Reverse Engineering for Mainframe Enterprise Applications

Patterns and Experiences

Reverse engineering voor mainframe enterprise applicaties
patronen en ervaringen

Joris Van Geet

Promotor:
Prof. Dr. Serge Demeyer

Proefschrift ingediend tot het behalen van de graad van
Doctor in de wetenschappen
Acknowledgments

Although obtaining a PhD is quite a lonely undertaking, many people deserve some gratitude for making it possible and/or bearable.

Thank you to my promotor, Serge Demeyer, for giving me the opportunity to use “industry as a lab”: I would not have started my PhD without it. Similarly, I must say thanks to Rudi Maelbrancke for inspiring and sponsoring the project and offering me KBC as a lab. I also need to thank Herman Van Durme for co-sponsoring the project and offering me AXA as lab. Since change is inevitable, I also need to thank Jo Uyttenhove (Rudi’s replacement) and Tom Samyn (Jo’s replacement), who provided me both with very valuable and helpful KBC guidance. During all this change, Tamara has always been there to help me out with practical problems at KBC.

Conducting this research in industry would not have been possible without help from “the trenches”. In particular I would like to thank Michel Geers for his wise advice and funny anecdotes; Johan Janssens and his team, who have been helpful beyond measure; Danny Backx, who taught me a lot about company politics; the TCO, TAX and BOS teams of OSIRIS for all their feedback and the entertaining lunches. Of course, I would also like to thank my colleagues from the “ivory tower”, in particular Kris De Schutter for helping me kickstart my research; and the entire AnSyMo and ex-LORE team for the copious feedback, the interesting lunch breaks and the delicious barbecues.

On a more playful note I would like to thank some colleagues at the university for the nice atmosphere. In particular Jeroen V. V., whose love of food has been a b(e)acon in an otherwise hungry world; Juan F. for inspiring calm with his Latin-American view of life; Jeroen A., whose single greatest achievement was to bring Murray to life; Gino for the nice times we had organizing scientific parties in childish universities and (nearly) blowing up car transmissions; Wim, who has been a great partner in “getting things done” and, as a consequence, an even better partner for not actually getting anything done. I would also like to thank the less known Gulielmus Puer for inspiring countless (office) games, and Michaelus Semita, Hieronymus Serius and Johannes Filius Petrae for helping us play them.

Besides my friends at work, I also need to thank Pieter for helping me stay in touch with the real world and (occasionally) with the gym; and Sven for sharing an apartment (and so much more) in the first few years: never a dull moment.

On a more familiar note I would like to thank my parents for letting me find (and go) my own way.
Finally, a **huge** thank you to Liesbeth who, apart from willingly appearing on the cover of this book, has given my life some meaning and me some lovely times; and a **tiny** thank you to Jitte, whose mere presence puts things in (the right) perspective.

Thank you all!\(^1\)

\[^1\text{Most of you can stop reading now :)}\]
Abstract

The ability to evolve and maintain software reliably is a major challenge for today’s organizations. Reverse engineering can support this challenge by recovering knowledge from existing systems. The reverse engineering community is a vibrant research community which has resulted in many useful techniques and research prototypes to recover this knowledge, however, few of them have been exploited industrially in significant ways.

To investigate this lacking industrial adoption, this dissertation explores the applicability of existing reverse engineering techniques in practice, more specifically, on mainframe systems in the financial services industry. We report our experience with applying software views, feature location and redocumentation; and we identify the main cause for the lacking adoption to be the mismatch between the characteristics of reverse engineering and the characteristics of the (development) processes in place in the organizations.

To resolve this situation, we recommend organizations to incorporate reverse engineering into their processes, urge researchers to apply their techniques not only in the lab but also in realistic circumstances, and provide practitioners with patterns that will allow them to apply reverse engineering in the mean time.
The following peer-reviewed papers have been published while working on my PhD research. They have all been incorporated into this dissertation.


Nederlandstalige samenvatting

De evolutie en het onderhoud van bestaande software systemen is een grote uitdaging voor vele bedrijven. Reverse engineering kan hierbij helpen door kennis van bestaande systemen te helpen opbouwen. Omdat reverse engineering een zeer actief onderzoeksgebied is, zijn er reeds vele technieken en prototypes gebouwd om deze kennis op te bouwen. Spijtig genoeg worden maar weinig van deze technieken ook toegepast in industrie.

Om dit te onderzoeken, bestuderen we in deze thesis de toepasbaarheid van bestaande reverse engineering technieken in de praktijk. Deze praktijk wordt in ons geval ingevuld door mainframe systemen gebruikt in de financiële wereld. We rapporteren onze ervaring met het toepassen van software views, feature lokalisatie en herdocumentatie. Tevens identificeren we de hoofdreden voor de gebrekkige adoptie van reverse engineering technieken in de praktijk, namelijk de mismatch tussen de inherente eigenschappen van reverse engineering en de eigenschappen van de ontwikkelprocessen gebruikt in deze financiële organisaties.

Als oplossing raden we organisaties aan om reverse engineering voorzien in hun processen, vragen we onderzoekers hun technieken niet enkel in het lab testen maar ook in realistische omstandigheden, en voorzien we een aantal patronen waardoor de mensen op de vloer ondertussen reeds reverse engineering kunnen toepassen.
# Contents

Acknowledgments iii  
Abstract v  
Publications vii  
Nederlandstalige samenvatting ix

## I Setting the Scene 1

1 Introduction 3  
1.1 Software Evolution 4  
1.2 Reverse Engineering 4  
1.3 Research Questions 5  
1.4 Guide to the reader 5  

2 Reverse Engineering 7  
2.1 Taxonomy 7  
2.2 Incentives 9  
2.3 Software Maintenance and Evolution 10  
2.4 Activities 11  
2.5 Comprehension 12  
2.6 Requirements 13  
2.7 Techniques vs Tools 13  
2.8 Patterns 13  
2.9 Summary 15  

3 Financial Services Industry 17  
3.1 Organizational Culture and Development Processes 17  
3.2 IT Structures 19  
3.2.1 Enterprise-wide Structures 19  
3.2.2 Business Domains 20
3.3 IT Activities ................................................................. 21
3.4 IT Roles ............................................................ 23
3.5 Enterprise Applications ........................................... 24
3.6 Mainframes .......................................................... 24
3.7 Summary .............................................................. 26

4 Systems under Study ................................................. 27
4.1 Document Generation and Management System ............ 27
4.2 Portfolio Management System .................................. 28
4.3 Custody Services Back-end ....................................... 30

II Techniques ................................................................. 33

5 Software Views ............................................................. 35
5.1 Polymetric Views ....................................................... 36
5.2 Polymetric Views for COBOL Mainframe Systems ......... 37
  5.2.1 Fact Extraction .................................................. 37
  5.2.2 Visualizing Dependencies ....................................... 38
  5.2.3 Interpreting Visualizations .................................... 39
5.3 Validation ............................................................. 41
  5.3.1 Observations .................................................... 41
  5.3.2 Validation Setup ................................................ 45
  5.3.3 Data Results .................................................... 46
  5.3.4 Interpretation ................................................... 48
5.4 Using Software Views ............................................... 49
  5.4.1 Treemapping as First Contact ............................... 50
  5.4.2 Polymetric Views for Initial Understanding ............... 51
  5.4.3 Resolving Issues During View Creation ................. 52
  5.4.4 Clustering for Initial Understanding ....................... 54
  5.4.5 Studying Exceptional Entities ............................ 57
5.5 Lessons Learned ..................................................... 58
  5.5.1 The Cost is in the Languages ............................... 59
  5.5.2 Nothing Works out-of-the-box .............................. 60
  5.5.3 Dinosaurs Don’t Like Pictures ............................ 60
  5.5.4 Knowledge is Scattered and Scarce ....................... 61
5.6 Summary .............................................................. 61

6 Feature Location .......................................................... 63
6.1 Replication Experiment ............................................. 64
  6.1.1 Technique ...................................................... 64
  6.1.2 Assumptions ................................................... 67
### III Patterns

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Fact Extraction</td>
<td>107</td>
</tr>
<tr>
<td>8.1</td>
<td>Gather Available Facts</td>
<td>110</td>
</tr>
<tr>
<td>8.2</td>
<td>Perform Text-Based Fact Extraction</td>
<td>112</td>
</tr>
<tr>
<td>8.3</td>
<td>Buy a Commercial Fact Extractor</td>
<td>114</td>
</tr>
<tr>
<td>9</td>
<td>Trace Capturing</td>
<td>117</td>
</tr>
<tr>
<td>9.1</td>
<td>(Ab)use a High-Level Monitoring Tool</td>
<td>120</td>
</tr>
<tr>
<td>9.2</td>
<td>(Ab)use a Debugger</td>
<td>122</td>
</tr>
<tr>
<td>9.3</td>
<td>Instrument Source Code</td>
<td>124</td>
</tr>
<tr>
<td>10</td>
<td>Process Patterns</td>
<td>127</td>
</tr>
<tr>
<td>10.1</td>
<td>Quick Feasibility Study</td>
<td>128</td>
</tr>
</tbody>
</table>

### IV Conclusions

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Conclusions and Recommendations</td>
<td>133</td>
</tr>
<tr>
<td>11.1</td>
<td>Research Questions Revisited</td>
<td>133</td>
</tr>
<tr>
<td>11.2</td>
<td>Recommendations</td>
<td>136</td>
</tr>
<tr>
<td>11.3</td>
<td>Concluding Thoughts</td>
<td>138</td>
</tr>
</tbody>
</table>

### V Appendices

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Thumbnail Patterns</td>
<td>141</td>
</tr>
<tr>
<td>A.1</td>
<td>Chat with the Maintainers</td>
<td>141</td>
</tr>
<tr>
<td>A.2</td>
<td>Read All the Code in One Hour</td>
<td>141</td>
</tr>
<tr>
<td>A.3</td>
<td>Interview During Demo</td>
<td>141</td>
</tr>
<tr>
<td>A.4</td>
<td>Build Prototypes</td>
<td>142</td>
</tr>
<tr>
<td>B</td>
<td>Pattern Format</td>
<td>143</td>
</tr>
<tr>
<td>C</td>
<td>Pattern Catalog</td>
<td>145</td>
</tr>
</tbody>
</table>

Bibliography 145
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Reverse engineering and related processes are transformations between or within abstraction levels, represented here in terms of life-cycle phases. Adopted from [Chikofsky and Cross, 1990], ©IEEE.</td>
</tr>
<tr>
<td>2.2</td>
<td>Staged model for the software life-cycle as described by Bennett and Rajlich [2000].</td>
</tr>
<tr>
<td>3.1</td>
<td>Four Tenets from the Manifesto for Agile Software Development.</td>
</tr>
<tr>
<td>3.2</td>
<td>Separated Business Domains with dedicated Domain Architectures supported by an Enterprise Architecture.</td>
</tr>
<tr>
<td>3.3</td>
<td>The disciplines and phases of the Rational Unified Process [Kruchten, 2003].</td>
</tr>
<tr>
<td>3.4</td>
<td>Preparation and execution of work while evolving and servicing a software system.</td>
</tr>
<tr>
<td>3.5</td>
<td>Typical Mainframe Environments</td>
</tr>
<tr>
<td>5.1</td>
<td>Up to five metrics can be visualized on one node. Image by Lanza and Ducasse [2003], ©IEEE, 2003</td>
</tr>
<tr>
<td>5.2</td>
<td>Overall Design Indicators</td>
</tr>
<tr>
<td>5.3</td>
<td>Exceptional Entities</td>
</tr>
<tr>
<td>5.4</td>
<td>Data Usage</td>
</tr>
<tr>
<td>5.5</td>
<td>The Functional Dependency View. Height and width of the nodes represent FAN-IN and FAN-OUT respectively. Areas of interest are manually annotated and magnified.</td>
</tr>
<tr>
<td>5.6</td>
<td>The Data Dependency View. Edges only represent data dependencies. Areas of interest are manually annotated and magnified.</td>
</tr>
<tr>
<td>5.7</td>
<td>Extracts from the Internal Functional Dependency View</td>
</tr>
<tr>
<td>5.8</td>
<td>Extracts from the Internal Data Dependency View. For image clarity, the internal copybooks are represented by black dots instead of white one; there are no missing copybooks in these views.</td>
</tr>
<tr>
<td>5.9</td>
<td>Extract from the Mixed Dependency View. Programs are denoted by white rectangles, the copybook by a white circle. Functional dependencies are black edges, data dependencies are gray edges.</td>
</tr>
<tr>
<td>5.10</td>
<td>Treemap of the Document Generation and Management System’s source code.</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Characteristics of a mechanistic versus an organic organization as listed by Vinekar and Huntley [2010], ©IEEE.</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Overview Systems under Study</td>
<td>31</td>
</tr>
<tr>
<td>5.1</td>
<td>Key Programs</td>
<td>47</td>
</tr>
<tr>
<td>5.2</td>
<td>Agreement on Clusters</td>
<td>47</td>
</tr>
<tr>
<td>5.3</td>
<td>Fact extraction methods we used.</td>
<td>59</td>
</tr>
<tr>
<td>6.1</td>
<td>Overview of Assumptions</td>
<td>67</td>
</tr>
<tr>
<td>6.2</td>
<td>Pilot Scenario-Feature Map</td>
<td>69</td>
</tr>
<tr>
<td>6.3</td>
<td>Scenario-Feature Map Iteration 1 &amp; 2</td>
<td>73</td>
</tr>
<tr>
<td>6.4</td>
<td>Candidates After Each Iteration</td>
<td>76</td>
</tr>
</tbody>
</table>
Part I

Setting the Scene
Chapter 1

Introduction

_Living in a vacuum sucks._
— Adrienne E. Gusoff

**in vitro.** *Biol.* [lit. ‘in glass’.] *In a test tube, culture dish, etc.; hence, outside a living body, under artificial conditions; also attrib., performed, obtained, or occurring in vitro.*

**in vivo.** *Biol.* *Within the living organism; also attrib., performed, obtained, or occurring in vivo.*

[Oxford English Dictionary]

In biology and medicine “in vitro” and “in vivo” are considered two complimentary sides of the same coin, living in symbiotic relationship with one another for the greater good of the research discipline. In vitro research is necessary, as laboratory conditions allow the investigator to have full control over the experimental context, necessary to study the causal relationship between the treatment and the outcome. In vivo research, on the other hand, allows the investigator to study a phenomenon in its real-life context, hence to confirm whether the treatment is applicable in reality.

