code for several constructors or types. We will explain these issues through a number of small examples.

Constructor-specific refinement Figure 6.4 shows a refinement of the Visitor class which implements a traversal that counts the number of variables occurring in a syntax tree. Both expression variables and type variables are counted.

```
public class VarCountVisitor extends Visitor {
    public int counter = 0;
    public void visitVar(Var x) {
        counter++;
    }
    public void visitTVar(TVar x) {
        counter++;
    }
}
```

Figure 6.4: Specific refinement: a visitor for counting variables.

```
public class ExprCountVisitor extends Visitor {
    public int counter = 0;
    public void visitExpr(Expr x) {
        counter++;
    }
}
```

```
public class NodeCountVisitor extends Visitor {
    public int counter = 0;
    public void visit(Object x) {
        counter++;
    }
}
```

Figure 6.5: Generic refinement: visitors for counting expressions and nodes.

This refinement extends Visitor with a counter field, and redefines the visit methods for Var and TVar such that the counter is incremented when such nodes are visited. The behavior for all other nodes is inherited from the generated Visitor: do nothing. Note that redefined methods need not restart the recursion behavior by calling an accept method on the children of the current node. The recursion is completely handled by the generated accept methods.

Generic refinement The refinement in the previous example is specific for particular node constructors. The visitors generated by JJForester additionally allow more generic refinements. Figure 6.5 shows refinements of the Visitor class that implement a more generic expression counter and a fully generic node counter. Thus, the first visitor counts all expressions, irrespective of their constructor, and the second visitor counts all nodes, irrespective of their type. No code duplication is necessary.

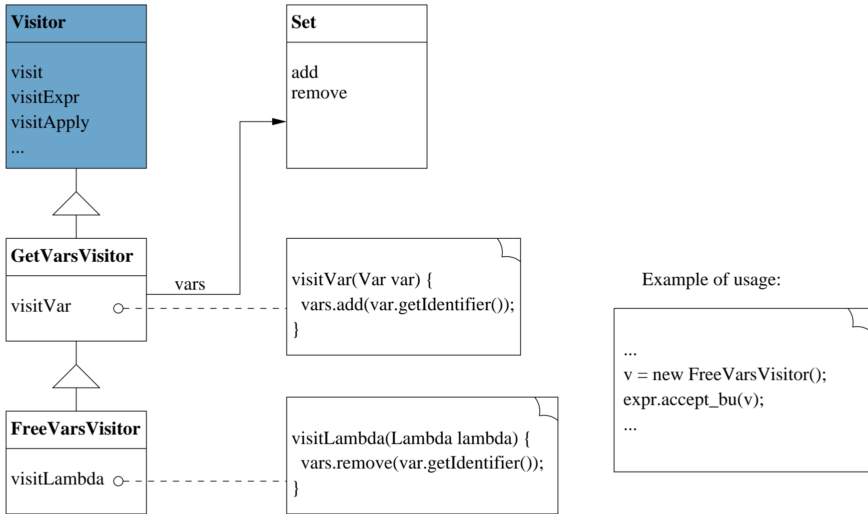


Figure 6.6: UML diagram for user code.

Step-wise refinement Visitors can be refined in several steps. For our example grammar, two subsequent refinements of the Visitor class are shown in Figure 6.6. The class GetVarVisitor is a visitor for collecting all variables used in expressions. It is defined by extending the Visitor class with a field `vars` initialized as the empty set of variables, and by redefining the visit method for the Var class to insert each variable it encounters into this set. The GetVarVisitor is further refined into a visitor that collects *free* variables, by additionally redefining the visit method for the Lambda class. This redefined method removes the variables bound by the lambda expression from the current set of variables. Finally, this second visitor can be unleashed on a tree using the `accept_bu` method. This is illustrated by an example of usage in Figure 6.6.

Note that the visitors in Figures 6.4 and 6.5 can be refactored as refinements of a common initial refinement, say CountVisitor, which contains only the field counter.

Of course, our running example does not mean to suggest that Java would be the ideal vehicle for implementing the lambda calculus. Our choice of example was motivated by simplicity and self-containedness. To compare, an implementation of the lambda calculus in the ASF+SDF Meta-Environment can be found in [DHK96]. In Section 6.3 we will move into the territory for which JJForester is intended: component-based development of program analyses and transformations for languages of non-trivial size.

6.2.5 Assessment of expressiveness

To evaluate the expressiveness of JJForester within the domain of language processing, we will assess which program transformation scenarios can be addressed with it. We distinguish three main scenarios:

Analysis A value or property is distilled from a syntax tree. Type-checking is a prime example.

Translation A program is transformed into a program in a different language. Examples include generating code from a specification, and compilation.

Rephrasing A program is transformed into another program, where the source and target language coincide. Examples include normalization and renovation.

For a more elaborate taxonomy of program transformation scenarios, we refer to [V$^+$]. The distinction between analysis and translation is not clear-cut. When the value of an analysis is highly structured, especially when it is an expression in another language, the label 'translation' is also appropriate.

The traversal examples discussed above are all tree analyses with simple accumulation in a state. Here, 'simple' accumulation means that the state is a value or collection to which values are added one at a time. This was the case both for the counting and the collecting examples. However, some analyses require more complex ways of combining the results of subtree traversals than simple accumulation. An example is pretty-printing, where literals need to be inserted *between* pretty-printed subtrees. In the case study, a visitor for pretty-printing will demonstrate that JJForester is sufficiently expressive to address such more complex analyses. However, a high degree of reuse of the generated visit methods can currently only be realized for the simple analyses. In the future work section (6.4.3), we will discuss how such reuse could be realized by generating special visitor subclasses or classes that model updatable many-sorted folds [LVK00].

Translating transformations are also completely covered by JJForester's expressiveness. As in the case of analysis, the degree of reuse of generated visit methods can be very low. Here, however, the cause lies in the nature of translation, because it typically takes every syntactic construct into account. This is not always the case, for instance, when the translation has the character of an analysis with highly structured results. An example is program visualization where only dependencies of a particular kind are shown, e.g. module structures or call graphs.

In the object-oriented setting, a distinction needs to be made between destructive and non-destructive rephrasings. Destructive rephrasings are covered by JJForester. However, as objects can not modify their *self* reference, destructive modifications can only change subtrees and fields of the current node, but they cannot replace the current node by another. Non-destructive rephrasings can be imple-

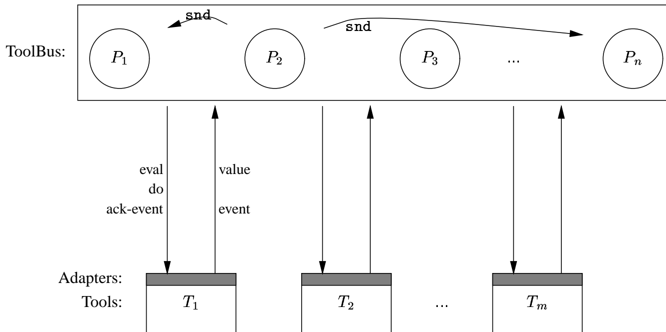


Figure 6.7: The Toolbus architecture. Tools are connected to the bus through adapters. Inside the bus, several processes run in parallel. These processes communicate with each other and the adapters according to the protocol defined in a T-script.

mented by refining a traversal that clones the input tree. A visitor for tree cloning can be generated, as will be discussed in Section 6.4.3.

A special case of rephrasing is decoration. Here, the tree itself is traversed, but not modified except for designated attribute fields. Decoration is useful when several traversals are sequenced that need to share information about specific nodes. JJForester does not cover decoration yet.

6.3 Case study

Now that we have explained the workings of JJForester, we will show how it is used to build a program analyzer for an actual language. In particular, this case study concerns a static analyzer for the ToolBus [BK98] script language. In Section 6.3.1 we describe the situation from which a need for a static analyzer emerged. In Section 6.3.2 the language to be analyzed is briefly explained. Finally, Section 6.3.3 describes in detail what code needs to be supplied to implement the analyzer.

6.3.1 The Problem

The ToolBus is a coordination language which implements the idea of a software bus. It allows applications (or *tools*) to be “plugged into” a bus, and to communicate with each other over that bus. Figure 6.7 gives a schematic overview of the ToolBus. The protocol used for communication between the applications is not fixed, but is programmed through a ToolBus script, or T-script.

A T-script defines one or more processes that run inside the ToolBus in parallel. These processes can communicate with each other, either via synchronous point-

to-point communication, or via asynchronous broadcast communication. The processes can direct and activate external components via *adapters*, small pieces of software that translate the ToolBus's remote procedure calls into calls that are native to the particular software component that needs to be activated. Adapters can be compiled into components, but off-the-shelf components can be used, too, as long as they possess some kind of external interface.

Communication between processes inside the ToolBus does not occur over named channels, but through pattern matching on terms. Communication between processes occurs when a term sent by one matches the term that is expected by another. This will be explained in more detail in the next section. This style of communication is powerful, flexible and convenient, but tends to make it hard to pinpoint errors in T-scripts. To support the T-script developer, the ToolBus runtime system provides an interactive visualizer, which shows the communications taking place in a running ToolBus. Though effective, this debugging process is tedious and slow, especially when debugging systems with a large number of processes.

To complement the runtime visualizer, a *static* analysis of T-scripts is needed to support the T-script developer. Static analysis can show that some processes can never communicate with each other, that messages that are sent can never be received (or vice versa), or that two processes that should not communicate with each other may do so anyway. Using JJForester, such a static analyzer is constructed in Section 6.3.3.