We argue that software engineering research could benefit from a similar symbiotic relationship between in vitro and in vivo research. Transcribing these terms into a software engineering vocabulary, in vitro research is then defined as “the manipulation of software artifacts, methods and techniques in a controlled, artificial environment”, while in vivo research becomes “the analysis of software artifacts, methods and techniques in uncontrolled, realistic circumstances”. Based on these definitions, we describe the kind of problems that we experienced while adopting reverse engineering techniques that have proven to be worthwhile in vitro into the in vivo context of large organization from the financial services industry.
CHAPTER 1. INTRODUCTION

1.1 Software Evolution

The ability to evolve software rapidly and reliably is a major challenge for today’s software engineering community [Mens and Demeyer, 2008].

“Success breeds change, change is unpredictable, and the design that seemed sufficient yesterday becomes today’s bottleneck.” [Beck, 2010]

This is how Kent Beck introduced his perspective on ‘The Inevitability of Evolution’; only failed projects are satisfied with their initial design. The fact that success breeds change is, of course, a manifestation of Lehman’s law of continuing change which states that a system must be continually adapted or become progressively less satisfactory [Lehman, 1979]. That this change is unpredictable, we know from over forty years of software engineering experience [Bennett and Rajlich, 2000]. There is no such thing as a perfect design to meet all possible future requirements. Beck describes this yearning for the “right” design as a generational anomaly. Both his father’s and his daughter’s generation know that change is a natural part of development [Beck, 2010].

For a software system not to become victim of its own success, constant care is necessary to keep it in a state that allows it to evolve. However, this kind of maintenance requires intimate knowledge of the system and how it is built [Bennett and Rajlich, 2000]. For many systems, though, this knowledge is lost due to experts leaving the project. Subsequent uncontrolled changes further deteriorate the system’s structure. These systems are called legacy systems: they are still valuable to the organization owning it, but they are inherited by employees who lack intimate knowledge about it [Demeyer et al., 2008].

In the financial services industry, the constant market pressure, together with the frequent mergers and acquisitions, force enterprises to constantly optimize their internal organization. This is necessarily tied to the software systems supporting the organization’s activities. When these systems resist modification and evolution they quickly become an insurmountable obstacle on the road towards an agile enterprise. For example, Homann et al. [2004] describe how banks are increasingly reorganizing their value chain into independently operable functional units to decrease costs and simultaneously enhance customer utility. However, reorganizing the business units according to the value chain requires to rethink the supporting IT-portfolio as well. Implementing such a reorganization requires an incremental approach (also known as “chicken little” [Brodie and Stonebraker, 1995]) where existing software components are reverse engineered to see if and how they might fit into the new enterprise architecture.

1.2 Reverse Engineering

Reverse engineering is the process of recovering knowledge from existing systems. As such, it can ease the modification of software systems to help them adapt to changing requirements [Tonella et al., 2007]. Researchers in reverse engineering like to see themselves as detectives, piecing together clues about a system’s design and the crimes that have been committed in its
evolution, or as archeologists, reconstructing models of structures buried in the accumulated deposits of software patches and fixes [Wills and Cross, 1996].

The reverse engineering community is a vibrant research community, resulting in many useful techniques and research prototypes to recover this knowledge. However, few of them have been exploited industrially in significant ways, which is evidenced by the lack of empirical (in vivo) research [Tonella et al., 2007] and the lack of a significant tools and service vendor market (enabling in vivo research) [Bennett and Rajlich, 2000].

1.3 Research Questions

This dissertation takes a first step in investigating the lacking industrial adoption of reverse engineering techniques by studying the applicability of existing state-of-the-art reverse engineering techniques on mainframes systems in the financial services industry.

While doing so, we try to answer the following questions.

— What is the goal of reverse engineering?
— What are the characteristics of state-of-the-art reverse engineering techniques?
— What are the characteristics of organizations in the financial services industry?
— What are the characteristics of mainframes used in the financial services industry?
— Who can benefit from reverse engineering in the financial services industry?
— Which factors — both technical and organizational — make it difficult to apply reverse engineering in the financial services industry?

After identifying these characteristics and the resulting difficulties, we summarize our experiences as recommendations by answering the following questions.

— What can organizations do to better support reverse engineering?
— What can the research community do to better support reverse engineering in the financial services industry?
— What can practitioners do to work around these difficulties while waiting for their organization and the research community to catch up?

1.4 Guide to the reader

Part I sets the scene. Chapter 2 (p.7) introduces reverse engineering and its different facets. It also lists the requirements for a good reverse engineering technique. Chapter 3 (p.17) explains the peculiarities of the financial services industry. In particular it explains technicalities of the mainframe, organizational structures and their impact on IT architecture and development methods.
CHAPTER 1. INTRODUCTION

All these characteristics are introduced to explain and understand the feasibility of applying reverse engineering in the financial services industry. Chapter 4 (p.27) lists the systems we have studied from two organizations in the financial services industry.

Part II focusses on three techniques of which we show the technical feasibility by construction. For each technique we argue why it is a good reverse engineering technique worth applying in industry, how we applied it, which difficulties we had to overcome and what the root causes of these difficulties were. We conclude with the lessons we have learned, both for academia and for industry. Chapter 5 (p.35) explains the application of software views using static information only. Chapter 6 (p.63) explains feature location, mainly based on dynamic analysis. Chapter 7 (p.89) explains the semi-automatic technical redocumentation of a legacy system.

Part III bundles a set of reverse engineering patterns which we discovered while applying these techniques in industry. They represent reusable knowledge for practitioners in the field. These chapters are meant to be concise, self-contained and usable in practice. Chapter 8 (p.107) describes potential approaches to extract static facts from legacy systems. Chapter 9 (p.117) describes several approaches to extract run-time information from legacy systems. Chapter 10 (p.127) contains one pattern on how to study the feasibility of the approaches from the previous two chapters. An overview of the patterns and their intents can be found in Appendix C (p.145).

Part IV wraps up by revisiting our research questions using the lessons we learned and formulating advice to academia, the financial services industry and practitioners.
In this chapter we introduce the notion of reverse engineering and its main objective: recovering knowledge. We list some incentives to conduct reverse engineering and position it within software maintenance and evolution. We delve deeper into the three activities that make up reverse engineering and uncover some requirements reverse engineering techniques should adhere to in order to effectively support comprehension. Lastly, we explain the difference between techniques and tools, both harboring explicit knowledge about how to reverse engineer, and we introduce patterns as a way to organize implicit knowledge.

2.1 Taxonomy

To explain reverse engineering, one must first revisit the notions of forward engineering and the lifecycle of a software system. For simplicity, Figure 2.1 (p.8) shows three lifecycle phases:

(i) requirements as the specification of the problem,
(ii) design as the specification of a solution, and
(iii) implementation as a realization of that solution.

Using these phases as illustration, forward engineering is the process of moving from requirements, over design to the physical implementation of a system. This is the typical process followed to create a system, where a system can be a code fragment or a complex set of interacting programs. During reverse engineering these phases are visited in the opposite direction. As originally defined by Chikofsky and Cross [1990], reverse engineering is:
Analyzing a system implies that there already is a system. Of course, something must have been engineered before you can reverse engineer it. Furthermore, reverse engineering is a process of examination, thus no changes are made to the subject system. Identifying components and relationships boils down to uncovering the structure of the system. Indeed, the main goal of reverse engineering software systems is to gain a basic understanding of a system and its structure. Creating representations of it in another form means providing an alternative view of the same information within the same abstraction level. For example, one can provide a data flow or control flow diagram to get a more human-readable and focussed view on a piece of source code. This is called redocumentation and can be seen as a transformation within one life-cycle phase in Figure 2.1 (p.8). Creating representations of it at a higher level of abstraction, on the other hand, is a transformation between life-cycle phases. This is called design recovery in which one recreates design abstractions not only from source code, but also from personal experience and general knowledge about the problem and application domains [Chikofsky and Cross, 1990].

Since the taxonomy was introduced in 1990, reverse engineering has extended well beyond redocumentation and design recovery. Although the intent of their reverse engineering definition remains intact, it was considered too limiting to characterize the entire field of reverse engineering. Therefore, almost two decades later, Tonella et al. [2007] broadened the definition by characterizing the reverse engineering discipline as follows.

“Every method aimed at recovering knowledge about an existing software system in support to the execution of a software engineering task.”
Knowledge is still being recovered from existing software systems, but no longer is it limited to “components and their interrelationships”. Furthermore, they emphasize the task-oriented nature of reverse engineering. One does not just try to understand a system for the fun of it, one is looking for very specific knowledge about (a part of) a system in order to, for example, make a modification. Thus, a reverse engineering step typically precedes a forward engineering step in which the newly gained knowledge is exploited to make modifications to the system. The combination of both steps is called reengineering.

This thesis specifically focuses on the applicability of reverse engineering without taking into account succeeding forward steps. However, bear in mind that reverse engineering is always applied in the context of facilitating succeeding (forward) software engineering tasks, be it the really executing those tasks — as the definition of Tonella et al. [2007] suggests — or merely studying the feasibility or estimating the cost of possible software engineering tasks.

2.2 Incentives

Since reverse engineering is the first step in a reengineering process, the incentives for applying reverse engineering are similar to those of reengineering. While describing their object oriented reengineering patterns, Demeyer et al. [2008] identified some early signs that warrant reverse engineering or preventive reengineering, including:

- obsolete or no documentation,
- departure of original developers,
- lack of intimate system and domain knowledge,
- limited understanding of the system as a whole.

Furthermore, based on his experience from the field, Dewar [1999] compiled a list of forces that seems to trigger larger reengineering projects. We distill three categories.

**Business Requirements** Making data more accessible to the customers or simply offering new products or a new combination of existing products can require the modification of existing business processes and their supporting software systems. These changing business requirements and the inability of legacy systems to cope with these changes is an incentive to reengineer.

**Corporate Standards** Large organizations typically impose standard packages or infrastructure which need to be adopted by satellite business units. Depending on the structure of legacy systems, this can be a big challenge.

**Technical Platform** The hardware or software that legacy systems depend on is no longer being supported by the vendor or the organization itself. Or the people within the organization lack the knowledge to operate and maintain this older technology, which necessitates a migration to newer, supported technology or a complete rebuild. Either way, the knowledge embedded in the legacy systems should not be lost.
2.3 Software Maintenance and Evolution

IEEE Standard 1219 [IEEE, 1999] defines software maintenance as ‘the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment’. Thus, all activities on a software system after initial delivery are considered maintenance.

A survey by Lientz et al. [1978] already showed that the maintenance cost ranges up to 75–80% of the total life-cycle cost of systems. Furthermore, they show that the incorporation of new user requirements is the core problem for software maintenance. This could be remedied if these changes could be anticipated at design time. But Bennett and Rajlich [2000] explain that the fundamental problem — supported by 40 years of hard experience — is that most required changes are unpredictable. Therefore, maintenance and maintenance research is very important.

To better define a research roadmap for software maintenance, Bennett and Rajlich [2000] have defined a staged model for the software life-cycle, depicted in Figure 2.2. The aim of this model is to split up maintenances into several stages that can be investigated separately. The following stages are described.

**Initial Development**  During initial development, the first version of the software system is created. It already possesses the architecture that will persist throughout the rest of its life. Furthermore, in this phase, the development team acquires the necessary knowledge for the second stage. This first stage is not part of maintenance.

**Evolution**  The goal of the evolution stage is to adapt to the ever-changing user requirements and operating environment. As Lehman [1979] stated, software is being evolved because it is successful on the marketplace. In this stage, return on investment for the software system is excellent. Yet, both the software architecture and the knowledge of the initial software
development team are crucial for this stage as they enable the team to make substantial changes without damaging the architectural integrity. This stage can last for twenty years or longer.

**Servicing** When either the software architecture or the initial knowledge disappears, the system loses evolvability and slips into the servicing stage. Such a loss of knowledge is typically triggered by the loss of key personnel. In this stage, only small tactical changes like patches, code changes and wrappers are possible, which will further deteriorate the architecture. The return on investment of changes is much more marginal.

**Phase-out** During phase-out, no more servicing is undertaken but the system remains in production. Users will have to work around the system’s deficiencies.

**Close-down** During close-down the software system is switched off and the users are redirected to a replacement.

The goal is to keep the system in the evolution stage — and afterwards in the servicing stage — as long as possible. Each stage has very different characteristics and, thus very different solutions are necessary to manage them. However, it is not always clear in which stage a software system resides.

A legacy systems typically reside in the servicing stage of the staged model. A legacy is something valuable that one inherits when someone else dies. Similarly, a legacy system is valuable (to the organization). The fact that it is inherited means that the original creators have died (or left the organization) and with it most likely the necessary knowledge to evolve the system.

We firmly believe that using reverse engineering techniques to recover knowledge and increase understanding can (i) help prevent existing software systems to slip from the evolution stage to the servicing stage, and (ii) help decrease the cost of keeping legacy systems in the servicing stage. However, this dissertation only focusses on the technical applicability of reverse engineering techniques in industry, which we see as a first step to actually showing their effectiveness.

### 2.4 Activities

In light of the main objective, namely understanding or comprehending a software system, Tilley et al. [1996] have identified three canonical activities that make up reverse engineering.

**Data Gathering** Gathering raw data from the subject system is an essential activity in reverse engineering. It is used to identify the system’s artifacts and the relationships between them. To obtain this raw data, one can use static analysis (e.g., source code parsing), dynamic analysis (e.g., execution profiling or tracing), historic analysis (e.g., repository mining) or informal data extraction (e.g., interviewing experts).

**Knowledge Organization** Knowledge is the sum of what is known. This does not only include the raw data that was gathered, but also higher level abstractions and relationships derived from the raw data. Organizing the knowledge in such a way that facilitates efficient storage and retrieval, permits analyses and reflects the user’s perspective of the system.
CHAPTER 2. REVERSE ENGINEERING

Information Exploration As opposed to knowledge, information is defined as contextually and selectively communicated knowledge. That is, a small part of knowledge that is relevant for the task at hand. Exploring the information holds the key to system understanding. It includes navigation of the knowledge base, which is inherently non-linear and multi-dimensional for software systems and analysis to derive extra information which can in turn be stored as new knowledge. Furthermore, the presentation of this information should be flexible because one cannot know in advance which information will be necessary in what form to accomplish a software engineering task.