6.3.2 T-scripts explained

T-scripts are based on ACP (Algebra of Communicating Processes) [BV95]. They define communication protocols in terms of *actions*, and operations on these actions. We will be mainly concerned with the communication actions, which we will describe below. Apart from these, there are assignment actions, conditional actions and basic arithmetic actions. The action operators include sequential composition ($a.b$), non-deterministic choice ($a + b$), parallel composition ($a \mid\mid b$), and repetition ($a * b$). The full specification of the ToolBus script language can be found in [BK94].

The T-script language offers actions for communication between processes and tools, and for synchronous and asynchronous communication between processes. For the purposes of this paper we will limit ourselves to the most commonly used *synchronous* actions. These are `snd-msg(T)` and `rec-msg(T)` for sending and receiving messages, respectively. These actions are parameterized with arbitrary data `T`, represented as ATerms [BJKO00]. A successful synchronous communication occurs when a term that is sent matches a term that is received. For instance, the closed term `snd-msg(f(a))` can match the closed term `rec-msg(f(a))` or the open term `rec-msg(f(T?))`. At successful communication, variables in the data of the receiving process are instantiated according to the match.

```
process Pump is
let D: int
in
( rec-msg(activate(D?)).
  rec-msg(on).
  snd-msg(report(D))
) *
delta
endlet

process Operator is
let C: int, D: int,
    Payment: int, Amount: int
in
( rec-msg(request(D?,C?)).
  Payment := D.
  snd-msg(schedule(Payment,C)).
  rec-msg(result(D?)).
  Amount := sub(Payment,D).
  snd-msg(remit(Amount))
) *
delta
endlet
process Customer is
let
C: int, D: int
in
C := process-id.
D := 10.
snd-msg(prepay(D,C)).
rec-msg(okay(C)).
snd-msg(turn-on).
printf(
 "Customer %d using pump\n",
  C).
rec-msg(stop).
rec-msg(change(D?)).
printf(
 "Customer %d got $%d change\n",
  C, D)
endlet

process GasStation is
let
D: int, C: int
in
( rec-msg(prepay(D?,C?)).
  snd-msg(request(D,C))
||rec-msg(schedule(D?,C?)).
  snd-msg(activate(D)).
  snd-msg(okay(C))
||rec-msg(turn-on).
  snd-msg(on)
||rec-msg(report(D?)).
  snd-msg(stop).
  snd-msg(result(D))
||rec-msg(remit(D?)).
  snd-msg(change(D))
)*
delta
endlet

toolbus(GasStation,Pump,
  Customer,Customer,Operator)
```

Figure 6.8: The T-script for the gas station with control process.

To illustrate, a small example T-script is shown in Figure 6.8. This example contains only processes. In a more realistic situation these processes would communicate with external tools, for instance to get the input of the initial value, and to actually activate the gas pump. The script's last statement is a mandatory `toolbus(..)` statement, which declares that upon startup the processes GasStation, Pump, Customer and Operator are all started in parallel. The first action of all processes, apart from Customer, is a `rec-msg` action. This means that those processes will block until an appropriate communication is received. The Customer process starts by doing two assignment statements. `process-id` (a built-in variable that contains the identifier of the current process) is assigned to `C`, and 10 to `D`. The first communication action performed by Customer is a `snd-msg` of the term `prepay(D,C)`. This term is received by the GasStation process, which in turn sends the term `request(D,C)` message. This is received by Operator, and so on.

The script writer can use the mechanism of communication through term matching to specify that any one of a number of processes should receive a message, depending on the state they are in, and the sending process does not need to know this. It just sends out a term into the ToolBus, and anyone of the accepting pro-

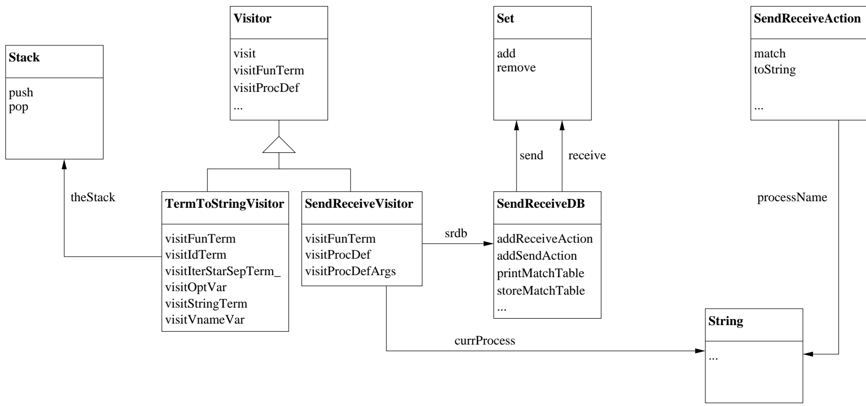


Figure 6.9: UML diagram of the ToolBus analyzer.

cesses can "pick it up". Unfortunately, when incorrect or too general terms are specified, communication will not occur as expected, and the exact cause will be difficult to trace. The static analyzer developed in the next section is intended to solve this problem.

6.3.3 Analysis using JJForester

We will first sketch the outlines of the static analysis algorithm that we implemented. It consists of two phases: collection and matching. In the collection phase, *all* send and receive actions in the T-script are collected into a (internal, non-persistent) database. In the matching phase, the send and receive actions in the database are matched to obtain a table of potential matching events, which can either be stored in a file, or in an external, persistent relational database. To visualize this table, we use the back-end tools of a documentation generator we developed earlier (DocGen [DK99a]).

We used JJForester to implement the parsing of T-scripts and the representation and traversal of T-script parse trees. To this end, we ran JJForester on the grammar of the ToolBus[2] which contains 35 non-terminals and 80 productions (both lexical and context-free). From this grammar, JJForester generated 23 non-terminal classes, 64 constructor classes, and 1 visitor class, amounting to a total of 4221 lines of Java code.

We will now explain in detail how we programmed the two phases of the analysis. Figure 6.9 shows a UML diagram of the implementation.

[2]This SDF grammar can be downloaded from the GrammarBase, at http://www.program-transformation.org/gb.

```
context-free syntax
 "process" ProcessName "is" ProcessExpr
      -> ProcessDef  {cons("procDef")}
 "process" ProcessName "(" {VarDecl ","}* ")" "is" ProcessExpr
      -> ProcessDef  {cons("procDefArgs")}
```

Figure 6.10: The syntax of process definitions.

```
public void visitProcDef(procDef definition) {
  currProcess = definition.getIdentifier0().toString();
}
public void visitProcDefArgs(procDefArgs definition) {
  currProcess = definition.getIdentifier0().toString();
}
```

Figure 6.11: Specialized visit methods to extract process definition names.

The collection phase

We implemented the collection phase as a top-down traversal of the syntax tree with a visitor called SendReceiveVisitor. This refinement of the Visitor class has two kinds of state: a database for storing send and receive actions, and a field that indicates the name of the process currently being analyzed. Whenever a term with outermost function symbol snd-msg or rec-msg is encountered, the visitor will add a corresponding action to the database, tagged with the current process name. The current process name is set whenever a process definition is encountered during traversal. Since sends and receives occur only *below* process definition in the parse tree, the top-down traversal strategy guarantees that the current process name field is always correctly set when it is needed to tag an action.

To discover which visit methods need to be redefined in the SendReceiveVisitor, the ToolBus grammar needs to be inspected. To extract process definition names, we need to know which syntactic constructs are used to declare these names. The two relevant productions are shown in Figure 6.10. So, in order to extract process names, we need to redefine `visitProcDef` and `visitProcDefArgs` in our specialized SendReceiveVisitor. These redefinitions are shown in Figure 6.11. Whenever the built-in iterator comes across a node in the tree of type `procDef`, it will call our specialized `visitProcDef` with that `procDef` as argument. From the SDF definition in Figure 6.10 we learn that a `procDef` has two children: a ProcessName and a ProcessExpr. Since ProcessName is a *lexical* non-terminal, and we chose to have JJForester identify all lexical non-terminals with a single type `Identifier`, the Java class `procDef` has a field of type `Identifier` and one of type `ProcessExpr`. Through the `getIdentifier0()` method we get the actual process name which gets converted to a String so it can be assigned to `currProcess`.

```
context-free syntax
  Vname                 -> Var      {cons("vnameVar")}
  Var                   -> GenVar   {cons("var")}
  Var "?"               -> GenVar   {cons("optVar")}
  GenVar                -> Term     {cons("genvarTerm")}
  Id                    -> Term     {cons("idTerm")}
  Id "(" TermList ")"   -> Term     {cons("funTerm")}
  {Term ","}*           -> TermList {cons("termStar")}
  Term                  -> Atom     {cons("termAtom")}
```

Figure 6.12: Syntax of relevant ToolBus terms.

```
public void visitFunTerm(funTerm term) {
  SendReceiveAction action = new
    SendReceiveAction(currProcess, term.getTermlist1());
  if (term.getIdentifier0().equals("\"snd-msg\"")) {
    srdb.addSendAction(action);
  } else if (term.getIdentifier0().equals("\"rec-msg\"")) {
    srdb.addReceiveAction(action);
  }
}
```

Figure 6.13: The visit method for send and receive messages.

Now that we have taken care of extracting process names, we need to address the collection of communication actions. The ToolBus grammar allows for arbitrary terms ('Atoms' in the grammar) as actions. Their syntax is shown in Figure 6.12.

Thus, send and receive actions are not distinct syntactical constructs, but they are functional terms (`funTerms`) where the `Id` child has value `snd-msg` or `rec-msg`. Consequently, we need to redefine the `visitFunTerm` method such that it inspects the value of its first child to decide if and how to collect a communication action. Figure 6.13 shows the redefined method.

The visit method starts by constructing a new `SendReceiveAction`. This is an object that contains the term that is being communicated and the process that sends or receives it. The process name is available in the `SendReceiveVisitor` in the field `currProcess`, because it is put there by the `visitProcDef` methods we just described. The term that is being communicated can be selected from the `funTerm` we are currently visiting. From the SDF grammar in Figure 6.12 it follows that the term is the second child of a `funTerm`, and that it is of type `TermList`. Therefore, the method `getTermlist1` will return it.

The newly constructed action is added to the database as a send action, a receive action, or not at all, depending on the first child of the `funTerm`. This child is of lexical type `Id`, and thus converted to an `Identifier` type in the generated

```
public static void main(String[] args) throws ParseException {
  String inFile = args[0];
  Tscript theScript = Tscript.parse(inFile);
  SendReceiveVisitor srvisitor = new SendReceiveVisitor();
  theScript.accept_td(srvisitor);       // collection phase
  srvisitor.srdb.constructMatchTable(); // matching phase
}
```

Figure 6.14: The main() method of the ToolBus analyzer.

Java classes. The `Identifier` class contains an `equals(String)` method, so we use string comparison to determine whether the current `funTerm` has “snd-msg” or “rec-msg” as its function symbol.

Now that we have built the specialized visitor to perform the collection, we still need to activate it. Before we can activate it, we need to have parsed a T-script, and built a class structure out of the parse tree for the visitor to operate on. This is all done in the `main()` method of the analyzer, as shown in Figure 6.14. The main method shows how we use the generated parse method for `Tscript` to build a tree of objects. Tscript.parse() takes a filename as an argument and tries to parse that file as a Tscript. If it fails it throws a ParseException and displays the location of the parse error. If it succeeds it returns a `Tscript`. We then construct a new `SendReceiveVisitor` as described in the previous section. The `Tscript` is subsequently told to accept this visitor, and, as described in Section 6.2.4 iterates over all the nodes in the tree and calls the specific visit methods for each node. When the iterator has visited all nodes, the `SendReceiveVisitor` contains a filled `SendReceiveDb`. The results in this database object can then be processed further, in the matching phase. In our case we call the method `constructMatchTable()` which is explained below.

The matching phase

In the matching phase, the send and receive actions collected in the `SendReceiveDb` are matched to construct a table of potential communication events, which is then printed to file or stored in a relational database. We will not discuss the matching itself in great detail, because it is not implemented with a visitor. A visitor implementation would be possible, but clumsy, since two trees need to be traversed simultaneously. Instead it is implemented with nested iteration over the sets of send and receive actions in the database, and simple case discrimination on terms. The result of matching is a table where each row contains the process names and data of a matching send and receive action.

We focus on an aspect of the matching phase where a visitor *does* play a role. When writing the match table to file, the terms (data) it contains need to be pretty-printed, i.e. to be converted to `String`. We implemented this pretty-printer with a

```
public void visitIterStarSepTerm_(iterStarSepTerm_ terms) {
  Vector v = terms.getTerm0();
  String str = new String();
  for (int i = 0; i < v.size(); i++){
    if (i != 0) {
      str += ",";
    }
    str += (String) theStack.pop();
  }
  theStack.push(str);
}
```

Figure 6.15: Converting a list of terms to a string.

bottom-up traversal with the `TermToStringVisitor`. We chose not to use generated `toString` methods of the constructor classes, because using a visitor leaves open the possibility of refining the pretty-print functionality.

Note that pretty-printing a node may involve inserting literals before, inbetween, and after its pretty-printed children. In particular, when we have a list of terms, we would like to print a "," between children. To implement this behavior, a visitor with a single `String` field in combination with a top-down or bottom-up accept method does not suffice. If JJForester would generate *iterating* visitors and *non-iterating* accept methods, this complication would not arise. Then, literals could be added to the `String` field in between recursive calls.

We overcome this complication by using a visitor with a *stack* of strings as field, in combination with the bottom-up accept method. The visit method for each leaf node pushes the string representation of that leaf on the stack. The visit method for each internal node pops one string off the stack for each of its children, constructs a new string from these, possibly adding literals in between, and pushes the resulting string back on the stack. When the traversal is done, the user can pop the last element off the stack. This element is the string representation of the visited term. Figure 6.15 shows the visit method in the `TermToStringVisitor` for lists of terms separated by commas[3]. In this method, the Vector containing the term list is retrieved, to get the number of terms in this list. This number of elements is then popped from the stack, and commas are placed between them. Finally the new string is placed back on the stack. In the conclusion we will return to this issue, and discuss alternative and complementary generation schemes that make implementing this kind of functionality more convenient.

After constructing the matching table, the `constructMatchTable` method writes the table to file or stores it in an SQL database, using JDBC (Java Database

[3]The name of the method reflects the fact that this is a visit method for the symbol {`Term ","`}*, i.e. the list of zero or more elements of type Term, separated by commas. Because the comma is an illegal character in a Java identifier, it is converted to an underscore in the method name.

Sender		Receiver	
Pump	report(D)	GasStation	report(D?)
GasStation	change(D)	Customer	change(D?)
Customer	prepay(D,C)	GasStation	prepay(D?,C?)
GasStation	okay(C)	Customer	okay(C)
Operator	remit(Amount)	GasStation	remit(D?)
GasStation	result(D)	Operator	result(D?)
GasStation	activate(D)	Pump	activate(D?)
GasStation	stop	Customer	stop
Customer	turn-on	GasStation	turn-on
Operator	schedule(Payment,C)	GasStation	schedule(D?,C?)
GasStation	request(D,C)	Operator	request(D?,C?)
GasStation	on	Pump	on

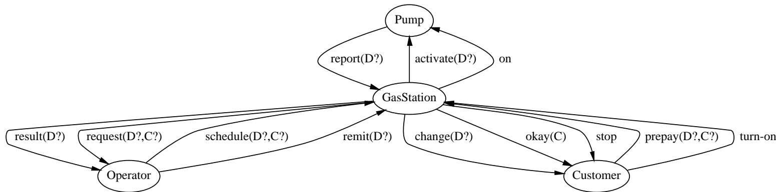


Figure 6.16: The analysis results for the input file from Figure 6.8.

Connectivity). We used a visualization back-end of the documentation generator DocGen to query the database and generate a *communication* graph. The result of the full analysis of the T-script in Figure 6.8 is shown in Figure 6.16.

Evaluation of the case study

We conducted the ToolBus case study to learn about feasibility, productivity, performance, and connectivity issues surrounding JJForester. Below we briefly discuss our preliminary conclusions. Apart from the case study reported here, we conducted a case study where an existing Perl component in the documentation generator DocGen was re-implemented in Java, using JJForester. This case study also corroborates our findings.

Feasibility At first glance, the object-oriented programming paradigm may seem to be ill-suited for language processing applications. Terms, pattern-matching, many-sorted signatures are typically useful for language processing, but are not native to an object-oriented language like Java. More generally, the reference semantics of objects seems to clash with the value semantics of terms in a language. Thus, in spite of Java's many advantages with respect to e.g. portability, maintainability, reuse, its usefulness in language processing is not evident.

The case study, as well as the techniques for coping with traversal scenarios outlined in Section 6.2, demonstrate that object-oriented programming *can* be applied usefully to language processing problems. In fact, the support offered by JJForester makes object-oriented language processing not only feasible, but even easy.

Productivity Recall that the Java code generated by JJForester from the ToolBus grammar amounts to 4221 lines of code. By contrast, the user code we developed to program the T-script analyzer consists of 323 lines. Thus, 93% of the application was generated, while 7% is hand-written.

These figures indicate that the potential for increased development productivity is considerable when using JJForester. Of course, actual productivity gains are highly dependable on which program transformation scenarios need to be addressed (see Section 6.2.5). The productivity gain is largly attributable to the support for generic traversals.

Components and connectivity Apart from reuse of generated code, the case study demonstrates reuse of standard Java libraries and of external (non-Java) tools. Examples of such tools are PGEN, SGLR and *implode*, an SQL database, and the visualization back-end of DocGen. Externally, the syntax trees that JJForester operates upon are represented in the common exchange format ATerms. This exchange format was developed in the context of the ASF+SDF Meta-Environment, but has been used in numerous other contexts as well. In [JV00] we advocated the use of grammars as tree type definitions that fix the interface between language tools. JJForester implements these ideas, and can interact smoothly with tools that do the same. The transformation tool bundle XT [JVV00] contains a variety of such tools.

Performance To get a first indication of the time and space performance of applications developed with JJForester, we have applied our T-script analyzer to a script of 2479 lines. This script contains about 40 process definitions, and 700 send and receive actions. We used a machine with Mobile Pentium processor, 64Mb of memory, running at 266Mhz. The memory consumption of this experiment did not exceed 6Mb. The runtime was 69 seconds, of which 9 seconds parsing, 55 seconds implosion, and 5 seconds to analyze the syntax tree. A safe conclusion seems to be that the Java code performs acceptably, while the implosion tool needs optimization. Needless to say, larger applications and larger code bases are needed for a good assessment.

6.4 Concluding remarks

6.4.1 Contributions

In this paper we set out to combine SDF support of the ASF+SDF Meta-Environment with the general-purpose object-oriented programming language Java. To this end we designed and implemented JJForester, a parser and visitor generator for Java that takes SDF grammars as input. To support generic traversals, JJForester generates non-iterating visitors and iterating accept methods. We discussed techniques for programming against the generated code, and we demonstrated these in detail in a case study. We have assessed the expressivity of our approach in terms of the program-transformation scenarios that can be addressed with it. Based on the case study, we evaluated the approach with respect to productivity, and performance issues.