2.5 Comprehension

Using these three activities, we can increase the overall understanding or comprehension of a software system. In doing so, we can resolve so-called “known unknowns”, i.e., the things that we know we do not know and by acknowledging this lack of information we can look for an answer. But also the “unknown unknowns”, i.e., things we do not know we do not know.\(^1\) The latter are much harder to address since we can only acknowledge them once we have encountered them.

Furthermore, reverse engineering in general — and the information exploration activity in particular — is a highly iterative and incremental process. It involves both bottom-up and top-down comprehension [Storey et al., 1997]. Starting from program code, bottom-up comprehension incrementally chunks information into higher-level abstractions. Top-down comprehension, on the other hand, reconstructs domain knowledge and maps that to the actual code. Top-down comprehension heavily relies on hypothesis refinement and verification where both the rejection and acceptance of a hypothesis contributes to the overall reverse engineering goal. When reverse engineering in the absence of domain knowledge — only having access to the running system and possibly the program code — an analyst is forced to start bottom-up, forming hypotheses and validating them using the same bottom-up information. In the presence of — most likely outdated — documentation, an analyst can use it to form initial hypotheses which can subsequently be verified using reverse engineering tools. The presence of such documentation can considerably speed-up the comprehension process. In the best case, there is a domain expert available who has detailed and up-to-date knowledge about the problem domain, yet is uncertain how this maps to the source-code. Nevertheless, in our experience, getting information from a domain expert is not that difficult, whereas getting the information you need is. Here, reverse engineering tools can help focus the interviews with the domain expert (cfr. “Interview during demo” in [Demeyer et al., 2008]).

\(^1\)According to wikipedia, the terminology of the “known unknowns” and “unknown unknowns” is attributed to the military establishment and is used to distinguish the different types of risk that one should be prepared for when planning a mission. The term “known knowns” does exist as well, but is not used here.
2.6 Requirements

Based on these characteristics of comprehension, we state that a reverse engineering technique that wants to successfully support comprehension will have to adhere to the following requirements.

- The technique should serve as a starting point for understanding, even without the presence of domain knowledge or expertise.
- The technique must support rapid incremental and iterative analysis to quickly verify or reject hypotheses.
- The technique must reveal unknowns (both known and unknown) that facilitate the formulation of new hypotheses.

We will use these requirements when selecting promising reverse engineering techniques to apply in the financial services industry.

2.7 Techniques vs Tools

As defined by Tonella et al. [2007], a reverse engineering technique is a solution to one or more known problems in reverse engineering. A reverse engineering tool, on the other hand, is defined as an application which implements and supports the usage of one or more techniques. Thus, each technique can be implemented or supported in different ways by several tools.

This dissertation focuses on the applicability of reverse engineering techniques, not of specific tools. However, Müller et al. [2000] identified tool adoption as the biggest challenge to increased effectiveness of reverse engineering: “tools can’t be effective if they aren’t used, and most software engineers have little knowledge of current tools and their capabilities”. This is also our major concern: how to implement state-of-the-art reverse engineering techniques on top of state-of-practice tools and processes. Therefore, we will mainly draw conclusions from the (in)ability to implement a technique in our industrial context.

2.8 Patterns

Patterns have been introduced by Alexander et al. [1977] as a way to ‘organize implicit knowledge about how people solve recurring problems when they go about building things’. Patterns are easy to remember and set out as if-then propositions. That is, if you have a problem within a certain context, then this solution could be a good way to solve it. If you have the same problem in another context, then another solution might be better. Furthermore, Mens and Demeyer [2008, chapter 5] describe how the problems solved by patterns entail a number of conflicting forces which need to be balanced and the described solution will discuss a number of trade-offs, i.e., the pros and cons of applying that pattern. As such, patterns help to convey the complex decision process that experts use to decide whether or not a solution suits a given problem.
Patterns are often seen as recipes, i.e., a step-by-step solution to a given problem. However, it is not as simple as that. In a light-hearted attempt to expose the difficulties in applying patterns, titled ‘Has the Pattern Emperor any Clothes? A Controversy in Three Acts’, d’Adderio et al. [2002] describe five ways patterns are actually used by practitioners.

- **Direct Reuse of Knowledge** — A solution to a recurring problem has been described and can be reused by other practitioners to solve a similar problem. The contribution, however, is not a step-by-step solution, but a discussion on conflicting forces and trade-offs.
- **Learning Tool** — By reading patterns, one can improve its knowledge on a certain topic. This kind of reading is reflective, relating what you read to what you have done. As such, novices and experts will learn different things when reading patterns.
- **Teaching Tool** — A well-written pattern does a good job explaining the pattern.
- **Communications Tool** — Using pattern names among experts can speed-up discussion of competing solutions.
- **Provoke Thought** — A pattern is an invitation to perform a thought experiment, it gives a tentative solution to consider.

Beck and Cunningham [1987] were the first to apply the pattern idea to software, but it has been popularized in the software engineering community by Gamma et al. [1995] with their book on design patterns. Since then, two categories of patterns have emerged: design oriented patterns and process oriented patterns.

Design oriented patterns describe a product, e.g., how a part of the design of a software system should look like in the end. Gamma et al. [1995] describe solutions for object oriented design problems. Fowler [2002] focusses on enterprise application architecture and Hohpe and Woolf [2003] on enterprise integration using messaging solutions. Recently, a number of patterns for service-oriented design have been described as well [Erl, 2009]. These are excellent sources when creating software systems, be it from scratch or when redesigning existing systems.

As opposed to describing what to produce, process oriented patterns describe how to get from 'here' to 'there’. Such patterns typically describe the situation before and after, along with the steps needed to make the transition. A perfect example is the book 'Refactoring to Patterns’ [Kerievsky, 2004] which explains in pattern form how to spot design problems and how to solve them by applying, e.g., the design patterns from [Gamma et al., 1995]. Other examples include patterns on how to organize software development [Coplien and Harrison, 2005], how to cope with legacy code [Feathers, 2004] and how to reengineer object oriented systems [Demeyer et al., 2008].

Part III of this dissertation includes some process oriented patterns on how to apply reverse engineering, including ways to perform the *Data Gathering* activity — both for static facts and run-time information — and one supporting pattern on conducting quick feasibility studies. The format of the patterns is explained in Appendix B (p.143). An overview of the patterns and their intents can be found in Appendix C (p.145).
2.9 Summary

The goal of reverse engineering is to recover knowledge about an existing software system. As such, it can help humans to better understand a system and enable them to execute subsequent software engineering tasks in the evolution and servicing stage of a system. To do so effectively, reverse engineering techniques and tools should reveal unknowns and allow rapid incremental and iterative analyses.

We use these requirements to select promising reverse engineering techniques in Part II and see whether they hold up in our industrial context. Furthermore, we provide some reverse engineering patterns in Part III which can help practitioners with the data gathering activity of reverse engineering.
This chapter introduces the industrial context in which we apply reverse engineering techniques. First, we highlight the typical organizational structures in large organizations within the financial services industry and how they are supported by mainframes. Then we take a look at how IT is typically structured and performed within these organizations. At last, we position reverse engineering within the confines of organization in the financial services industry by identifying specific IT activities and roles that we believe can benefit from reverse engineering.

### 3.1 Organizational Culture and Development Processes

According to Burns and Stalker [1994] organizations either have a more flexible *organic* culture or a more structured *mechanistic* culture. Vinekar and Huntley [2010] summarize the characteristics of both extremes in Table 3.1 (p.18). The mechanistic culture is very much focussed on centralized top-down control. There are very clearly separated and small units of work which are distributed and supervised using rigid rules and standard operating procedures. An organic culture, on the other hand, uses more fluid work processes. It stimulates employees to take different roles and work together. Control is more decentralized. Of course, both cultures have their merits and can be deliberately created and maintained to use employees in the most efficient manner depending on the organization’s circumstances [Burns and Stalker, 1994, p.119].

Similarly, there are two widely different approaches to developing software. One is the more industrialized, plan-based way of working following, e.g., the waterfall model. This method stems from the ‘real’ engineering disciplines where it is successfully used to build bridges, aircraft, etc. The other is agile software development, promoting a more organic way of working, responding
to change rather than following a plan. The core of agile methods is summarized by the agile manifesto \(^1\) in Figure 3.1 (p.18). Simplistically speaking, one could say that the agile community values the left side more, whereas the plan-based community focusses more on the right side. Both camps contend that their method is superior to the other, but again, both have their merits [Vinekar and Huntley, 2010].

Traditionally, agile methods are believed to be more suited for dynamic projects with considerable change, whereas plan-based methods are more suited for relatively stable projects [Boehm and Turner, 2003]. However, Vinekar and Huntley [2010] argue that the organizational culture is also an important factor influencing the applicability and effectiveness of a development process. More specifically, they argue that it will be very difficult for a mechanistic organization to follow the first two tenets of the agile manifesto as their policies dictate structured processes and tools and require comprehensive documentation as a control structure. However, introducing frequent customer meetings and planning more and shorter iterations can easily be done within the confines of this mechanistic culture. Vice versa, while not touching the first two tenets for the organic organizations, meeting the customers might not be necessary that often when facing stable requirements and they can have fewer, longer iterations with more upfront design. Mixing these two processes results in what they call “iterative plan-based development” and “optimized agile development”. These findings are based on observations from the field where these hybrid approaches are common practice. IBM’s Rational Unified Process [Kruchten, 2003] is an excellent example of an iterative plan-based approach, whereas studies\(^2\) show that most agile teams do some upfront design, especially upfront architecture seems to be a big win in practice [Vinekar and Huntley, 2010].

\(^1\)http://agilemanifesto.org/
\(^2\)http://www.ambysoft.com/surveys/projectInitiation2009.html

---

**Table 3.1:** Characteristics of a mechanistic versus an organic organization as listed by Vinekar and Huntley [2010]. ©IEEE.

<table>
<thead>
<tr>
<th>Organic</th>
<th>Mechanistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employees take different roles</td>
<td>Labor specialization</td>
</tr>
<tr>
<td>Network of teams</td>
<td>Hierarchy of authority</td>
</tr>
<tr>
<td>Lateral communication</td>
<td>Vertical communication</td>
</tr>
<tr>
<td>Consulting, not command</td>
<td>Superiors and subordinates</td>
</tr>
<tr>
<td>Information and advice</td>
<td>Instructions and decisions</td>
</tr>
<tr>
<td>Commitment to the goal</td>
<td>Loyalty and obedience</td>
</tr>
<tr>
<td>Employees work together</td>
<td>Employees work separately</td>
</tr>
<tr>
<td>Decentralization</td>
<td>Centralization</td>
</tr>
<tr>
<td>Fluid work processes</td>
<td>Rules and standard operating procedures</td>
</tr>
<tr>
<td>Verbal communication</td>
<td>Written communication</td>
</tr>
</tbody>
</table>

**Figure 3.1:** Four Tenets from the Manifesto for Agile Software Development.
That the culture and structure of an organization has an influence on software development is no surprise. In 2002, Xia and Lee [2004] conducted an empirical study on 541 information system development projects of North American organizations. In this study, they relate technological and organizational complexity to project performance metrics such as delivery time, cost, functionality and user satisfaction. Their main findings are that the technological aspects of a development project are perceived as ‘most complex’, but the numbers show that the organizational aspects have a more significant impact on the actual performance and outcome of the project. Clearly, organizational culture and structures play an important role when developing and maintaining software.

Organizations in the financial services industry — and specifically the two where we conducted our research — have a mechanistic culture. Hence, they promote labor specialization and try to maintain centralized control. Therefore, they rely on rigorous processes and standardized tools resulting in an iterative plan-based development process.

3.2 IT Structures

Julius Caesar knew the power of Divide and Conquer long before the introduction of IT. When something is too big to manage as a whole, break it up into manageable parts. The idea is applicable to all kinds of systems ranging from software systems to societies to entire eco-systems. Whether it is a hierarchical structure imposed from the top down (such as in governments) or an organic structure emerging from the bottom up (such as in ant colonies), structure is necessary to sustain a system. And the bigger the system, the more structure is needed.

Large banking and insurance organizations are typically divided into departments related to the kind of products they offer. Each department is responsible for one core business activity of the organization. Of course, other, more supporting departments will be necessary as well, such as Human Resources (HR). These are no core activities, but nonetheless necessary to make the organization function as a whole.

Unsurprisingly, the IT to support all these activities is structured in the same way. Figure 3.2 (p.20) shows an example of such a structure. You see several clearly separated business domains within which activities relating to a certain type of product are performed. Each business domain has its own architecture and software systems. However, to preserve coherence, all these domains are governed by one enterprise architecture.

3.2.1 Enterprise-wide Structures

On an architectural level, these enterprise-wide structures contain rules on what kind of interactions are possible within and between business domains. For example, how can data flow within a business domain, how can data be exposed to other business domains, when is synchronous communication allowed, when is asynchronous communication desired, and so on. Furthermore, some technological limitations can be imposed, such as which tools are mandatory to use, which software packages are supported by the organization, what kind of technology is absolutely
prohibited, and so on. Besides the rules about artifacts within the organization, there are also
guidelines about how to create them. These **thinking frameworks** provide a structured approach of
doing IT. All these rules and guidelines provide an enterprise-wide structure which is enforced
onto each business domain.

Apart from this enterprise architecture, there are also more concrete artifacts that are not specific
to one business domain, represented by the underlying layers in Figure 3.2 (p.20). For example, the
technical infrastructure in place to support the IT process and run the systems. Furthermore, some
business services can be shared by most business domains, such as a shared customer database or
a shared report creation framework.

### 3.2.2 Business Domains

The architecture of one business domain— which we will call domain architecture— comprises
of business processes and an indication of which parts of these processes are automated and by
which software system. Essentially, they represent the mapping between how business works and
how this is supported by IT. It contains the common vocabulary between business and IT for that
particular business domain.

Starting from such a domain architecture, several systems and applications will have to be built
to meet the business requirements. Several systems cover the automation needs of all business
functionalities within the business domain. Each system on its own comprises of several business
applications which can be used by business users, e.g., financial analyst or local branch managers.
For each of these systems or applications functional analysis will be conducted (e.g., by means
of use cases) and a technical design will be created to see how this might fit into already existing
implementations. All these people and artifacts should, of course, adhere to the rules as described
in both the domain architecture and the enterprise architecture.
3.3 IT Activities

There are two large stages within the life-cycle of a software system: creating the system (getting it running) and maintaining the system (keeping it running). While the initial development of systems typically receives the most attention (both in research and in industry), Lientz et al. [1978] already showed that maintenance takes up most of the total life-cycle cost in practice.