6.4.2 Related Work

A number of parser generators, "tree builders", and visitor generators exist for Java. JavaCC is an LL parser generator by Metamata/Sun Microsystems. Its input format is not modular, it allows Java code in semantic actions, and separates parsing from lexical scanning. JJTree is a preprocessor for JavaCC that inserts parse tree building actions at various places in the JavaCC source. The Java Tree Builder (JTB) is another front-end for JavaCC for tree building and visitor generation. JTB generates two iterating (bottom-up) visitors, one with and one without an extra argument in the visit methods to pass objects down the tree. A version of JTB for GJ (Generic Java) exists which takes advantages of type parameters to prevent type casts. Demeter/Java is an implementation of adaptive programming [PXL95] for Java. It extends the Java language with a little (or domain-specific) language to specify traversal strategies, visitor methods, and class diagrams. Again, the underlying parser generator is JavaCC. JJForester's main improvement with respect to these approaches is the support of *generalized* LR parsing. Concerning traversals, JJForester is different from JJTree and JTB, because it generates iterating accept methods rather than iterating visitors. JJForester is less ambitious and more lightweight than Demeter/Java, which is a programming system rather than a code-generator.

ASDL (Abstract Syntax Definition Language [WAKS97]) comes with a visitor generator for Java (and other languages). It generates non-iterating visitors and non-iterating accept methods. Thus, traversals are not supported. ASDL does not incorporate parsing or parser generation; it only addresses issues of *abstract* syntax.

In other programming paradigms, work has been done on incorporating support for SDF and traversals. Previously, we have combined the SDF support of the ASF+SDF Meta-Environment with the functional programming language Haskell [KLV00].

In this approach, traversal of syntax trees is supported with updatable, many-sorted folds and fold combinators [LVK00]. Recently, support for generic traversals has been added to the ASF interpreter. These traversals allow concise specification of many-sorted analyses and rephrasing transformations. Stepwise refinement or generic refinement of such traversals is not supported. Stratego [VBT99] is a language for term rewriting with strategies. It offers a suite of primitives that allow programming of (as yet untyped) generic traversals. Stratego natively supports ATerms. It is used extensively in combination with the SDF components of the ASF+SDF Meta-Environment.

6.4.3 Future Work

Concrete syntax and subtree sharing Currently, JJForester only supports processing of *abstract* syntax trees. Though the parser SGLR emits full *concrete* parse trees, these are imploded before being consumed by JJForester. For many program transformation problems it is desirable, if not essential, to process concrete syntax trees. A prime example is software renovation, which requires preservation of layout and comments in the source code. The ASF+SDF Meta-Environment supports processing of concrete syntax trees. In order to broaden JJForester's applicability, and to ensure its smooth interoperation with components developed in ASF, we consider adding concrete syntax support.

When concrete syntax is supported, the trees to be processed are significantly larger. To cope with such trees, the ASF+SDF Meta-Environment uses the ATerm library which implements maximal subtree sharing. As a Java implementation of the ATerm library is available, subtree sharing support could be added to JJForester. We would like to investigate the repercussions of such a change to tree representation for the expressiveness and performance of JJForester.

Decoration and aspect-orientation Adding a Decoration field to all generated classes would make it possible to store intermediate results inside the object structure inbetween visits. This way, a first visitor could calculate some data and store it in thee object structure, and then a second visitor could "harvest" these data and perform some additional calculation on them.

More generally, we would like to experiment with aspect-oriented techniques [KL+97] to customize or adapt generated code. Adding decoration fields to generated classes would be an instance of such customization.

Object-oriented folds and strategies As pointed out in Sections 6.2.5 and 6.3.3, not all transformation scenarios are elegantly expressible with our generated visitors. A possible remedy would be to generate additional instances of the visitor class for specific purposes. In particular, visitors for unparsing, pretty-printing, and equality checking could be generated. Also, the generated visitors could offer

additional refinable methods, such as `visitBefore` and `visitAfter`. Another option is to generate iterating visitors as well as non-iterating ones. Several of these possibilities have been explored in the context of the related systems discussed above. Instead of the visitor class, an object-oriented variation on updatable many-sorted folds could be generated. The main difference with the visitor pattern would be that the arguments of visit functions are not (only) the current node, but its children, and only a bottom-up accept method would be available. More experience is needed to establish which of these options would best suit our application domains.

The Visitor pattern, both in the variant offered by JJForester, where iteration is in the accept methods, and in the more common variant where iteration is in the visit methods, is severely limited in the amount of *control* that the user has over traversal behaviour. Generation of classes and methods to support folding would enrich the traversal repertoire, but only in a limited way. To obtain *full* control over traversal behaviour, we intend to transpose concepts from *strategic rewriting*, as embodied by Stratego and the rewriting calculus [CK99], to the object-oriented setting. In a nutshell the approach comes down to the following. Instead of doing iteration either in visit or accept methods, iteration would be done in neither. Instead, a small set of traversal combinators can be generated for each grammar, in the form of well-chosen refinements of the Visitor class. These traversal combinators would be direct translations of the strategy combinators in the aforementioned rewriting languages. For instance, the sequence combinator $a; b$ can be modelled as a visitor with two fields of type Visitor, and visit methods that apply these two argument visitors one after another. Using such combinators, the programmer can *program* generic traversal strategies instead of merely selecting one from a fixed set. As an additional benefit, such combinators would remove the need for multiple inheritance for combining visitors. We intend to broaden JJForester's generation scheme to generate traversal combinators, and to explore programming techniques with these.

Availability JJForester is free software, distributed as open source under the GPL license. It can be downloaded from http://www.jjforester.org.

Acknowledgements We would like to thank Arie van Deursen for his earlier work on building visitors for structures derived from SDF, and the discussions about this work. Ralf Lämmel and Paul Klint provided us with useful comments on a draft version.

Chapter 7

Legacy to the Extreme

In this chapter the relation of the techniques developed earlier in this thesis with a software development process called "Extreme Programming" is examined. Extreme programming is billed as a development method, i.e., a method to develop *new* software systems. This chapter examines the feasibility of applying this method to the maintenance of legacy systems.[1]

7.1 Introduction

In this paper, we explore the relationship between legacy systems and extreme programming. We explain how the use of (reverse engineering) tools can help to reduce the cost of change in a legacy setting, and illustrate the use of these tools. Subsequently, we discuss how and which XP practices can be incorporated into the maintenance of legacy software systems, and we analyze how and why the positive effects for regular and legacy XP projects are different. We conclude with an episode in which a pair of XP programmers face the task of changing hostile Cobol code (examples included), and are able to do so thanks to their tools and bag of XP practices.

One of the key elements of *extreme programming* (XP) is *design for today*, so that the system is equally prepared to *go any direction tomorrow*. As Beck argues, one of the reasons XP gets away with this minimalist approach because it exploits the advances in software engineering technology, such as relational databases, modular programming, and information hiding, which all help to reduce the cost of changing software [Bec99]. The result of this is that the software

[1]This chapter was published earlier as: A. van Deursen, T. Kuipers, and L. Moonen. Legacy to the extreme. In M. Marchesi and G. Succi, editors, *eXtreme Programming Examined*. Addison-Wesley, Reading, Massachusetts, May 2001.

developer does not need to worry about future changes: the change-cost curve is no longer exponential, but linear. Making changes easily is further supported by XP in various ways:

- Releases are small and frequent, keeping changes small as well;
- The code gets refactored every release, keeping it concise and adaptable;
- Testing is at the heart of XP, ensuring that refactored code behaves as it should.

The assumption that the system under construction is easily modifiable rules out an overwhelming amount of existing software: the so-called *legacy systems*, which by definition *resist change* [BS95]. Such systems are written using technology such as Cobol, IMS or PL/I, which does not permit easy modification. (As an example, we have encountered a 130,000 lines of code Cobol system containing 13,000 go-to statements.) Moreover, their internal structure has degraded after repeated maintenance, resulting in systems consisting of duplicated (but slightly modified) code, dead code, support for obsolete features, and so on. The extreme solution that comes to mind is to throw such systems away — unfortunately, it takes time to construct the new system, during which the legacy system will have to be maintained and modified.

Now what if an extreme programmer were to maintain such a legacy system? (Which probably means he was either forced to do so or seduced by an extreme salary). Should he drop all XP practices because the legacy system resists change? We will try to demonstrate in this paper why he should not. He could write test cases for the programs he has to modify, run the tests before modification, refactor his code after modification, argue for small releases, ask for end-user stories, and so on, practices that are all at the heart of XP.

7.2 Tools Make It Possible

Refactoring legacy source code is, in principle, no different from refactoring "regular" source code. Refactoring is done to improve the code, whether improving can mean many things. Modifications can be made to improve adaptability, readability, testability, or efficiency.

There are some things particular to refactoring legacy code. In order to make sure that a refactoring does not alter the functionality of the system, unit tests are run before and after the refactoring. In a legacy setting there are no unit tests beforehand, so they need to be written specifically for the refactoring.

Refactoring also requires the developers to have a great deal of detailed knowledge about the system. A good example of this is the modification of the transaction interface as described by [Bec99, Chapter 5]. This knowledge can come from

someone who knows the system intimately, or can be provided by tools that allow a developer to get to these details quickly and accurately.

Modern development systems in general provide those details. They provide all sorts of development-time and run-time information which allows hunches to be verified within seconds. Most legacy maintenance (and development) is done on a mainframe however, and the mix of JCL, Cobol and others has to be controlled without advanced development tools. Usually, even basic search tools such as *grep* are not available. The development team manages to get by, only because part of the team has been working on the system for years (ever since that mainframe was carried into the building). New team members are introduced to the system on a need-to-know basis.

More and more often, these systems get "outsourced", (the development team is sold off to another company, and the maintenance of the system is then hired from this new company). After such an outsourcing the original development/maintenance team usually falls apart, and knowledge of the system is lost. And it is still running on that same mainframe, without *grep*.

Consequently, maintenance on these systems will be of the break-down variety. Only when things get really bad, someone will don his survival suit and venture inside the source code of the system, hoping to fix the worst of the problems. This is the state most administrative systems in the world are in [BL76].

7.2.1 The Legacy Maintenance Toolbox

We have been developing a toolset over the last few years which integrates a number of results from the areas of reverse engineering and compiler construction. The toolset is the Legacy Maintenance Toolbox (LMT). It consists of a number of loosely coupled components. One of the components is DocGen [DK99a], (so called because of its basic ability to generate documentation from the source code). DocGen generates interactive, hyperlinked documentation about legacy systems. The documentation is interactive in that it combines various views of the system, and different hierarchies, and combines those with a code browser (see Figure 7.1 for an example session). DocGen shows call graphs for the whole system, but also per program. It shows database access, and can visualize data dependencies between different programs. Here, we try to provide the programmer with as much information as we can possibly get from the source. (One of the problems is that the source may be written in a vendor specific dialect of a more conventional language, of which no definition is published.)

We augment the DocGen code browsing facility with TypeExplorer [DM99], a system which infers types for variables in an untyped language (typically Cobol), and lets the programmer browse the code using those types. TypeExplorer can be used, for instance, to aid in impact analysis sessions. When the requirements of a financial system change from "make sure all amounts are British Pounds" to

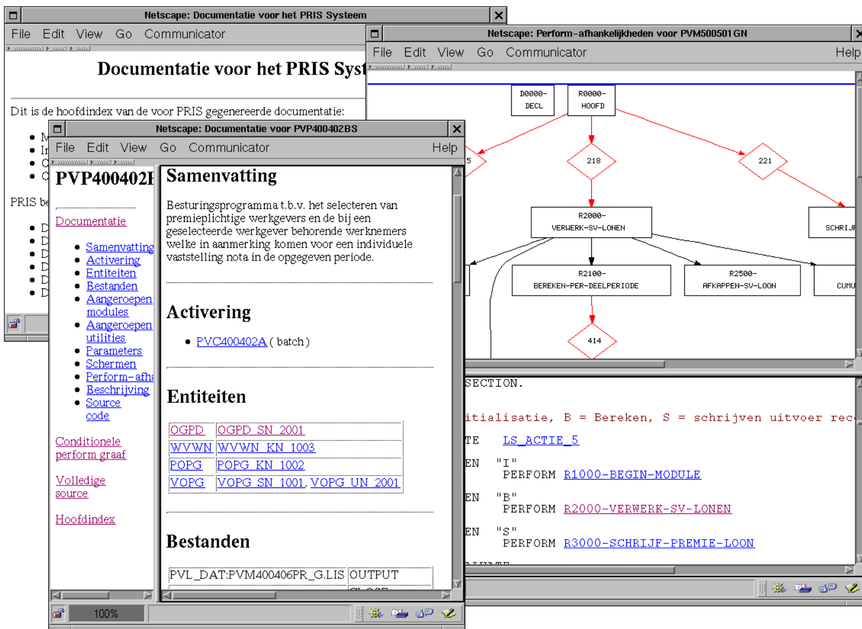


Figure 7.1: An example DocGen session.

"make sure all amounts are Euros" this will inevitably have an impact on all data (both variables in the code, and data in databases) which are of type "amount". Because TypeExplorer can come up with a list of all variables that are in the same *type equivalence class*, the programmer only has to identify a single variable which deals with amounts to identify all variables in the same type equivalence class (and therefore also dealing with amounts).

The combination of DocGen and TypeExplorer proves to be a powerful tool for gaining insight into the details of a system. DocGen and TypeExplorer get their information from a repository, which is filled by a combination of parsing (if we have a grammar for the legacy system's language) and lexical analysis. This repository can also be queried directly, using standard SQL queries.

One of the key properties of LMT is that it is *open*: external tools can be easily integrated. An example tool is CloneDr, from Semantic Designs, which detects (near) *clones* (or "copy-paste code") in sources, and removes them (by replacing them with a single procedure and a number of calls to that procedure, for instance) [BYM+98]. Code clone removal can be seen as an automated refactoring operation that adheres to the XP principle of *say it once and only once*. Apart from the obvious benefit of reducing the amount of code to be understood, a less expected benefit comes from having to give a name to the newly created procedure. This obviously is a human activity, and helps to focus the thoughts of a maintainer on a particular piece of code, which, since it was duplicated in the original program, must be of some use...

Newer (less developed) components of LMT are *concept analysis*, which aids in the remodularization of legacy systems [DK99b, KM00], and *data flow analysis* which aids in tracking data through the system [Moo97].

Using LMT, maintenance programmers can learn about the legacy system. They gain confidence about their knowledge by verifying that for instance a database table is only written to and never read from, and therefore it can be removed. They can see that two variables do not occur in the same type equivalence class, so values of the first variable never get passed to the second, and so on. As they use LMT initially to hunt down specific problems, they automatically increase their knowledge of the system, much like they would have when they were brought in during the development of the system.

7.2.2 More Tools

LMT is the result of research in the area of reverse engineering and program understanding, and builds upon related work in those areas (see [DK99b, DM99] for a detailed comparison). Two tools that are similar in nature to LMT are Rigi [WTMS95] and PBS [BHB99], which also can extract various pieces of data from the sources, and which can present them in various ways. Rigi and PBS have been used more for C then for Cobol, which involves significant differences (for exam-

ple, the lack of types and a parameter mechanism in Cobol, and the data-intensive nature of typical Cobol systems). On the commercial side, related Cobol tools are Viasoft's Existing Systems Workbench, Reasoning's InstantQA tools, and McCabe's testing and understanding tools. These tools tend to be closed, making it not only difficult to integrate them with other tools, but also to deal with customer or application-specific issues (think of dialects, coding conventions, I/O utilities, and so on), which occur very frequently in Cobol applications. Outside the Cobol arena there are various tools to analyze C, C++, or Java code, such as TakeFive's Sniff+ tools.

7.3 Adopting XP Step by Step

Adopting the XP-approach in a legacy setting can only mean one thing: aim at simplicity. How does this affect us when we decide to introduce XP in an existing legacy maintenance project?

First of all, we have to get a picture of the existing code base. This means that we generate online, hyper-linked documentation, using the DocGen technology discussed in the previous section. This allows us to browse through the legacy system, and to ask queries about the usage of programs, copybooks (the Cobol variant of an included source file), databases, and so on. Moreover, it can be regenerated after any modification, thus ensuring up-to-dateness and consistency.

Next, we have to get into contact with the end-user. We need to collect end-user stories for modification requests. Given the current state of the system, such modification requests are likely to include technical requests as well, such as increasing the stability of the system.

Then we have to divide the modification stories into small iterations. For each modification, we identify the affected code, and estimate the effort needed to implement the request. Observe that such an "impact analysis" can only be done with some understanding of the code, which is provided by the TypeExplorer technology presented in the previous section. As in regular XP, the effort estimates are made by the developers, whereas the prioritization (which story first) is done by the end-user.

We then start working release by release. Each release goes through a series of steps:

- We write test cases for the code that is to be affected by the change request, and run the tests.

- We refactor the affected code so that we can work with it, using the reverse engineering tools described earlier. This means removing extreme ugliness, duplicated functionality, unnecessary data, copy-paste clones, standardizing the layout, and so on. We then re-run the test cases just constructed, in order to make sure that no damage has been done while refactoring.

- After that, the code is in such a shape that we feel sufficiently confident that we can modify it. If necessary, we adapt the test cases to reflect the modified features, implement the modification request, and re-run the test cases.
- Finally we refactor again, re-test, and re-generate the system documentation.

For XP-programmers these steps will sound extremely familiar. So what are the differences with regular XP?

First of all, the productivity per iteration is lower than in regular XP. This is because (1) there are no test cases, which will have to be added for each refactoring and modification; (2) the code has not been previously refactored; and (3) the programming technology used is inherently more static than, for example, Smalltalk.

Second, the code base itself is not in its simplest state. This means that program understanding, which constitutes the largest part of actually changing a program, will take much more time. Luckily, XP-programmers work in pairs, so that they can help each other in interpreting the code and the results from invoking their tool set. The code not being in its simplest state also means that while studying code (during impact analysis, for example), the pair is likely to identify many potential ways of refactoring, for example when encountering dead code.

One might consider doing a one-shot, up front refactoring of the entire legacy system to avoid such problems. However, successful refactoring is not an automatic process but requires human intervention. Moreover, there are no test cases available a priori. Last but not least, a total refactoring may be unnecessary anyway if parts of the system do not need modification or are likely to be removed (simplicity requires us not to worry about things we are not going to need).

Another observation is that in normal XP the positive effects of refactoring are accumulated – keeping the system flexible at all times. When applying XP to a legacy system, only after starting to follow XP principles parts of the system get refactored. The accumulated effect of this is much lower than in regular XP.

A final question to ask is whether the scenario sketched is realistic. If it is so good, why has it not been done before? Reasons may be a lack of awareness of the XP-opportunities, fear of the overwhelming amount of legacy code leading to paralysis, confusion with the expensive and unrealistic one-shot refactoring approach, or the plain refusal to invest in building test cases or refactoring. The most important reason, however, is that it is only during the last few years that reverse engineering technology has become sufficiently mature to support the XP approach sketched above. Such technology is needed to assist in the understanding needed during planning and modification, and to improve existing code just before and after implementing the modification.

deze strook niet meezenden
AFLOSSING ILS
TERM.APR
6002727993
000431
100 00
Corresp.nr. :
Vervaldatum:
30-04-2000
op rekening
19.23.21.900
Informatie Beheer Groep
Groningen

acceptgiro
over te schrijven/te storten
100 gld 00 ct
van girorekening of van bankrekening
470477288
van/door
naam T KUIPERS
adres
plaats
op rekening 19.23.21.900
tgv Informatie Beheer Groep
Postbus 50101 - 9702 GA Groningen

AFLOSSING LANGLOPENDE SCHULD (ILS)
CORRESPONDENTIENR. :
BETREFT TERMIJN(EN): APR
VERVALDATUM: 30-04-2000
6002 7279 9300 0431+
handtekening
zijn alle rode rubrieken ingevuld?
de ruimte hieronder niet beschrijven
formulier met blauwe of zwarte inkt invullen
gezamenlijke banken en spaarbanken postbank

betalingskenmerk | van rekening | gld | ct | diversen | naar rekening | code

6002727993000431+ 470477288< 000100001+ 192321900+33>
voor gebruiksaanwijzing z.o.z.