Maintenance activities have been categorized in various ways. Most notably, Swanson [1976] introduced adaptive, corrective and perfective maintenance, a categorization based on the intention of the activity. More recently, Kitchenham et al. [1999] have created an ontology with a more objective categorization. They differentiate between corrections and enhancements where the latter can be due to new requirements, changed requirements or simply a change in implementation.

However, while these categories can help researchers study maintenance, they do not help practitioners perform or organize maintenance activities. Instead, industry uses two much more pragmatic categories. We will call them evolutionary development and routine maintenance. Evolutionary development aims to incorporate large chunks of new or changed requirements, which is usually accomplished by starting a development project, ultimately resulting in a significantly different version of the software system. Routine maintenance, on the other hand, aims to fix bugs and implement small enhancements, which is accomplished by a dedicated service team, constantly reacting to change requests and bug reports.

Evolutionary development, like initial development, is typically covered by elaborate development processes such as, for instance, the Rational Unified Process [Kruchten, 2003] as depicted in Figure 3.3 (p.21). It identifies nine disciplines that represent groups of related activities performed...
when developing a software system. One needs to model (part of) the business domain, gather requirements, analyze the problem, design solutions, implement, test and eventually deploy the system. Furthermore, the process is subdivided into four phases, where each phases focusses more on some activities and less on others. In the inception phase one mainly specifies the idea and scope of the project. The elaboration phase focusses on analyzing the problem domain and establishing a sound architectural foundation to be able to create an elaborate project plan. The construction phase is in fact a manufacturing process with a focus on optimization. Here, all remaining parts of the system are developed, integrated and tested. The transition phase is all about getting the manufactured system into production. Each of these phases can be subdivided in several iterations. Note that this is not a pure waterfall process. One can start implementing prototypes in the inception phase and adjust requirements in the construction phase. It does, however, focus on making industrialized software engineering feasible in practice.

Clearly, the approaches to tackle evolutionary development and routine maintenance are quite different. Which category a certain change belongs to is based on the effort needed to make the change. If the impact of the change is limited then it will be categorized as routine maintenance. However, if the impact is profound or unclear then a project will be dedicated to it. This is shown on the horizontal axis of Figure 3.4 (p.22).

A second dimension in categorizing maintenance work is the distinction between work preparation and work execution, which is shown in the vertical axis of Figure 3.4 (p.22). The goal of work preparation is to get a reliable cost estimate. For routine maintenance this simply means analyzing the impact of a change request to decide whether or not it should become a project. For a project this means performing a prestudy to eliminate unknowns that could derail the project later on. This corresponds to the inception and elaboration phase of the Rational Unified Process, after which enough risks have been mitigated to start manufacturing the software system. The work execution phase for routine maintenance is characterized by the implementation of a change, be it a bug fix or a small enhancement. For evolutionary development, the execution phase is characterized by the construction and deployment of a new version of the software system. This should be relatively straightforward as most unknowns will have been eliminated in the preparation phase.
3.4 IT Roles

We now have an idea about what kind of activities are performed, but not about who performs them. The Rational Unified Process relates several roles to each discipline. A role defines the behavior (a specific activity) and the responsibility (usually a resulting artifact) of an individual. The basic Rational Unified Process [Kruchten, 2003] defines up to 30 roles. The Enterprise Unified Process [Ambler et al., 2005], an extension to the Rational Unified Process including, e.g., enterprise architecture, defines 25 more roles.

For example, there is no such thing as an architect. There are enterprise and technology architects overseeing enterprise-structures and decisions. There are domain architects ensuring consistency within their business domain. Furthermore, there are usually architects on project and application level as well. The same holds for analysts, where there are business analysts gathering and analyzing the requirements of one business domain and functional analysts analyzing the requirements of one software system. Furthermore, there are technical designers mapping functional solution onto technical structures, developers implementing software systems, maintainers making changes to existing systems. Testers exist on the IT side for developer and integration testing, but also on the business side for more functional and acceptance testing.

These are just some roles from the core process disciplines, which are the first six disciplines depicted in Figure 3.3 (p.21). There are also many roles for the core supporting disciplines, the last three disciplines depicted in Figure 3.3 (p.21). The latter include all kinds of managers, but also dedicated operational groups to keep the supporting infrastructure running. For example, database management, transaction management, change management, configuration management, managing the development infrastructure, etc.

Although Rational Unified Process encourages organizations to make employees take up several roles, the mechanistic culture typically forces employees to only take one or two roles.

This dissertation will focus on the following roles.

**Prestudy Leader** The role of prestudy leader will either be taken up by a business analyst, as the ideal liaison between business and IT, or by a project architect, overseeing all parts of the project. A prestudy leader is responsible for making a reliable cost estimate for a subsequent development project. As such, he has the means to initiate certain analyses to identify and resolve unknowns that pose a risk.

**Technical Designer** The role of technical designer will usually be taken up by an expert of the software system in question. He is responsible for assessing the impact of proposed changes to the current technical structure of the system. These proposed changes can be articulated by, for example, a prestudy leader looking for more expertise.

**Maintainer** The role of maintainer embodies interpretation of bug reports and enhancement request, locating code fragments that need modification and implementation and testing of these changes. He is in charge of performing routine maintenance on an existing software system. As such, she will incrementally gain intimate knowledge about the inner workings of the system, but only of those parts she has to modify.
When discussing the techniques in the remainder of this dissertation, we will use the term analyst as a general term to denote anyone using reverse engineering to analyze a system, regardless of the three specific roles above. These are also the kind of people we refer to as practitioners when formulating advice in our conclusion.

Furthermore, the following supporting role will be used in this dissertation.

**Technical Operator** We use the term technical operator to denote any person performing a technical role within the supporting departments. That is, the people responsible for the enterprise-wide technical infrastructure enabling developers, designers, analysts and architects to do their job.

### 3.5 Enterprise Applications

There are many different kinds of software. The software necessary in the telecom industry, for example, is widely different from the software used in the financial services industry. The latter is usually categorized under the term enterprise applications, also known as information systems.

Fowler [2002] has a very pragmatic take on the notion of enterprise applications. Although unable to give a precise definition, he gives some characterizing indications. Enterprise applications usually involve a lot of persistent data which is accessed concurrently by many people. A lot of data accessible to many different kinds of people means even more user interface screens to handle it. Since enterprise applications rarely live on an island, there is also a need to integrate with other applications scattered around the enterprise, built with different technologies and used within different business units as part of different business processes. Lastly, these enterprise applications contain business logic that implement business rules which are not under the control of software engineers. Dealing with these strange rules that often interact in surprising ways, and dealing with the thousands of one-off special cases that will have to be incorporated in the rules, leads to complex business “illogic”. In short, enterprise applications use business logic to process data on a large scale, which is also called commercial data processing.

### 3.6 Mainframes

While the mainframe is regarded by many as the dinosaur of computing, there is no denying its dominance over the landscape of large-scale enterprise computing over the last five decades. Although other infrastructure is used as well, they are still indispensable in many — if not all — of today’s enterprise systems, because they are very well suited for commercial data processing. The reliability of the hardware, the security of the operating system and the integrity of the data are unmatched by any other technology. Furthermore, the mainframe is known for its enormous transaction processing capabilities and for supporting thousands of users and application programs concurrently accessing numerous resources and managing terabytes of data. Lastly, the mainframe provides several facilities for ensuring accountability (who did what and when), a must for all large
enterprises, especially those in the financial services industry. On the other hand, mainframes are very bad at *number crunching* and have almost no graphical support (think textual 3270 terminals) [Stephens, 2008].

When trying to define the term ‘mainframe’, IBM comes up with some interesting characteristics, such as “centralized control of resources” and “a style of operation, often involving dedicated operations staff who use detailed operations procedure books and highly organized procedures [...]” [Ebbers *et al.*, 2006]. The latter illustrates the typical mechanistic culture of organizations using mainframe. The sheer size of these organizations requires them to be highly organized, typically resulting in separate division and teams with dedicated responsibilities using strict operational procedures.

In addition to these organizational characteristics reflected in mainframes, there are also some technical characteristics that make it different from other computing environments. We list three of them that will be relevant in the remainder of this dissertation.

**Mainframe Languages** Many languages — both old and new — are being used on mainframes today. Some have very specific purposes, such as JCL for controlling batch jobs and SQL for querying databases. Others are used as general purpose programming languages for implementing the necessary business logic or infrastructural code, such as COBOL, PL/I and FORTRAN. Furthermore, many levels of abstraction are supported by different languages that range from very low-level languages like ASSEMBLER, which is very close to machine code, to high-level languages like VAX and EGL. The latter were created to ease the implementation of business logic by abstracting away technical tasks like accessing files or databases and usually generate more low-level code like COBOL or JAVA. A last category are the CASE-tools. On top of providing a high-level programming language, they provide an entire environment supporting not only development but also analysis and design of software. This type of tools also generates more low-level code to be compiled. Some of these languages are vendor-specific, others are standardized but may come with a plethora of vendor-specific dialects.

**Batch vs Online Processing** There are typically two modes in which a mainframe can operate: in *online* mode and in *batch* mode. Online mode implies interaction with users, which typically takes the form of transactional processing supported by infrastructure such as IMS (IBM’s Information Management System) or CICS (IBM’s Customer Information Control System). Batch mode, on the other hand, implies non-interactive processing used for processing data in *bulk*. These batch processes usually run at night or during the weekend and or controlled by scripts written in JCL.

**Mainframe Environments** An example of the rigid structures in mechanistic organizations is their development and deployment process which is technically visible in the different mainframe environments that are used. As shown in Figure 3.5 (p.26), there are typically three to four environments corresponding to different stages in the software development process. The *DEV* environment is used by developers to create, correct and test individual programs. The *TST* environment is used for developer integration testing. To reduce costs, the *DEV* and *TST* environments are sometimes merged into one environment. After developer testing, the programs are typically promoted to the quality assurance or pre-production environment (QA), i.e., the
environment used for acceptance testing by business. This environment usually contains a mirror of the next production release which will run on the production environment (PRD) used by the end-users. As such, both environments are *alive* using real data and communicating with all third-party systems. All significant changes pass through these three (or four) stages, no stage is skipped. The entire deployment process is driven by a very strict release cycle, with typically four releases per year.

In conclusion, while mainframes will not win you over for their good looks and are very bad at number crunching and graphics, they are currently unmatched for commercial data processing. Nevertheless, in order to survive, they too must adapt to changing requirements, which will become increasingly more difficult as knowledge disappears with people leaving or retiring. Hence, the vital importance of reverse engineering on the mainframe in years to come.

### 3.7 Summary

Organizations in the financial services industry have a mechanistic culture favoring plan-based development processes. As such they promote labor specialization resulting in clearly delineated tasks and impose rigorous processes supported by standardized tools to maintain centralized control.

Mainframes provide the perfect infrastructure to support commercial data processing in this mechanistic culture. They clearly implement the organizational structures and rigorous processes, allowing for little flexibility. Several mainframe environment with different characteristics are set up to support the typical plan-based development process. Furthermore, because of their age, mainframes also come with various languages and dialects.

We have identified two areas in which we believe these organizations can benefit from reverse engineering techniques: during prestudies for evolutionary development and during routine maintenance. As such, the following roles can benefit the most from reverse engineering: (i) *prestudy leaders* when studying the feasibility or estimating the cost of a proposed project, (ii) *technical designers* when confronted with technical questions from a prestudy leader or when deciding whether a change request is to be considered routine maintenance, (iii) *maintainers* when asked to implement a change to a part of the system they do not know very well.
Chapter 4

Systems under Study

To err is human – and to blame it on a computer is even more so.
— Robert Orben

This chapter gives an overview of the three systems from two different organizations we have studied in the context of this dissertation. For each system we will briefly discuss its purpose and list some technical characteristics. Additionally, we will position each system within the software life-cycle stages as presented in Section 2.3 (p.10) and discuss whether the system is still undergoing evolutionary development or only routine maintenance. Lastly, we will relate some incentives to perform reverse engineering to each system, based on the incentives as provided in Section 2.2 (p.9). An overview and comparison of the three systems under study is provided in Table 4.3 (p.31).

4.1 Document Generation and Management System

Purpose  Courier is a Document Generation and Management System that enables an insurance organization to (i) define, check and manage different types of documents which are to be sent in some way to their customers, and (ii) gather and sort these documents so that they can be printed and sent as efficiently as possible. The focus of Courier is on the documents that make up most of the organization’s mail-flux and is not intended for the use of very specialized, one-time documents. To accomplish the above requirements, Courier can be used interactively via mainframe terminals and through batch jobs for bulk processing. Furthermore, domain-specific applications can be built for other systems to make use of Courier. As such, it is set up as a general purpose tool, i.e., it does not care about the kinds of documents it processes, nor what they contain. It achieves this by letting the documents be defined through parameters in a database, describing everything from the high-level specification to the low level details (including text, required inputs, etc.).
CHAPTER 4. SYSTEMS UNDER STUDY

Technical  Courier is a Cobol application with small parts in assembler and glued together with JCL scripts for batch processing. It consists of 200k lines of COBOL code with embedded SQL (including whitespaces and comments), divided over 401 COBOL programs and 336 COBOL copybooks, and 38k lines of JCL. It runs on an IBM z/OS mainframe using IMS for transaction processing and DB2 as database management system. ASF is used for typesetting, which can only compose documents in batch. The maintainers use PVCS and Mainframe Express for their routine maintenance tasks. It provides source management, version management and locking of files. Compiling takes place on the maintainers’ machine, after which the binary is placed onto the mainframe.

Software Life-Cycle Stage  The system is almost 20 years old and has survived more than one merger and acquisition by the organization. There is one very experienced maintainer, the original developer, who practically knows the system by heart, but who is about to retire. He has three other people in his team that are less experienced and have been working for ± 5 years on Courier. A lot of documentation is available, though not in a standardized format, without clear categories and most of it out of date. Although the original developer is still actively involved, the system is a stereotypical example of a system in the servicing stage. The code has become difficult to evolve and the technology is too outdated to support new business requirements. As such, current changes are limited to routine maintenance in the form of bug fixes and patches.