Figure 7.2: An example accept-giro

7.4 XP on Legacy Code

So how would all of this benefit a bunch of maintenance programmers facing a mountain of Cobol code? We will try to answer that question by describing a concrete step-by-step maintenance operation. The example is from the invoicing system of a large administrative system (from the banking/insurance world). All code used in this example is real. We have changed it slightly as to camouflage actual amounts and account numbers.

But first, some culture: In the Netherlands, bill paying is largely automated with companies sending out standardized, optically readable forms called "accept-giro" (see Figure 7.2 for an example). Normally, all information including the customers account number, the companies account number and the amount to be paid, is preprinted on these accept-giros, and all the customer has to do is sign them and send them back to accept the mentioned amount being charged off his account. These forms then are read automatically by a central computer operated by all associated Dutch banks, and the appropriate amount is transfered from one account to the next, even between different banks.

The task at hand for the programmers is: we have changed banks/account numbers, and all invoices printed from next month should reflect that. That is, all bills should be paid to our new account number.

Once the team understands the task, they start to work. First they need to find out what file is being printed on the blank forms. They know that their system only creates files, and that these files are then dumped to a specialized high volume printer somewhere. After asking around and looking at the print job descriptions, it turns out that all data for the invoices is in a file called INVOP01. As INVOP01

```
xxxx yyyy
zzzz wwww                  100   00     xxxx yyyy zzzz wwww

   100 00

                      TESTNAME T
                      TESTST 12
                      9999 XX TESTCITY

555.12.12              555.12.12
LARGE                  LARGE CORP.
CORP.                  AMSTERDAM


X               xxxxyyyyzzzzwwww+       10000x+5551212+37>
```

Figure 7.3: INVOP01 from the initial system

is the end product of this particular task, the team runs the system on their test data and keep a copy of the resulting INVOP01 file. Now they know that when they are finished with the task, the INVOP01 file they generate should be the same as the current INVOP01, apart from the account number. The INVOP01 on the test data can be seen in Figure 7.3.

Because they have the system analysis tools described earlier, the team can now check what programs do something with the INVOP01 print file. Figure 7.4 shows all facts that have been derived for INVOP01.

It turns out that the only program operating on INVOP01 is INVOMA2. The information derived from INVOMA2 shows that this program only uses one input file: INVOI01. Executing the system on the test data reveals that INVOI01 does not contain account numbers, rather it contains the names and addresses of customers. If the account number is not read from file, and INVOMA2 does not access any databases, then the account number should be in the code! The team does a find on the string 5551212 in the code, and they find the code as shown in Figure 7.5.

(Note the old account number commented out in the three lines starting with an asterisk ...) They change the account number to 1212555 (the new account number) and run the system using the test data. Much to their surprise, the test version of INVOP01 comes out like shown in Figure 7.6.

The last line of the test file shows the correct account number, together with the `+37>` that is also visible in the code. This is the part of the form that will be read optically. However, the part of the form that is meant for humans still shows the *old* account number. The team look at each other, shake their heads, and do a find on `555.12.12` in the source code. What shows up is in Figure 7.7.

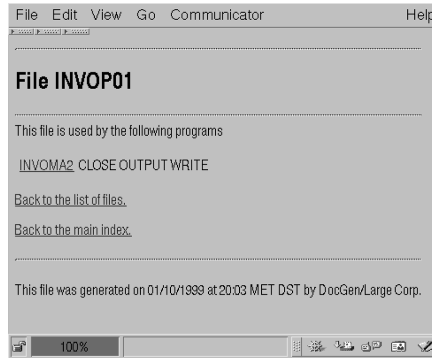


Figure 7.4: All facts derived for file INVOP01

```
  05  P009-BEDRAG-CONTR PIC 9.
  05                    PIC X     VALUE "+".
  05                    PIC X(10) VALUE SPACE.
  05            .
*     07                PIC 9(8)  VALUE 8765432.
*     07                PIC X     VALUE "+".
* 05                    PIC X(4)  VALUE " 37>".
      07                PIC 9(09) VALUE 5551212.
  05                    PIC X(4)  VALUE "+37>".
```

Figure 7.5: Code found by searching for 5551212

```
xxxx yyyy
zzzz wwww               100   00      xxxx yyyy zzzz wwww

   100  00

                    TESTNAME T
                    TESTST 12
                    9999 XX TESTCITY

555.12.12            555.12.12
LARGE                LARGE CORP.
CORP.                AMSTERDAM


X               xxxxyyyyzzzzwwww+       10000x+1215555+37>
```

Figure 7.6: Test version of INVOP01 after modification

```
03  P008A.
    05            PIC X(19)     VALUE SPACE.
    05            PIC X(09)     VALUE "555.12.12".

03  P008B.
    05            PIC X(09)     VALUE "555.12.12".
    05            PIC X(07)     VALUE SPACE.
    05            PIC X(11)
                  VALUE "LARGE CORP.".
```

Figure 7.7: Code found by searching for 555.12.12

They change the two(!) account numbers and run the test again. Now everything comes out as expected. They write a todo item that this part of the code needs urgent refactoring, or maybe they immediately implement a procedure that formats account numbers. Or maybe the system can be left to die and they can spend their time on an XP reimplementation of the whole system.

7.5 Conclusions

In this paper we have looked at extreme programming from the viewpoint of legacy systems. We observed that the programming environment used for regular XP projects provides capabilities not available for most mainframe-based legacy systems. At the same time we described progress in the area of reverse engineering tools that can be used to overcome these limitations. We used these findings to come up with a way to adopt XP practices during legacy system software maintenance.

Chapter 8

Conclusions

In the introduction to this thesis, three research questions have been posed. They were:

- How can we reduce the search space when searching for a particular artifact in a legacy software system?
- How can we obtain sufficiently detailed information about a legacy software system to perform a structural change (semi-)automatically?
- How can we alter the software engineering process such that we no longer produce legacy systems?

8.1 Reduce Search Space

In Chapter 2, it was shown how we can use a number of analysis techniques, lexical analysis being the prime one, to derive facts from a legacy system. A tool architecture for performing such analyses was described. Using these techniques, a call-dependency analysis, or a data-usage analysis can be made of a system. The results from such an analysis reduce the search space of someone looking for a particular artifact because, for instance, they are looking for a module that modifies a particular piece of data.

Chapter 3 builds on those results by integrating all analyses in a compact, browseable presentation format. This way, all analysis results are readily available, and are all linked to each other. An engineer can move easily from an overview of all modules in a system, to an overview of all modules that operate on a piece of data, to an overview of all modules that actually modify that piece of data, and so on. Building these documentation generators gets facilitated by so-called island grammars, an analysis technique which borrows from lexical analysis its flexibility, and from syntactic analysis its thoroughness.

8.2 Information for Structural Change

Since legacy systems are usually very large, the amount of facts extracted from them is usually large too. Somehow, this information needs to be filtered, preferably automatically. One way of automatically filtering is to try to derive an object-oriented design from an existing, non-object-oriented system. Once this design is derived, the actual structural change can be performed manually. Chapter 4 examines two methods that can be used for the automatic derivation of objects. These methods use exactly the facts that can be derived using the techniques developed in the earlier chapters. Although deriving an object-oriented design completely automatically from a legacy system turns out to be undesirable, if not impossible, the techniques examined in this chapter do support the human engineer in automatically relating data and procedures. Having this information available is a very good starting point for an actual object-oriented redesign.

Another way to interpret the derived facts is by automatically grouping the data in a system by the concept they are related to. That is, to group data of the same *type*. Chapter 5 uses a technique called type-inference to infer the type of a variable in a system. Using type inference, a variable can be determined to hold either a date, or a monetary value, or a social-security number, etcetera. Using the concept analysis from Chapter 5 on programs and the inferred type of the variables they use appears to give a better starting point for an object-oriented redesign than using just the variables.

When we are looking for detailed information, even syntactical analysis in itself will not, in general be enough. Usually, a number of syntactical analyses need to be performed to get the right fact out of the system. More often than not, these analyses have a conditional dependency, and they use each others results as input. Having a system that allows for the elegant and concise development of such combinations of analyses is the topic of Chapter 6. In it, a form of parse tree traversal is developed that can be used to perform exactly the kinds of interdependent syntactical analyses as described.

8.3 Altering the Software Engineering Process

New insights have improved the software engineering process over the last decades. These improvements are usually only applied to software development (the building of new software systems), not to software maintenance. In Chapter 7 the relation of the techniques developed earlier in this thesis to a software development process called “Extreme Programming” is examined. Furthermore, the feasibility of applying this process to the maintenance of legacy systems is discussed. Using this, or a similar methodology, and supported by the right analysis techniques, a team of engineers can maintain a legacy system in such a way that it stops being a legacy system, and becomes an understandable, modifiable, and maintainable system, once again.

Bibliography

[AKW88] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[B$^+$00] M.G.J. van den Brand et al. The ASF+SDF Meta-Environment: a component-based language development environment. Submitted for publication, 2000.

[Bec99] K. Beck. *Extreme Programming Explained. Embrace Change*. Addison Wesley, 1999.