Incentive  Business wants to integrate a live preview functionality into Courier, but maintainers resist because the application was built for bulk-processing. More importantly, the use of AFS as typesetting facility prohibits the live preview because AFS only supports composing documents in batch. As such, the entire Document Generation and Management System will be rewritten in JAVA, enabling the integration of a live preview and, at the same time, dividing their application into components which can potentially be outsourced afterwards. The migration to this new system will be gradual in the sense that Courier will be phased out and run in parallel with the new system. Other systems can choose to use the new system or continue using the old, but no new features will be added to Courier anymore. Nonetheless, knowledge about Courier will be important to build the new system, and in the meantime, Courier still needs to be maintained.

4.2 Portfolio Management System

Purpose  The Portfolio Management System is part of a large Belgian bank and provides the back-end for keeping track of all financial possessions of a client (e.g., securities and savings) in a so called portfolio. As such, the Portfolio Management System collects information from a lot of other systems within the organization. The goal is to assist clients in managing their portfolios, which it does based on the client’s profile. Clients can create, inspect, modify and regroup their own portfolios and even simulate buy and sell actions to see the effect on a certain portfolio. Important back-end features include calculating the real value of possessions (i.e., how much are my shares worth at this moment) and calculating how certain possessions are spread over a portfolio (e.g., to perform some risk analyses).
4.2. PORTFOLIO MANAGEMENT SYSTEM

Technical  The system is coded in two different 3GL languages, namely APS and VAG, from which COBOL code is generated that runs on an IBM z/OS mainframe. APS has a very COBOL like syntax but abstracts away implementation details thereby focusing on the business functions. VAG doesn’t resemble COBOL as much as APS, it has a more functional syntax but still maps easily onto COBOL. Past development has always been in APS, while all the new development is in VAG, i.e., all subsystems are a mixture of both APS and VAG. This is technically possible because both languages invoke programs by their COBOL name, which is the same in both languages. The dependencies between those programs are automatically kept up to date and stored in database tables that can be accessed through a tool called CROSSREF or queried directly using SQL.

The system consists of more than 1.8M lines of VAG code and more than 1.2M lines of APS code, distributed over 2343 executable COBOL modules of which 1230 are interacting with other modules in the system. The other 1113 are stand-alone modules such as isolated maintenance modules (used once to fix something in a release) and stand-alone batch programs.

The change and release management is tracked by Changeman, transaction processing is provided by IMS and the database management system is DB2. Furthermore, a tool called XPEDITER is used for analyzing, testing and debugging mainframe application. It provides standard debugging functionality like setting breakpoints and watching variables.

Software Life-Cycle Stage  The Portfolio Management System is clearly in the evolution stage, it is a very important and profitable system for the organization. As such, about ten people are performing routine maintenance and over 30 are actively developing new features. There are two very experienced people within the maintenance team: one is an analyst within the maintenance team with a developer background, as such he has both a functional and a technical perspective on the system, the other one is the lead maintainer, who knows all the technical details of the system. At the same time, three domain architects are actively maintaining and evolving the domain architecture in which the Portfolio Management System plays an important role.

Documentation is provided in the proprietary tool MEGA and theoretically spans from the functional level to the executable entities (programs, subprograms, functions). In practice, however, there are gaps. Partial redocumentation efforts have been conducted in the past. Routine maintenance and evolutionary development is documented, but there is a historical backlog which is not easily processed.

Incentives.  Because the Portfolio Management System is being actively evolved as one of the most important system within the organization, it regularly introduces new concepts which are later reused within other system of the organization. In other words, the system introduces new — or changes existing — corporate standards. As such, some features within the system will eventually have to be extracted into a separate enterprise-wide service reusable within the organization. Furthermore, domain architects are actively exploring ways to expand the scope of their domain. As such, they act as leaders in several prestudies to investigate the feasibility of using this system in other European countries, whereas now it is only used in Belgium. Currently, they gather their information only from the technical designers. Reverse engineering can be used as an extra source of information to take the system itself into account as well.
CHAPTER 4. SYSTEMS UNDER STUDY

4.3 Custody Services Back-end

Purpose OSIRIS provides the Custody Services Back-end for a large Belgian Bank. In short, custody is a service consisting in holding and administering securities on behalf of third parties [Chan et al., 2007]. The New Oxford American Dictionary defines a security as “a certificate attesting credit, the ownership of stocks or bonds, or the right to ownership connected with tradable derivatives”. In early days, these certificates where papers stored in vaults, nowadays all this information is stored, checked and traded electronically.

Technical OSIRIS is implemented in CA GEN, a Computer Aided Software Engineering (CASE) application development environment marketed by Computer Associates\(^1\). CA GEN was originally known as the CA GEN CASE TOOL (Information Engineering Facility\(^2\)). Developers can create entities, procedures, procedure steps and action blocks in the CASE tool using both visual models and textual code. As such, not all information is retrievable from plain source code. However, CA GEN stores all its information in a proprietary database format which can be interactively queried via the CE-ACCESS tool or directly using SQL queries. From this repository, COBOL code and DB2 database schemas are generated which are compiled and deployed on an IBM mainframe running z/OS where IMS is used for transaction processing and DB2 as a database management system. Although the bulk of the system is created with CA GEN, some modules are coded in PL/1 to provide communication mechanisms with other systems. Furthermore, JCL is used to control all the batch processing.

Software Life-Cycle Stage Over the year OSIRIS has reached a stable state, both technically and functionally. As such, it is clearly in the servicing stage where the bulk of routine maintenance is concerned with legal and fiscal adjustments. However, over the years, it has grown to contain too much functionality and technologically it is drawing its last breath, running on a completely isolated mainframe environment which constitutes both a high risk and a high cost. Furthermore, documentation is rather limited. There has been an effort to document the current system, but it has never been a priority. The little documentation that is available is not used and not kept up to date. The maintainers get most of their work done using the CE-ACCESS tool.

Incentives. The system is currently undergoing a technical conversion for two reasons. First, it aims to move away from the CA GEN environment, which will no longer be supported, and replace it with EGL, a platform-independent, business-oriented language that is being introduced throughout the entire organization. Second, it aims to move away from an outdated mainframe environment which is currently only being supported to run this one system created in CA GEN. On top of that, the current and future maintenance activities are being off-shored to India. As a prerequisite, sufficient documentation needs to be available.

\(^1\)http://www.ca.com/
\(^2\)http://en.wikipedia.org/wiki/Information_Engineering_Facility
<table>
<thead>
<tr>
<th>Platform</th>
<th>Document Generation and Management System</th>
<th>Portfolio Management System</th>
<th>Custody Services Back-end</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IBM’s z/OS with IMS and DB2</td>
<td>IBM’s z/OS with IMS and DB2</td>
<td>IBM’s z/OS with IMS and DB2</td>
</tr>
<tr>
<td>Languages</td>
<td>COBOL, JCL, ASSEMBLER</td>
<td>APS, VAG, JCL</td>
<td>CA GEN, PL/I, JCL, ASSEMBLER</td>
</tr>
<tr>
<td>Size</td>
<td>200K lines COBOL</td>
<td>1.2M lines APS + 1.8M lines VAG</td>
<td>21M lines COBOL generated from CA GEN</td>
</tr>
<tr>
<td>Software Life-Cycle Stage</td>
<td>servicing stage moving towards phase-out: only routine maintenance</td>
<td>evolutionary stage: both routine maintenance and evolutionary development</td>
<td>servicing stage: only routine maintenance</td>
</tr>
<tr>
<td>Team Size</td>
<td>4 maintainers</td>
<td>10 maintainers, 30 people working on development projects</td>
<td>1 local + 6 off-shore maintainers</td>
</tr>
<tr>
<td>Incentives</td>
<td>new business requirement + outdated technology</td>
<td>changing business requirements + introducing corporate standards</td>
<td>unsupported technical platform + adhering to corporate standards</td>
</tr>
</tbody>
</table>

**Table 4.1:** Overview Systems under Study
CHAPTER 4. SYSTEMS UNDER STUDY
Part II

Techniques
Machines take me by surprise with great frequency.
— Alan Turing

Software Views

Software views are seen as an essential tool in an analyst’s toolbox, among others used to assess an architecture’s fitness for purpose (i.e., against new or changing requirements) [Clements et al., 2002]. Furthermore, visual system representations are particularly well-suited during the so-called First Contact phase of a reverse engineering project, to complement traditional techniques such as code reading and interviews [Demeyer et al., 2008, Chapter 3]. Indeed, a visual representation — if done well — provides the ability to comprehend large amounts of data [Ware, 2000]. A prestudy leader asked to investigate the feasibility of certain changes on a system will first have to familiarize himself with the system, create an initial understanding of the entire system and investigate certain structural properties in detail. This is where well-chosen views on the software system can be a tremendous aid.

The software views in this dissertation only incorporate the visualization of structural properties, i.e., the static building blocks of the system under study. We will not take into account behavioral (run-time) or historic (version) information.

This chapter has four major components. First, we identify the polymetric views by Lanza and Ducasse [2003] as a promising reverse engineering technique by explaining the basic principles and highlighting its use within different stages of comprehending a software system. Second, we try to apply this technique in the context of COBOL mainframe systems in Section 5.2 (p.37), and try to study its usefulness in Section 5.3 (p.41). Thirdly, we identify a number of software views useful in different stages of the comprehension process in Section 5.4 (p.49). By complementing these views with concrete, anecdotal information, it can be used as inspiration for practitioners. Lastly, we conclude with the lessons we learned by applying software views on various mainframe enterprise applications in Section 5.5 (p.58).
5.1 Polymetric Views

Polymetric views have been used as a technique to present the structural decomposition of a system emphasizing areas of interest using metrics [Lanza and Ducasse, 2003]. Each polymetric view requires a layout (the position of nodes and their relationships), a set of entities (which parts of the system are shown and which are not) and a set of metrics to shape the nodes. Figure 5.1 shows how up to five metrics can be visualized on one node. Furthermore, Lanza and Ducasse [2003] have grouped their wide variety of polymetric views into categories based on the kind of unknowns the analyst wants to resolve: First Contact views for gaining a first overview of the size, complexity and structure of the system under study, Inheritance Assessment views for studying the structure and use of inheritance in object oriented systems, Candidate Detection views for identifying potential structural problems, and Class Internal views for studying the internals of important classes, e.g., the ones resulting from the candidate detection.

Polymetric views provide what we expect from a good reverse engineering technique. The First Contact views are ideal as a starting point for understanding the system, even in the absence of an expert. The Candidate Detection views focus on certain anomalies, which may serve as a basis for formulating hypotheses and can further be investigated by using Class Internal views. But most importantly, the technique is highly interactive (customizing views, switching between them as necessary and letting the visual clues guide your understanding), thereby promoting incremental and iterative analysis. Moreover, the researchers testify that they have applied polymetric views on several industrial applications ranging from a SMALLTALK framework of approximately 3,000 classes (600 kLOC) to a system of approximately 1.2 Million lines of C++ code, strengthening our believe that it will be usable in practice.

Nevertheless, the original polymetric views are geared towards object-oriented systems visualizing classes, methods and inheritance trees, and using metrics like class size, number of invocations and number of accesses. When applying them on traditional mainframe systems, we have to visualize other entities (e.g., for COBOL: files, programs, copybooks, . . . ) and only use metrics like lines of code (LOC), fan-in and fan-out.

Analogous to the categories as proposed by Lanza and Ducasse [2003], we propose three stages in the comprehension process that can benefit from different kinds of software views.

- First Contact to quickly get an idea about the software system and its environment.
- Initial Understanding to refine those ideas by studying the structure and complexity of the
5.2. POLYMETRIC VIEWS FOR COBOL MAINFRAME SYSTEMS

This section is based on our experience with applying polymetric views to gain an initial understanding of COBOL code [Van Geet and Demeyer, 2007]. This initial understanding entails an initial overview of the structure and complexity of the system, and identifying exceptional entities for further investigations. As such, we focus on high-level functional and data dependencies among COBOL programs.

In what follows we explain the COBOL artifacts we take into account and how we extract them, the resulting software views and how we visualize them and some pointers on how to interpret these software views.

5.2.1 Fact Extraction

To characterize functional and data dependencies in COBOL source code, the following artifacts are of interest.

- **A program** is a functional block of COBOL code uniquely identified by a Program-ID. Programs are the basic building blocks of a COBOL system.
- **A copybook** is a reusable snippet of COBOL code, contained within one source file, that usually consists of a data declaration to be shared by different programs.
- **A CALL statement** is responsible for invoking a program from within another program using its unique Program-ID. The thread of execution is then passed on to the called program until the execution ends and control returns to the calling program.
- **A COPY statement** is responsible for copying the contents of a copybook directly into the COBOL source. This construct enables code level reuse of data declarations.
- **A missing program or copybook** is a program or copybook that is referenced from within the system under study but not part of the system itself. They have been identified as IDs used in CALL and COPY statements which we could not map to available source files.

From these source code artifacts we extract the following dependencies.

- **A functional dependency** is a relationship between two programs which is implemented by a CALL statement in the first program referencing the second program. This dependency has a weight associated with it equalling the number of CALL statements from the first to the second program.
A data dependency is a relationship between a program and a copybook implemented by a COPY statement in the program referencing the copybook. This dependency can also occur between two copybooks.

Recall that there are two usage modes of mainframe programs, on for online processing and one for batch processing. In online mode, programs typically interact via COBOL calls; in batch, these programs are usually controlled via JCL scripts (where JCL stands for Job Control Language). We do not take into account the JCL artifacts, thus we are missing some dependency information.

Besides these dependencies, we also define two metrics on a COBOL program.

- The FAN-IN measures the number of incoming calls, i.e., the total number of CALL statements within the entire system referencing that program.
- The FAN-OUT measures the number of outgoing calls, i.e., the total number of CALL statements within that program to any other program.

We study these metrics because they are an indicator for the types of usage of COBOL programs. Programs with a high FAN-IN can be seen as suppliers as they supply a functionality that is popular among other programs. Programs with a high FAN-OUT, on the other hand, can be seen as clients as they use a lot of functionality from other programs. The terminology of suppliers and clients has been adopted from [Mitchell and Mancoridis, 2006].

We extract this information from the COBOL sources using Lexical Fact Extraction (pattern 8.2) only. We have created a Perl script using simple regular expressions implemented in the standard UNIX tool GREP to extract all source code artifacts. Extraction of the COBOL artifacts and creation of the data model for the Document Generation and Management System was completed in less than five seconds indicating that the approach is very lightweight and likely to scale well on even bigger systems.

5.2.2 Visualizing Dependencies

We have created the following views to represent a part of the software system as a directed graph:

(i) a Functional Dependency View containing COBOL programs as nodes and the functional dependencies between them as a directed edge between two nodes,
(ii) a Data Dependency View containing COBOL programs and copybooks as nodes and the data dependencies between them as a directed edge between two nodes, and
(iii) a Mixed Dependency View containing both functional and data dependencies.

We also provide internal variants to the functional and data dependencies, which filter out the missing programs and copybooks, thereby shifting the focus of the view from interaction with other systems to the internal structure of the system under study.

To obtain such visualizations, we export the structural information to GDF (Guess Data Format) after which it can be visualized in GUESS [Adar, 2006], a flexible graph exploration tool. This environment assists a user in exploring graphs by providing capabilities such as applying graph
5.2. POLYMERIC VIEWS FOR COBOL MAINFRAME SYSTEMS

layouts, highlighting, zooming, moving and filtering. Furthermore, we use the Graph EMbedder (GEM) algorithm [Frick et al., 1994] as a layout algorithm because it positions highly connected nodes closer together thereby providing a visual indication of possible clusters and groups.

Unless stated otherwise, we adhere to the following visual characteristics: programs are represented by rectangles and copybooks by circles; functional dependencies are represented by gray edges between programs, data dependencies by gray edges between a program and a copybook or between two copybooks; internal programs and copybooks are white, whereas missing programs and copybooks are black. Furthermore, the height and width of the nodes are relative to the FAN-IN and FAN-OUT metrics of the corresponding program. The size of copybooks is directly proportional to its use by programs.

5.2.3 Interpreting Visualizations

In what follows we identify some visual indicators of structural properties and group them according to a more specific task than initial understanding. We motivate the task, present visual indicators and interpret them in light of the specific task.

A. Get a feel for the overall design

Motivation Functional dependencies reveal first hand information on the complexity of the design. Identifying strongly interdependent groups of programs can reveal opportunities or risks early in a project.

Visual Indicators (Figure 5.2) Isolated nodes (a) have no edges, whereas a monolith (d) has an abundance of edges between all nodes. Disconnected clusters (b) are isolated groups of interconnected nodes, whereas with connected clusters (c) the groups are connected by few edges.

![Figure 5.2: Overall Design Indicators](image)

Interpretation Assuming that the functional dependencies we extract are the only way to invoke a program, isolated nodes and small disconnected clusters would reveal dead code. On the other hand, if there are other functional dependencies which are not part of our model, these nodes might reveal clearly separated functional blocks. Connected but loosely coupled clusters indicate functional blocks that are easy to separate.

Relevant Views These indicators are visible in the functional dependency view.
CHAPTER 5. SOFTWARE VIEWS

B. Identify Exceptional Entities

Motivation  Detecting exceptional entities helps in quickly gaining focus when faced with a lot of information. In this case, locating clients and suppliers can help decide where to look first.

Visual Indicators (Figure 5.3)  Clients are visually represented as very wide nodes (a) whereas suppliers can easily be identified as very tall nodes (b). A combination of both results in a tall and wide node (c).

![Illustration of client, supplier, and client-supplier nodes](image)

Figure 5.3: Exceptional Entities

Interpretation  Suppliers, characterized by a high FAN-IN, indicate a popular functionality in the system. In the best case it could be an interface into an important functionality, although it might as well be a smell indicating poor separation of concerns. Clients, characterized by a high FAN-OUT, indicate more high level functionalities. Typically, (graphical) user interfaces exhibit such characteristics. A combination of both client and supplier possibly indicates an interface into lower level functionality (thus, making it a supplier) but delegating the real work to the core programs (thus, making it a client).

Relevant Views  While the FAN-IN and FAN-OUT metrics are solely based on functional dependencies we choose to visually represent the programs like this in all views. This way we can more easily recognize programs in the data views which we already identified in the functional views.

C. Characterize Data Usage

Motivation  Unsurprisingly, commercial data processing systems involve a lot of data. As such, data dependencies can provide clues to the relevance of data structures in a certain context. Identifying common data dependencies can expose groups of data structures used for a similar purpose, whereas identifying data facades show how data usage within and across systems is separated.

Visual Indicators (Figure 5.4)  A central data concept is visually represented as a group of programs directly connected to the same copybook (a), whereas common data is represented as a set of copybooks all connected to several programs (b). A data facade, on the other hand, is represented as one program connected to a set of copybooks which are only connected to that program. If the copybooks are mainly missing copybooks we call it an external data facade (d), otherwise it is an internal data facade (c).
5.3. Validation

Interpretation  As opposed to grouping programs by shared functional dependencies, we can also group them according to shared data dependencies. Programs sharing data are likely to work on a similar concept, especially when the data represents a business concept. On the other hand, programs monopolizing data are likely to encapsulate certain data and/or the corresponding functionality, making them more easy to separate.

Relevant Views  These indicators or visible on both the data dependency views. Of course, you will not find External Data Facades in an data dependency view that has filtered out all missing copybooks. On the other hand, the Central Data and Common Data indicators can also occur with missing copybooks.

5.3  Validation

To validate the usefulness of polymetric views as just presented in Section 5.2 (p.37), we report about a case study in which we apply the polymetric views on the Document Generation and Management System, which we introduced in Section 4.1 (p.27). After generating the functional and data dependency views for the system under study, we write down our observations solely based on these views and their assumed interpretation. Next, we attempt to validate the comprehension technique by verifying the reported observations against external sources such as code documentation and knowledge of the maintainers.

The main goal of this study is to see whether and how these proposed polymetric views are useful for gaining initial understanding of a COBOL system. We refine this goal into a number of subsidiary research questions:

RQ1  Can we discover the structural decomposition of the system?
RQ2  Can we identify key programs based on lightweight metrics?
RQ3  Can predefined visual indicators help us understand specific parts of the system?

5.3.1  Observations

Functional Dependency View  The first thing we notice in Figure 5.5 (p.42) is the typical monolithic structure of the legacy system. Almost everything seems to be interconnected with no apparent order or hierarchy. There are some isolated nodes on the bottom and the right (A) that might constitute dead code, as they have no functional dependencies. The small disconnected
Figure 5.5: The Functional Dependency View. Height and width of the nodes represent FAN-IN and FAN-OUT respectively. Areas of interest are manually annotated and magnified.
Figure 5.6: The Data Dependency View. Edges only represent data dependencies. Areas of interest are manually annotated and magnified.
clusters on the right (B) show a distinct pattern: one parent node functionally depending on three child nodes. Which could indicate a recurring form of delegation. When looking for exceptional entities, the most pertinent supplier is the tallest white node (S1). Others are annotated as well. Besides the white suppliers there are also missing programs (e.g., S4, S5 and S6) that are classified as suppliers. Client C1 is connected to several missing programs, whereas C2 is connected to a lot of programs within the system. One last node that really stands out is the big white rectangle (D) classified as both a client and a supplier, indicating a popular interface into lower level functionality.

**Internal Functional Dependency View**

For a clearer view, we also looked at the internal variant of the Functional Dependency View by removing all missing programs. Doing this breaks down the monolithic structure into two loosely coupled clusters. In line with the observations of Mitchell and Mancoridis [2006], we filter out the biggest supplier to break down the view even further. This produces some more disconnected and loosely coupled clusters, as depicted in Figure 5.7 (p.44).

**Data Dependency View**

Besides the monolithic structure in Figure 5.6 (p.43), there is one group of nodes that is clearly separated. This cluster (E) reveals a Central Data Concept used by several programs. Furthermore, client C3 has many data dependencies with copybooks that have no dependencies with other programs, therefore it acts as an internal data facade. Because it is classified as a supplier, we can deduce that it has a considerable amount of functional dependencies as well. Furthermore, client program C1 depends on a lot of missing copybooks. When looking at group1 of copybooks, C1 acts as a Data Facade for the external data. The copybooks from group2, on the other hand, are also used by other programs, thereby apparently violating the facade property. In the Functional Dependency View (Figure 5.5), C1 is connected to several missing programs; this similarity leads us to assume that C1 uses external programs to process external data.
5.3. VALIDATION

Figure 5.9: Extract from the Mixed Dependency View. Programs are denoted by white rectangles, the copybook by a white circle. Functional dependencies are black edges, data dependencies are gray edges.

**Internal Data Dependency View** The internal variant of the Data Dependency View, obtained by filtering out all missing copybooks, revealed some programs participating in the Common Data indicator, which refined our understanding on the structural decomposition of the system. Extracts are shown in Figure 5.8 (p.44).

**Mixed Dependency View** When looking at the Mixed Dependency View to find more similarities between functional and data interactions, we reencountered the Central Data Concept (E) from the Data Dependency View, but here the involved programs exhibit the behavior of the disconnected clusters (B) from the Functional Dependency View. However, what strikes as odd in Figure 5.9 (p.45) is the fact that the program groups have no incoming calls.

5.3.2 Validation Setup

To validate the proposed visualization technique, we consult three alternative sources of information to verify whether they confirm the observational statements made for the system under study. First, for those programs that struck out as matching the visual indicators introduced in Section 5.2.3 (p.39), we performed code inspection (with a focus on the source comments). Second, we interviewed the lead maintainer of the system. Third, we composed a questionnaire targeting the maintainers of the system. The results of these three sources are combined to address the research questions that we initially posed.

**Documentation** We had access to the source code of all 401 programs of the system. To verify our observations about certain programs we mainly focussed on the semantic value of the source comments (one line per program explaining its purpose). On occasion, we also conducted brief code inspection on a specific program to gain more structural understanding. We also had access to the sources of the copybooks, however, we did not use them as they did not contain any comments and were very cryptic.
CHAPTER 5. SOFTWARE VIEWS

Interview We confronted the lead maintainer with our observations after we verified them using the documentation. We posed questions such as ‘Are these programs part of the same subsystem?’ and ‘This looks like an Internal Data Facade but the documentation does not support that observation, which is it?’.

Questionnaire The questionnaire consisted of two major parts. In the first part the maintainers were asked to give their top-10 of key programs in the system. We described ‘key programs’ as the programs they would explain first to a novice when asked to comprehend the system, e.g., (i) programs fulfilling a crucial functionality, (ii) programs they have to modify often or (iii) programs they use as entry point when starting a maintenance action. In the second part we confronted them with five groups of programs which we selected from the internal views (Figures 5.7, 5.8 and 5.9). For each group, they were asked to state whether or not those programs were related, i.e., whether they interact to fulfil a particular function within the system or whether they are part of the same subsystem. If so, we asked them to briefly clarify the particular function or subsystem. Besides the content questions, we also asked them to quantify their experience with the system (#years) and which parts of the system they are more familiar with than others. The questionnaire was offered to, and filled in by, all four maintainers of the system.

5.3.3 Data Results

From the documentation we gathered that the programs participating in the Central Data Concept in Figure 5.9 (p.45) and the Common Data indicators in Figure 5.8 (p.44) also match in the comments, with recurring terms in all programs like ‘DLM’, ‘fiscal’ and ‘statistics’. However, the programs in the clusters from the Internal Functional Dependency View in Figure 5.7 (p.44) did not contain any similarities in comments.

Investigating the comments of exceptional entities revealed two big suppliers (S1, S3) as read interfaces for databases, whereas S2 is responsible for generating anomaly reports for checks on a specific database. Client C1 is responsible for retrieving data from external databases, C2 checks the consistency of documents, C3 is responsible for preparing documents for printing/sending. The comments of program D, indicated as both client and supplier, revealed it as the interface into the entire system under study. This would explain its classification as client since it invokes all functionality within the system, but not as supplier.

Code inspection of the programs from the delegation clusters (B) revealed a switch statement in all the top programs checking the value of a flag in a data record and delegating responsibility to one of his three child programs. Furthermore, the comments of the child programs show a recurring pattern of the terms ‘insert, delete and modify’.

The interview learned us that each child node in the delegation clusters (B) is responsible for either inserting, deleting or modifying data in a specific table of the documents database. And that a row in such a table is uniquely identified by the data in the Central Data Concept (E). This backs up former evidence about the DLM subsystem. Furthermore, the interview revealed that the top programs in clusters B are called dynamically (via the supplier program in Figure 5.9 (p.45)) based on information stored in a database, which explains why they have no incoming calls in our views,
5.3. Validation

Table 5.1: Key Programs

<table>
<thead>
<tr>
<th>Program</th>
<th>agreement</th>
<th>FAN-IN</th>
<th>FAN-OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>JCLGEN</td>
<td>4/4</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>D</td>
<td>3/4</td>
<td>52</td>
<td>72</td>
</tr>
<tr>
<td>C2</td>
<td>3/4</td>
<td>2</td>
<td>42</td>
</tr>
<tr>
<td>FuncClustIA</td>
<td>3/4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>C1</td>
<td>2/4</td>
<td>0</td>
<td>64</td>
</tr>
<tr>
<td>Online/Batch</td>
<td>2/4</td>
<td>7</td>
<td>37</td>
</tr>
<tr>
<td>MainBatch</td>
<td>2/4</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>Batch</td>
<td>2/4</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>FuncClustIB</td>
<td>1/4</td>
<td>0</td>
<td>43</td>
</tr>
<tr>
<td>Unknown</td>
<td>1/4</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>Unknown</td>
<td>1/4</td>
<td>8</td>
<td>19</td>
</tr>
<tr>
<td>Unknown</td>
<td>1/4</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>Batch</td>
<td>1/4</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>Online/Batch</td>
<td>1/4</td>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>Unknown</td>
<td>1/4</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>Online/Batch</td>
<td>1/4</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>Batch</td>
<td>1/4</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Batch</td>
<td>1/4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>S1</td>
<td>1/4</td>
<td>410</td>
<td>2</td>
</tr>
<tr>
<td>Batch</td>
<td>1/4</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.2: Agreement on Clusters

<table>
<thead>
<tr>
<th></th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFDV 1</td>
<td>partially</td>
<td>yes</td>
<td>partially</td>
</tr>
<tr>
<td>IFDV 2</td>
<td>partially</td>
<td>partially</td>
<td>no</td>
</tr>
<tr>
<td>Fiscal</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Statistics</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>DLM</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

which are solely based on static information.