[BHB99] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *21st International Conference on Software Engineering, ICSE-99*, pages 555–563. ACM, 1999.

[BHK89] J. A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism asf. In *Algebraic Specification*, chapter 1, pages 1–66. The ACM Press in coöperation with Addison-Wesley, 1989.

[BJKO00] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[BK94] J. A. Bergstra and P. Klint. The ToolBus: a component interconnection architecture. Technical Report P9408, University of Amsterdam, Programming Research Group, 1994. Available from http://www.science.uva.nl/research/prog/reports/reports.html.

[BK98] J. A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.

[BL76] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[BM97] E. Burd and M. Munro. Enriching program comprehension for software reuse. In *International Workshop on Program Comprehension; IWCP'97*. IEEE Computer Society, 1997.

[BMW96] E. Burd, M. Munro, and C. Wezeman. Extracting reusable modules from legacy code: Considering the issues of module granularity. In *3rd Working Conference on Reverse Engineering; WCRE'96*, pages 189–196. IEEE Computer Society, 1996.

[Boe88] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5), 1988.

[dB+96] M. G. J. van den Brand et al. Industrial applications of ASF+SDF. In *Algebraic Methodology and Software Technology (AMAST'96)*, volume 1101 of *LNCS*, pages 9–18. Springer-Verlag, 1996.

[dBKV96] M. G. J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K.G. Jeffery, J. Král, and M. Bartošek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *LNCS*, pages 235–255. Springer-Verlag, 1996.

[dBSV97a] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In *Fourth Working Conference on Reverse Engineering; WCRE'97*, pages 144–155. IEEE Computer Society, 1997.

[dBSV97b] M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Obtaining a Cobol grammmar from legacy code for reengineering purposes. In *Proceedings of the 2nd international workshop on the theory and practice of algebraic specifications*, Electronic workshops in computing. Springer Verlag, 1997.

[dBSV98] M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Proceedings of the sixth International Workshop on Program Comprehension*, pages 108–117. IEEE, 1998.

[Bro83] R. Brooks. Towards a theory of the comprehension of computer programs. *Int. 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.

[BV95] J. C. M. Baeten and C. Verhoef. Concrete process algebra. In *Handbook of Logic in Computer Science*, volume 4, pages 149–268. Clarendon Press, Oxford, 1995.

[BYM+98] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance, ICSM'98*, pages 368–377. IEEE Computer Society Press, 1998.

[CB91] G. Caldiera and V. R. Basili. Identifying and qualifying reusable software components. *IEEE Computer*, pages 61–70, February 1991.

[CC90] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[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.

[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, *Int. Conf. on Software Maintenance; ICSM 95*, pages 66–75. IEEE Computer Society, 1995.

[CK99] Horatiu Cirstea and Claude Kirchner. Introduction to the rewriting calculus. Rapport de recherche 3818, INRIA, December 1999.

[CV95] A. Cimitile and G. Visaggio. Software salvaging and the call dominance tree. *Journal of Systems Software*, 28:117–127, 1995.

[DDF+97] A. De Lucia, G. A. Di Lucca, A. R. Fasolino, P. Guerra, and S. Petruzzelli. Migrating legacy systems towards object-oriented platforms. In *International Conference on Software Maintenance; ICSM'97*, pages 122–129. IEEE Computer Society, 1997.

[DK97] A. van Deursen and T. Kuipers. Finding classes in legacy code using cluster analysis. In S. Demeyer and H. Gall, editors, *Proceedings of the ESEC/FSE'97 Workshop on Object-Oriented Reengineering*. Report TUV-1841-97-10, Technical University of Vienna, 1997.

[DK98] A. van Deursen and T. Kuipers. Rapid system understanding: Two COBOL case studies. In S. Tilley and G. Visaggio, editors, *Sixth International Workshop on Program Comprehension; IWPC'98*, pages 90–98. IEEE Computer Society, 1998. Also published as Chapter 2 of this thesis.

[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.

[DK98] A. van Deursen and P. Klint. Little languages, little maintenance? *Journal of Software Maintenance*, 10(2):75–92, 1998.

[DK99a] A. van Deursen and T. Kuipers. Building documentation generators. In *International Conference on Software Maintenance, ICSM'99*, pages 40–49. IEEE Computer Society, 1999. Also published as Chapter 3 of this thesis.

[DK99b] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *21st International Conference on Software Engineering, ICSE-99*, pages 246–255. ACM, 1999. Also published as Chapter 4 of this thesis.

[DM98] A. van Deursen and L. Moonen. Type inference for COBOL systems. In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proc. 5th Working Conf. on Reverse Engineering*, pages 220–230. IEEE Computer Society, 1998.

[DM99] A. van Deursen and L. Moonen. Understanding COBOL systems using types. In *Proceedings 7th Int. Workshop on Program Comprehension, IWPC'99*, pages 74–83. IEEE Computer Society, 1999.

[dOBvSdPL98] Ch. de Oliveira Braga, A. von Staa, and J. C. S. do Prado Leite. Documentu: A flexible architecture for documentation production based on a reverse-engineering strategy. *Journal of Software Maintenance*, 10:279–303, 1998.

[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 *26th Annual Symposium on Principles of Programming Languages, POPL'99*. ACM, 1999. To appear.

[FGK88] E. B. Fowlkes, R. Gnanadesikan, and J. R. Kettenring. Variable selection in clustering. *Journal of Classification*, 5:205–228, 1988.

[FHK+97] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K.Kontogiannis, H. A. Müller, J. Mylopoulos, and S. G. Perelgut. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

[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 *Proc. Conf. 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 *Fourth Workshop on Program Comprehension; IWPC'96*. IEEE Computer Society, 1996.

[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.

[GW99] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.

[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.

[JV00] M. de Jonge and Joost Visser. Grammars as contracts. In *Generative and Component-based Software Engineering (GCSE)*, Erfurt, Germany, October 2000. CD-ROM Proceedings. To be published in Lecture Notes in Computer Science (LNCS), Springer.

[JVV00] M. de Jonge, Eelco Visser, and Joost Visser. XT: a bundle of program transformation tools. *Submitted for publication*, 2000.

[KC99] R. Kazman and J. Carrière. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6:107–138, 1999.

[KL+97] Gregor Kiczales, John Lamping, et al. Aspect-oriented programming. In *Proceedings of ECOOP'97*, number 1241 in LNCS. Springer Verlag, 1997.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[KLV00] Jan Kort, Ralf Lämmel, and Joost Visser. Functional transformation systems. In *9th International Workshop on Functional and Logic Programming*, Benicassim, Spain, September 2000.

[KM00] T. Kuipers and L. Moonen. Types and concept analysis for legacy systems. In *Proceedings of the International Workshop on Programming Comprehension (IWPC 2000)*. IEEE Computer Society, June 2000. Also published as Chapter 5 of this thesis.

[Knu84] D.E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[KR90] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990.

[Lak97] A. Lakhotia. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software*, pages 211–231, March 1997.

[LHGF88] L. D. Landis, P. M. Hyland, A. L. Gilbert, and A. J. Fine. Documentation in a software maintenance environment. In *Proc. Conference on Software Maintenance*, pages 66–73. IEEE Computer Society, 1988.

[LS97] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *19th International Conference on Software Engineering, ICSE-19*, pages 349–359. ACM, 1997.

[LVK00] Ralf Lämmel, Joost Visser, and Jan Kort. Dealing with large bananas. In Johan Jeuring, editor, *Workshop on Generic Programming*, Ponte de Lima, July 2000. Technical Report, Universiteit Utrecht.

[LW90] S. S. Liu and N. Wilde. Identifying objects in a conventional procedural language: An example of data design recovery. In *International Conference on Software Maintenance; ICSM'90*, pages 266–271. IEEE Computer Society, 1990.

[MV96] A. von Mayrhauser and A. M. Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6):424–438, 1996.

[MV97] A. von Mayrhauser and A. M. Vans. Hypothesis-driven understanding processes during corrective maintenance of large scale software. In *International Conference on Software Maintenance; ICSM'97*, pages 12–20. IEEE Computer Society, 1997.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[Mic96] Micro Focus Revolve user guide. Burl Software Laboratories Inc., USA, 1996.

[MN96] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering 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, *Theory and Practice of Algebraic Specifications; ASF+SDF'97*, Electronic Workshops in Computing, Amsterdam, September 1997. Springer-Verlag.

[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*, 5(4):181–204, 1993.

[Nei96] J. M. Neighbors. Finding reusable software components in large systems. In *3rd Working Conference on Reverse Engineering; WCRE'96*, pages 2–10. IEEE Computer Society, 1996.

[NK95] P. Newcomb and G. Kottik. Reengineering procedural into object-oriented systems. In *Second Working Conference on Reverse Engineering; WCRE'95*, pages 237–249. IEEE Computer Society, 1995.

[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.

[Pig97] T. M. Pigoski. *Practical Software Maintenance – Best Practices for Managing Your Software Investment*. John Wiley and Sons, 1997.

[PXL95] Jens Palsberg, Cun Xiao, and Karl Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, March 1995.

[Raj97] Vaclav Rajlich. Incremental redocumentation with hypertext. In *1st Euromicro Working Conference on Software Maintenance and Reengineering CSMR 97*. IEEE Computer Society Press, 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 *26th Annual Symposium on Principles of Programming Languages, POPL'99*. ACM, 1999. To appear.

[Sch91] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *13th International Conference on Software Engineering, ICSE-13*, pages 83–92. ACM, 1991.

[SCHC99] S. E. Sim, C. L. A. Clarke, R. C. Holt, and A. M. Cox. Browsing and searching software architectures. In *Int. Conf. on Software Maintenance, ICSM'99*, pages 381–390. IEEE Computer Society, 1999.