When asking about exceptional entities, the lead-maintainer explained that C1 is responsible for automatically filling in documents with information from different databases outside the system under study, supporting former evidence from the documentation. The violation of the facade property by group2 of copybooks in Figure 5.6 (p.43) is less crucial because other programs need access to some of the same tables for retrieving access rights of users, which is certainly not the kind of information that C1 is looking for. Supplier and client D is indeed explained as the interface into the entire system. While this is in conflict with the observation of its high FAN-IN, this is explained by a lot of initialisation calls from two helper programs to get the user interface up and running. Furthermore, the missing suppliers are identified as either general core in-house programs (S5, S6) or core mainframe utilities (S4).

From the questionnaire we learned that the lead maintainer has more experience than the three others (17 years vs. 4 or 5 years). This is also reflected in the fact that all maintainers have their own area of expertise within the system, whereas the lead maintainer is familiar with all parts. This phenomenon — typical in a mechanistic organization — has a big impact on the results of the questionnaire.

Table 5.1 aggregates the results of the top-10 key programs into 20 identified key programs with their corresponding maintainer agreement. As we expected, the results varied between different maintainers, because they are all responsible for specific parts of the system. There was, for example, only one program on which they all (4/4) agreed.

For the cluster verification, one of the maintainers only made descriptive remarks about the possible functions of the clusters instead of answering with yes or no. Rather than interpreting her remarks
we decided to discard her results. Table 5.2 depicts the maintainer agreement on the five clusters and shows three groups that were unanimously verified as clusters. Two of those three clusters were clarified as separate subsystems that are only organisationally under the control of the maintenance team, whereas the third one was identified as one of the functionalities of the Document Generation and Management System itself. The two first groups, on the other hand, were not verified as clusters. The typical responses were (i) breaking up the group in smaller functional clusters and (ii) indicating programs that were not related.

5.3.4 Interpretation

RQ1 Can we discover the structural decomposition of the system? While both the documentation and the questionnaire agreed on the three clusters that resulted from the Data Views, indicating that our technique helps in discovering structural decomposition, they also disagreed on the two clusters revealed from the Functional Views. This can be explained by the batch nature of the system in which each program already has a clearly defined purpose. After that, they are composed in a chain of actions by JCL scripts that are not part of our data model. Furthermore, this system also uses a dynamic calling mechanism for increased flexibility. This limits the usefulness of the functional dependencies for the case study presented here.

In summary, our technique helped to discover structural decomposition with the added remark that, based on this case study, the Data Dependency Views look more promising than the Functional Dependency Views. Dependencies from the JCL scripts should be added to make the latter views more useful.

RQ2 Can we identify key concepts based on lightweight metrics? Looking at the metric values in table 5.1 we see a lot of variation for the FAN-IN, the maximum value (410) is present but also several programs with a FAN-IN of 0, indicating that the FAN-IN is a bad indicator for key programs. The FAN-OUT, on the other hand, is less variable and has higher values for the key programs with higher maintainer agreement, indicating that the FAN-OUT is a better indicator than the FAN-IN.

We used these results also for quantifying the precision and recall of our technique. As baseline we used the key programs on which more than one maintainer agreed (the top 8 from table 5.1). For both FAN-IN and FAN-OUT we generated a list of programs sorted according to their metric value. Since we do not want to introduce a fixed threshold on the FAN-IN and FAN-OUT values for determining whether a program is key, we repeatedly calculated the precision and recall of a growing set of retrieved programs to find the optimal threshold. However, the best result we received was a precision of 25% and a recall of 50% when retrieving the top 16 clients, for the suppliers the precision did not even reach 10%.

As such, the raw metrics are not a clear indicator for detecting key programs. We attribute this to the fact that the information provided by the metrics is too limited and too low level, revealing technical bottlenecks only. Other metrics should be taken into account as well. Still we argue that the combination of these metrics with the proposed views are useful. Many of the exceptional entities we identified in the views — not only by their metric value, but also by their interaction with
other programs — are also identified as key programs by the maintainers (cfr. boldfaced programs in table 5.1). And if not for identifying key programs than for identifying structural bottlenecks in the visualizations that can be filtered out to result in a more comprehensive view.

**RQ3** Can predefined visual indicators help us understand specific parts of the system. As with RQ1 it is clear that the Data Usage visual indicators helped us understand the corresponding subsystems. While the Overall Design indicators were better suited for interpreting the general characteristics of the system, some indicators have been very useful in gaining better technical understanding of specific parts, for example, the interplay between the delegation clusters (B) in the DLM subsystem. Similarly, the Exceptional Entities were better suited for very technical understanding, especially in combination with the other visual indicators.

Of course, these visual indicators are of limited usefulness in isolation, interaction with domain experts is necessary to assign the correct interpretation to the indicators. In particular, the level of interaction between artifacts is near impossible to extract from such abstract visualizations alone. As such, we can conclude that, for this case study, the predefined visual indicators were — although not sufficient on their own — helpful in understanding parts of the system. This is of course in line with the iterative and incremental nature of reverse engineering in general and comprehension in particular.

### 5.4 Using Software Views

Where the previous sections report in detail on the application and usefulness of polymetric views for initial understanding adjusted to our context of mainframe COBOL systems, this section reports briefly on applying several software views in various stages of the comprehension process.

The goal of this section is to illustrate the iterative nature of system comprehension, how software views support this iterative nature and how different views are relevant in different stages of comprehension. Furthermore, the different experiences described in this section can be used as inspiration by practitioners willing to use software views in practice.

For each experience we do the following.

(i) We state the stage of the comprehension process we are in (first contact, initial understanding, studying internals) and with it the intent of the activity based on the problem at hand.

(ii) We list the facts we extracted and how we extracted them.

(iii) We explain the composition of the views and the tools and techniques we used to construct them.

(iv) We argue that it is a good view or technique by showing some anecdotal evidence based on the Document Generation and Management System.
5.4.1 Treemapping as First Contact

When an outsider is asked to look at a software system, she typically receives a CD/DVD containing source code, documentation and other related artifacts with little or no explanation. Her first contact with the system should be brief, yet revealing enough to determine whether something can be done with it and how to proceed. To assist during this first contact, Demeyer et al. [2008] list some interesting patterns, such as chatting with the maintainers (pattern A.1) and reading all the code in one hour (pattern A.2). Of course, reading all the code is impossible; the idea is to get a first impression on the quality of the code [Demeyer et al., 2008, chapter 3].

To keep the code reading somewhat coherent and to pinpoint some locations to start reading, a visual perspective on the received code is very helpful. Treemapping can provide such a visual perspective. It is a method for displaying tree-structured data by using nested rectangles.

A lot of treemapping software exists taking various kinds of inputs ranging from spreadsheets over databases to xml files. We used the tool TREEMAP as it is able to take a folder structure as input. Furthermore, it provides metadata about the file type, size and age which can be used to interactively filter and layout the treemap. Hence, it can be used to create a very specific kind of polymetric views.

Figure 5.10 (p.51) shows an example treemap that represents the source code — as we received it — from the Document Generation and Management System. There are several folders (the biggest ones being SRC, JCL and CPY) each containing several files all represented as rectangles. Furthermore, the size of the rectangles is proportionate to the file size and the color is mapped to the file type (based on the extension of the files). The rectangles are spatially sorted according to file size (top left to bottom right) while still adhering to their containment relationship.

Note that files on mainframe do not have file extensions. Rather, they are categorized in data sets according to their file type. The extensions were added to indicate this type when downloading the files from mainframe. As a consequence, the resulting folder structure reveals no architectural information, something that is common in languages like JAVA and C++ where the folders mirror the package structure. Nonetheless, this visual perspective can guide a first contact.

For example, the smaller folders mostly contain one file type, i.e., the JCL folder contains JCL-files, the CPY folder contains COBOL copybooks, the PROC folder contains PROC-files, etc. The SRC folder, on the other hand, contains several types of files: COBOL files (.CBL), COBOL files with embedded SQL (.SQB), ASSEMBLER files (.ASS). Furthermore, there are some other files which are responsible for database, batch and transaction configuration.

In the JCL folder, there are two overly large JCL files. Chatting with the maintainers (pattern A.1) about this phenomenon triggered an interesting discussion about the difference between online and batch processing of documents and how this is implemented. Most processing is done in batch, which is executed once or twice a day. Each paper format has a separate JCL script because they

\[\text{http://en.wikipedia.org/wiki/Treemapping}\]

\[\text{http://en.wikipedia.org/wiki/List_of_Treemapping_Software}\]

\[\text{http://www.cs.umd.edu/hcil/treemap/}\]
are sent to different kinds of customized printers optimized per paper format. Urgent and smaller sets of documents can be requested interactively (i.e., online processing). They are processed every two hours and cannot interfere with the batch jobs and their printers. This is implemented in two JCL scripts which have to be able to process all paper formats.

Mapping this experience to the characteristic of a reverse engineering technique, we can say that this polymetric view on the source code in the form of a treemap is ideal to be used as a starting point for understanding, without prior knowledge about the system. Furthermore, it reveals unknowns. Of course, this is not that difficult without prior knowledge, but the visualizations represent something concrete you can take to the expert to ask a specific question, like we did with the two JCL files. Lastly, the treemapping technique is very fast and interactive, allowing quick customizations based on the metadata recovered from the file system.

5.4.2 Polymetric Views for Initial Understanding

After the first contact with the system and its users, you have an idea about the software system. The next step is to refine this idea into an initial understanding to support further reverse engineering activities [Demeyer et al., 2008, chapter 4]. To refine our understanding of the system, we decided to focus on the COBOL code, as this accounted for the bulk of the system. As we explained in Section 5.2 (p.37), we set out to extract functional and data dependencies and visualize them in a
meaningful way.

We used Lexical Fact Extraction (pattern 8.2) to extract programs, copybooks, CALL relations and COPY relations from the COBOL sources, which was a very lightweight approach robust against various dialects and likely to scale well on even bigger systems. Using these facts, we created several views highlighting functional and data dependencies within the system and in interaction with other systems.

For anecdotal evidence we refer to Section 5.3 (p.41). But again, we experienced the power of interactivity: switching between different views, filtering out some unnecessary parts and chatting with the maintainers for feedback about newly revealed unknowns.

5.4.3 Resolving Issues During View Creation

Before obtaining these polymetric views for initial understanding, we had to overcome several issues with our data input. One of these issues was a problem with the mapping between COBOL Program-IDs and COBOL files. We noticed this issue after an interview during demo (pattern A.3). Several COBOL programs that had been mentioned by our interviewee were missing from our views. Furthermore, some programs did appear in our views, but we could not find any reference to them in the source listing.

To identify the problem, we decided to use the same infrastructure to visualize the relationship between the COBOL source files we received and the COBOL Program-IDs they contain. The blue dots in Figure 5.11 (p.53) represent the source files whereas the orange bars represent the Program-IDs; the edges represent containment. On the bottom left, there is one source file that contains many Program-IDs. This construction adheres to the COBOL standard as one compilation unit can contain several programs. On the bottom right, on the other hand, there are several source files containing the same Program-ID. This should not be possible because it means that different programs are using the same name. Higher in the figure, there are even some examples of source files sharing several Program-IDs. Clearly something is wrong here.

When confronting the maintainers with these findings, a story emerged about the Great Migration. As a result of a merger and acquisition several years ago, the system had been migrated to a different technical platform. This new platform did not use the COBOL Program-ID as identifier for its programs, instead it used the name as provided to the change management environment without checking consistency with the COBOL Program-ID statements. Of course, developers did not bother to change the Program-ID anymore when they copied an existing program to start a new one, resulting in several programs with — apparently — the same name. On the other hand, the COBOL construct of having source files with several Program-IDs wasn’t even in use before the migration. That is, either these particular parts of the code were not used anymore, or some obscure assembler code was used to jump to a specific position in the file, without using the Program-ID as a language construct. The conclusion of this exercise was that we should use the names of the source files (as provided by the change management environment) to identify a COBOL program instead of the Program-ID as defined in the COBOL standard.

This experience illustrates the benefit of having a visualization environment to rapidly create a
Figure 5.11: Overview of the COBOL files and the Program-IDs they contain.
Although the views, as studied in Section 5.3, resulted in Figure 5.14, we decided to filter out the exceptional dependencies (weight \( \leq 3 \)) to unclutter the view. This is a clear example of the incremental and iterative nature of system comprehension, where creating views of different perspectives of the system as needed aid in resolving unknowns.

### 5.4.4 Clustering for Initial Understanding

Although the views, as studied in Section 5.3 (p.41), contained some interesting visual indicators, the one that stands out is the apparent monolithic structure of the overall design of the system. In an attempt to find some structure in what seems to be a mess, we turned to the program names. As you can see in Figure 5.12 (p.54), they seem to be adhering to some kind of naming convention. All names have the prefix, most likely denoting the Document Generation and Management System.

The first few names have a \( \text{PG} \) as prefix, while all the others have a \( \text{PG} \) in the fourth position, while all the others have a \( \text{W} \). Based on these kinds of patterns, we manually grouped the programs into clusters and aggregated all the functional dependencies, resulting in Figure 5.13 (p.55).
5.4. USING SOFTWARE VIEWS

**Figure 5.13:** Functional Dependency View of Document Generation and Management System after clustering based on naming conventions.

**Figure 5.14:** Functional Dependency View of Document Generation and Management System after clustering based on naming conventions and removing all *exceptional dependencies.*
Figure 5.15: Schematic illustration of possible layering within the Document Generation and Management System.

Figure 5.16: Functional Dependency View of the Document Generation and Management System after filtering out the two base layers.
A manual sketch of the possible layers in Figure 5.15 (p.56) clearly shows that the PG0T and the PG0WX clusters form the basis for the entire system. This also leads us to believe that the programs in these clusters are the ones that cause the monolithic nature of our Functional Dependency View. Thus, to get a clearer picture of that view, we filtered out all programs from those two layers. The resulting view in Figure 5.16 (p.56) shows nicely colored clusters with more clearly separated interactions between the clusters. This indicates that the naming conventions do have some structural meaning.

This experience shows how the creation of visualizations (in this case the Functional Dependency View) can trigger the need for more facts gathering (the name-based clusters). Feeding these new facts (color of the clusters) and refined knowledge (filtering out the bottom layers) back into the original visualization creates a clearer picture which enables more focussed understanding.