[SH98] S. E. Sim and R. C. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *20th Int. Conf. on Software Engineering; ICSE-98*, pages 361–370. ACM, 1998.

[SN95] H. M. Sneed and E. Nyáry. Extracting object-oriented specification from procedurally oriented programs. In *Second Working Conference on Reverse Engineering; WCRE'95*, pages 217–226. IEEE Computer Society, 1995.

[Sne92] H. M. Sneed. Migration of procedurally oriented COBOL programs in an object-oriented architecture. In *International Conference on Software Maintenance; ICSM'92*, pages 105–116. IEEE Computer Society, 1992.

[Sne96] G. Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–189, April 1996.

[Sne98] G. Snelting. Concept analysis — a new framework for program understanding. In *Proceedings of the 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, 2000. To appear.

[SR97] M. Siff and T. Reps. Identifying modules via concept analysis. In *International Conference on Software Maintenance, ICSM97*. IEEE Computer Society, 1997.

[ST98] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *Foundations of Software Engineering, FSE-6*, pages 99–110. ACM, 1998. SIGSOFT Software Engineering Notes 23(6).

[Swa76] E. B. Swanson. The dimensions of maintenance. In *Proceedings of the Second International Conference on Software Engineering*, pages 492–497. ACM, 1976.

[TL95] H. B. K. Tan and T. W. Ling. Recovery of object-oriented design from existing data-intensive business programs. *Information and Software Technology*, 37(2):67–77, 1995.

[Tom85] M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.

[V+] E. Visser et al. The online survey of program transformation. http://www.program-transformation.org/survey.html.

[VBT99] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).

[Vis97a] E. Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, University of Amsterdam, Programming Research Group, 1997.

[Vis97b] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[WAKS97] D. C. Wang, A. W. Appel, J. L. Korn, and C. S. Serra. The Zephyr abstract syntax description language. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 213–28, Berkeley, CA, October 15–17 1997. USENIX Association.

[WBF97] T. Wiggerts, H. Bosma, and E. Fielt. Scenarios for the identification of objects in legacy systems. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *4th Working Conference on Reverse Engineering*, pages 24–32. IEEE Computer Society, 1997.

[Wig97] T. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *4th Working Conference on Reverse Engineering*, pages 33–43. IEEE Computer Society, 1997.

[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.

[YRK99] R. J. Yarger, G. Reese, and T. King. *MySQL & mSQL*. O'Reilly, 1999. http://www.mysql.org/.

Summary

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

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

The results of the analyses described above can be consolidated by building a documentation generator. This thesis presents the notion of a documentation generator as a system that performs any number of more or less detailed analyses on a legacy software system, and presents the results in an interlinked way, with different levels of abstraction. For some of the more detailed analyses that are performed as part of a documentation generator, lexical analysis as used during rapid system understanding is not powerful enough, but full syntactic analysis may

be too restrictive, or too expensive. This thesis introduces the notion of island grammars, from which an analysis can be generated that is liberal when analyzing the larger part of the source code (the “water”), but very strict when analyzing the relevant part (the “island”).

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

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

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

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

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

Extreme programming is a software engineering methodology that bundles a number of programming practices and tries to make their interdependency explicit.

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

Samenvatting

Deze dissertatie laat een aantal experimentele technieken zien om *legacy* software systemen te kunnen begrijpen. Software systemen die gebruikt worden, moeten worden onderhouden. Het onderhouden van een complex software systeem is een intimiderende taak. De onderhoudsactiviteit bestaat uit het aanpassen van de broncode van het systeem. Een groot gedeelte van die activiteit is het vinden van de exacte locatie van het voorwerp dat veranderd moet worden in de broncode. De veranderingen die moeten worden uitgevoerd kunnen verschillen van het uitvoeren van een kleine verandering, bijvoorbeeld het repareren van een kleine fout, tot structurele veranderingen die nodig zijn omdat het ontwerp van het systeem is veranderd. Voor de verschillende soorten wijzigingen zijn verschillende soorten informatie nodig.

Een manier om zulke informatie te verkrijgen is door "snel systeem begrip". Lexicale analyse wordt dan gebruikt om een aantal feiten uit een Cobol *legacy* systeem te extraheren. Een van de feiten is een zogenaamde systeeminventaris, die een aantal basismetrieken over het systeem laat zien. Verder wordt er gedetailleerde informatie over databasegebruik per module afgeleid en informatie over waar welke datavelden in het systeem worden gebruikt. De relatie tussen de Cobol secties wordt geanalyseerd en gepresenteerd, zowel grafisch als tekstueel. Tot slot worden een aantal conclusies getrokken over de analyseresultaten. Door gebruik te maken van een lichtgewicht en (daardoor) niet heel precieze analysetechniek kan er een initieel antwoord worden gegeven op vragen als: "Bevat het systeem herbruikbare code?" en "Welk gedeelte van de code is platformspecifiek?". De antwoorden zijn zeker niet gedetailleerd genoeg om een structurele verandering van het systeem uit te voeren, maar ze geven genoeg informatie om te beslissen of een nadere analyse zinvol is, en geven deze informatie op een snelle en kosteneffectieve manier.

De resultaten van de hierboven beschreven analyses kunnen worden geconsolideerd door een documentatiegenerator te bouwen. In deze dissertatie is een documentatiegenerator een systeem dat een aantal min of meer gedetailleerde analyses op een *legacy* softwaresysteem uitvoert en de resultaten op een geïntegreerde manier laat zien, met verschillende abstractieniveaus. Voor sommige van de meer

gedetailleerde analyses die worden uitgevoerd als onderdeel van de documentatiegenerator is lexicale analyse niet krachtig genoeg, maar volledige syntactische analyse te beperkend, of te duur. Deze dissertatie introduceert de notie van "eilandgrammatica's", waaruit analyses gegenereerd kunnen worden die liberaal zijn wanneer ze het grootste gedeelte van de broncode analyseren (het "water"), maar zeer nauwkeurig als ze het relevante gedeelte van de broncode analyseren (het "eiland").

Een documentatiegenerator integreert handmatige documentatie met automatisch gegenereerde documentatie. De generator gebruikt grafische, *gehyperlinkte* representaties van de afhankelijkheden tussen verschillende soorten van modules, databestanden, databases enzovoorts.

Als een systeem structureel veranderd moet worden dan moet er een structuur worden opgelegd aan een grotendeels ongestructureerd, of in ieder geval niet goed genoeg gestructureerd systeem. Een manier om zo'n structuur af te dwingen is een procedureel systeem te migreren naar een object-georiënteerd systeem. Deze dissertatie vergelijkt twee technieken om de gegevens en de procedures van een procedureel systeem aan elkaar te relateren, om zo een beginpunt te vormen voor een object-georiënteerd (her-)ontwerp van dat systeem. Deze technieken zijn "clusteranalyse" en "conceptanalyse". Clusteranalyse werkt door een afstand te berekenen tussen verschillende objecten en de objecten die minder dan een zekere afstand van elkaar liggen te groeperen in een cluster. In dit geval wordt de afstand tussen verschillende datavelden in een systeem berekend, gebaseerd op het feit dat datavelden al dan niet in dezelfde module worden gebruikt. Conceptanalyse groepeert gerelateerde voorwerpen met hun kenmerken in "concepten". Een concept is een maximale deelverzameling van voorwerpen die exact dezelfde kenmerken hebben. In dit geval zijn de voorwerpen de datavelden en hun kenmerken de modules waar ze in gebruikt worden.

Conceptanalyse blijkt beter geschikt voor objectidentificatie dan clusteranalyse. Een casus demonstreert dat als bepaalde datavelden van een Cobol systeem worden gebruikt als voorwerpen (en bepaalde andere datavelden niet) de conceptanalyse van die velden en hun bijbehorende modules leidt tot een beginpunt voor een object-georiënteerd herontwerp van het originele systeem.

Legacy systemen zijn grotendeels ongetypeerd. Dit zorgt ervoor dat de systemen moeilijker zijn te onderhouden, ten dele omdat het moeilijk is de "relevante" gegevens in een systeem te vinden. Type-inferentie is een manier om dit probleem deels op te lossen. Alle datavelden in een systeem worden dan gegroepeerd in "types", gebaseerd op de manier waarop ze met elkaar interacteren. Als deze types gebruikt worden als voorwerpen in de conceptanalyse, in plaats van de datavelden, dan verbetert het resultaat van de conceptanalyse als beginpunt voor een herontwerp.

Om softwaresystemen te analyseren met de mate van detail die nodig is voor bijvoorbeeld type-inferentie moet het systeem geparseerd worden en de parseer-

bomen geanalyseerd. In deze dissertatie wordt een systeem geïntroduceerd dat het mogelijk maakt een parseerboom op object-georiënteerde wijze te benaderen. Het systeem produceert ook zogenaamde *traversals* die het mogelijk maken dat degene die de analyses schrijft zich slechts met die analyses bezig hoeft te houden. Hij kan een van de traversal-strategieën die het systeem biedt kiezen. De analyses die met behulp van dit systeem zijn geschreven laten zich zonder moeite koppelen aan een grote variëteit van toepassingen.

Extreme programming is een manier van software-ontwikkeling die een aantal programmeerpraktijken bundelt en hun wederzijdse afhankelijkheden expliciet probeert te maken. *Extreme programming* focust op veranderbaarheid van systemen. Een systeem dat is ontwikkeld met behulp van de *extreme programming* methodologie zou gereed moeten zijn voor elke verandering die de eigenaar van het systeem nodig heeft, zowel klein als structureel. Het laatste hoofdstuk van deze dissertatie onderzoekt of de technieken van *extreme programming* met terugwerkende kracht kunnen worden gebruikt bij het onderhouden van een *legacy* systeem, om zo het begrip en uiteindelijk de veranderbaarheid van zo'n systeem te verhogen.

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