5.4.5 Studying Exceptional Entities

Based on this uncluttered Functional Dependency View in Figure 5.16 (p.56), we started to investigate some exceptional entities in more detail. Instead of only looking at the code, we opted for a lightweight visual representation of the internals of one COBOL program. To accomplish this, we used the research prototype COViTo [Dejonghe, 2007], created as part of a master’s thesis at the University of Ghent. The visualization engine is a combination of two other research prototypes: SANDBOX and RUNYA. Of particular interest for us is RUNYA [De Schutter, 2006] as it can create a visual representation of the control flow of one COBOL paragraph.4

To be able to visualize control flow, one has to first extract the control flow from the code. To do so, COViTo uses an island-based parser, which has since then evolved into a stand-alone Cobol Parser Generator called KOOPA5, that can parse large parts of the COBOL85 standard. It can handle the typical hierarchical structure of COBOL with its divisions, sections and paragraphs, but also control flow statements like call, perform, and goto. However, to get it to process our code from the financial services industry correctly, we had to make several adjustments to the parser grammar, e.g., to fix the identification issue as explained in Section 5.4.3 (p.52). Luckily, the tool is open source and we could cooperate with the original KOOPA developers to fit it to our needs. This way, we were able to make COViTo visualize the control flow of our system under study.

As an example of this control flow visualization, we revisit the Central Data Concept and its participating programs from Figure 5.9 (p.45). As you can see, these groups of four programs

---

4 This work has been inspired by CFLOW: http://www.gnu.org/software/cflow/
5 http://koopa.sourceforge.net/
EVALUATE CPARAM OF OPARAM
   WHEN '1'
      OPEN OUTPUT F-DD10
      WRITE ENR-DD10 FROM IMP1
      WRITE ENR-DD10 FROM IMP2
   WHEN '2'
      WRITE ENR-DD10 FROM ENR-OUTP
   WHEN '3'
      CLOSE F-DD10
END-EVALUATE.

Figure 5.18: Control Flow Visualization of the top and bottom program of Consistency Checking Cluster.

reappear in the uncluttered version in Figure 5.16 (p.56), all in the same color, indicating that they have been clustered together based on their name. This is another indication that these programs do in fact serve a common goal. Figure 5.17 (p.57) shows the control flow of the top program of one such group. As you can see, it are merely three branches checking a parameter (the little hills). Based on this parameter, the program calls one of the child-programs (the diamonds) to either insert, delete or modify something in the database. After the call returns, a jump to an exit paragraph is performed (the upward arrow).

As an other example, we take the gray cluster on the top of Figure 5.16 (p.56). As you can see, the hierarchy of programs contains a lot of clients (wide programs) in the middle with one big supplier (tall program) on the bottom and a small program on the top. Figure 5.18 (p.58) reveals that the top program is delegating all the work to the bottom programs. Again, several branches (represented as hills) are visible, but this time with two calls (two diamonds) per branch. The first call is always to a client (denoted by the green annotations) while the second call is always to the supplier on the bottom (denoted by the red annotations). Skimming through the code comments revealed that all the client programs in the middle are responsible for checking the consistency of certain parts of the Documentation Database. The supplier program on the bottom, on the other hand, is responsible for writing resulting inconsistencies to an output. As you can see on the bottom of Figure 5.18 (p.58), this supplier program can be called to either (1) open the output stream, (2) write to this output stream, or (3) close the output stream. As you can see, the stream is opened once before all the checks, and then closed after all the checks have been performed.

While this level of detail might not always be necessary when trying to grasp the general structure of a system, it does lead to concise explanations of specific exceptional entities that pop up in the polymetric views. This makes it easy to (i) either discard the exceptional entity as normal, or (ii) decide to investigate it in even more detail.

5.5 Lessons Learned

Our experience with applying these various software views and visualization techniques on different mainframe enterprise applications enabled us to uncover four difficulties, which we list and explain in this section in the form of lessons learned.
5.5. LESSONS LEARNED

<table>
<thead>
<tr>
<th>Type of System</th>
<th>Languages Involved</th>
<th>Fact Extraction (only for bold-faced languages)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document Management</td>
<td>COBOL, JCL, ASSEMBLER</td>
<td>lexical fact extraction using GREP</td>
</tr>
<tr>
<td>Portfolio Management</td>
<td>APS, VAG, JCL</td>
<td>changeman cross-reference information</td>
</tr>
<tr>
<td>Custody Services Back-end</td>
<td>CA GEN, PL/I, ASSEMBLER</td>
<td>querying integrated CASE tool database</td>
</tr>
</tbody>
</table>

Table 5.3: Fact extraction methods we used.

5.5.1 The Cost is in the Languages

Over the past three years, we have studied various systems involving different languages. An overview is presented in Table 5.3 (p.59). We have used software views on all these systems for various purposes. The first step in creating these views, however, is extracting the necessary facts to visualize. Clearly, this is a challenge in an environment with that many languages and little standardization. We discuss several ways of tackling this issue in Chapter 8 (p.107) on Fact Extraction Patterns. Not having the budget to buy a Commercial Fact Extractor (pattern 8.3) or the resources to build a full-fledged fact extractor for each of these languages, we looked for other ways to extract facts.

For the Document Generation and Management System, the company had no fact extractors available for any of the languages involved. As explained before, we use very lightweight Lexical Fact Extraction (pattern 8.2) on the source code of the most heavily used language (in this case COBOL). However, as the case study progressed, it became increasingly clear that the most interesting dependencies were the ones embedded in the JCL scripts (used for batch processing) essentially hiding dependencies between existing COBOL modules. This illustrates the need for cross language dependency extraction, something most commercial tools don’t provide and certainly an interesting avenue for further research.

The second case tackled Portfolio Management which is implemented with a mix of APS (1.2M lines) and VAG (1.8M lines). Here, we did not extract dependencies from the source code. Instead we Gathered Available Facts (pattern 8.1) from the mainframe change management environment that keeps track of language-independent cross-reference information. However, while this saves the cost of implementing fact extractors for each language, it severely limits the information available for visualization.

In the context of the Custody Services Back-end, we have studied a CA GEN system (21M lines of generated uncommented COBOL + 67k lines of commented PL/I). Since CA GEN is more a CASE tool than a programming language, it does not store its “code” as plain text but rather in a database accompanied by an integrated environment. Here we Gathered Available Facts (pattern 8.1) by querying this database.

Although we were able to obtain sufficient results for our cases using these cheaper work-arounds, we only had access to a limited amount of information. Unlocking the full potential of this technique will require a significant investment in infrastructure, not only extracting information for each language but also extracting cross-language information. It remains to be seen whether this is economically viable in the presence of many different languages, especially because cross-language dependency extraction is a largely unexplored area of research.
CHAPTER 5. SOFTWARE VIEWS

5.5.2 Nothing Works out-of-the-box

An off-the-shelf fact extractor will not work out-of-the-box. Whether it is a commercial or an open-source fact extractor, there will always be some oddity in your language variant or system setup that is different from all others. Ira Baxter, CEO of Semantic Designs inc.\textsuperscript{6}, calls this the BIBSEH phenomenon: ‘Because In Big Systems Everything Happens’. Solving these everythings requires ad-hoc work-arounds.

One example involves the DLM subsystem of the Document Generation and Management System discussed in Section 5.3 (p.41), where the Central Data Concept in Figure 5.9 (p.45) shows several groups of programs that are responsible for inserting, deleting or modifying data in a specific table of the documents database. However, they were classified as dead code because accessing them was handled by a dynamic call involving a look-up table. Another example is the identification issue discussed in Section 5.4.3 (p.52) where the standard way of naming COBOL programs is not adhered to. A last example involves the visualization of COBOL program internals with CoViTo. This tool has a rudimentary COBOL parser built-in to be able to create these internal representations. To get the tool to work, we had to adjust the parsing mechanism to take into account several vendor-specific oddities in the COBOL code. Luckily, we had the source code available for this research prototype so we were able to make the necessary adjustments.

Never assume it will work. On the contrary, always assume it will not. Hence, when preparing the use of an existing package, make sure you are able and authorized to make the necessary changes as you go.

5.5.3 Dinosaurs Don’t Like Pictures

As we stated before, the mainframe — a.k.a. the dinosaur of computing — is not a visual environment. As such, we also had some interesting experiences with the usage and adoption of the polymeric views in the context of mainframe organizations.

Lanza and Ducasse [2003] envisioned their polymeric views to show its full potential in an integrated environment, i.e., being able to easily switch between visual entities and corresponding code. This way, maintainers would get the most value out of these visualizations. However, in our experience, they are not interested in such ‘fanciness’, they would rather compose some textual query and dive into the code. We believe this lack of interest can be attributed to the limited graphical capabilities of the mainframe dinosaur. Therefore, integrating these visualizations with the development environment will only be possible if development does not take place on the mainframe itself. Prestudy leaders, on the other hand, are usually not that familiar with the mainframe and do not require access to the code. They can benefit from these kinds of visualizations as objective structural information they can use in addition to interviewing the experts (pattern A.3).

\textsuperscript{6}http://www.semdesigns.com/
5.5.4 Knowledge is Scattered and Scarce

Lastly, we noticed that getting empirically sound validations of our findings is extremely difficult “in vivo”, mainly because knowledge about the system is typically either centralized in one person or scattered among different people each with their own expertise. This prohibits cross-validation of opinions as there is usually at most one “expert” per part of the system. For all practical reasons, this stresses the importance of reverse engineering tools that can assist in verifying the knowledge of experts.

5.6 Summary

In this chapter, we have shown that the polymetric views technique is a good reverse engineering technique. It can support knowledge acquisition in various stages of comprehension. It can be used during first contact to get an idea of the software system, or to refine that idea into an initial understanding and even to investigate some parts in more detail by studying the internals. To enable this, the technique is highly interactive, allowing for customized views and switching between them as necessary to formulate hypotheses and afterwards reject or verify them. This is the very essence of iterative and incremental exploration.

However, in applying these views on mainframe enterprise applications we have uncovered some difficulties. The many languages and dialects make it difficult to perform the data gathering activity, using off-the-shelf fact extractors — whether they are commercial or open-source — is not a quick fix to this problem. Furthermore, sound empirical validation of knowledge acquired through visualizations is difficult because knowledge in these mechanistic organizations is scattered and scarce. However, the biggest issue to the adoption of software views on mainframe systems is the lack of visual support on the mainframe. As such, maintainers will not be able to benefit fully from these visualizations.
Chapter 6

Feature Location

*In theory, theory and practice are the same. In practice, they are not.*
— Laurence Peter “Yogi” Berra

Feature location is the act of mapping features (i.e., functions a system can perform) to pieces of code implementing that feature. It was first introduced by [Wilde and Scully, 1995] and it is now well-studied in the literature. It helps to resolve so-called “known unknowns”, i.e., the things that we know we don’t know but by acknowledging this lack of information we can look for an answer. As stated before, an analyst is also interested in identifying and resolving the “unknown unknowns”, i.e., things we do not know we don’t know. These unknowns are much harder to address since we can only acknowledge them once we have encountered them and they require a more exploratory analysis of a given system. A feature location technique supported by an iterative and incremental process is able to address both requirements.

By cooperating with the financial services industry, we have identified several recurring problems in which such a feature location technique can help.

— Introducing new, enterprise-wide, shared services will replace existing features within current software systems. Locating their implementation is necessary to be able to disable (or remove) the features. The prestudy leader can use this information to estimate the impact that the integration of the new service will have on the existing system.

— Documentation about certain (or all) features of legacy system can be lacking. This is a problem when trying to maintain a system or when the need arises to rebuild an existing system. Since code is the most up-to-date form of documentation, this is what we need to re-document the feature. Feature location can help look for this feature-specific code.

— A new developer joins a project or service team and needs to familiarize himself with (a part of) the system. Studying the behavior of the system while interacting with it induces a more profound comprehension of the system and sheds new light on available documentation.
CHAPTER 6. FEATURE LOCATION

Feature location looks very promising to solve the recurring problems stated above, but of course, we first need to investigate whether such techniques can be applied in our “in vivo” context of mainframe systems from the financial services industry, which will be the focus of this chapter. In Section 6.1 (p.64) we select a promising feature location technique from the literature and conduct a replication experiment in which we apply the technique in our context and try to replicate the findings of the original authors. In Section 6.2 (p.77), we provide a summary of the lessons we learned from the replication experiment. Taking into account the shortcomings of our setup in the replication experiment, we list some attempts to use a different setup in Section 6.3 (p.79). All these attempts failed or were cancelled in the end. Yet, the causes of these failures provide valuable information for practitioners interested in setting up a similar experiment.

6.1 Replication Experiment

The industrial context for this replication experiment is a large Belgian bank that needs to analyze their systems to identify which parts can be reused in other (European) countries. A prestudy leader performing such an analysis is interested to know whether the code implementing a given feature is located in a few separate modules (in which case reuse becomes feasible) or scattered across many modules (in which case reuse becomes difficult). Using the survey of Cornelissen et al. [2009], we looked for a feature location technique that was also good for system comprehension and that had been reported to work on an industrial scale. The survey indicated one technique that satisfied all the criteria, namely the feature location technique described by Eisenbarth et al. [2003], which uses formal concept analysis to analyze run-time information.

In what follows, we report on an experiment where we try to replicate the findings of Eisenbarth et al.. In Section 6.1.1 (p.64), we describe the principal steps of the technique, we identify its key strengths and argue why it is a good reverse engineering technique. In Section 6.1.2 (p.67) we list the explicit and implicit assumptions that the technique makes about the system under study. Section 6.1.3 (p.67) reports about a pilot experiment where we verify whether we can replicate the technique on a small scale. In addition, the section serves as a clarifying example of the technique. Next, we shortly revisit the industrial context in which we are working, derive the technical implications this has on the techniques to capture execution profiles. Capturing these execution profiles is part of the data gathering activity which is crucial for this feature location technique. After describing the experimental set-up in Section 6.1.4 (p.69), we report on the different iterations we needed to map the given set of features to a reasonable amount of code modules in Section 6.1.5 (p.71). Lastly, we list the aspects of the technique that worked well and the aspects that could be improved in the “lessons learned” in Section 6.2 (p.77).

6.1.1 Technique

In a nutshell, Eisenbarth et al. [2003] present a semi-automatic technique that combines static and dynamic analyses to rapidly focus on the system’s parts that relate to a specific (set of) feature(s). These system parts are called computational units and are defined as executable parts of a system,
such as routines, modules, classes and components. A feature is defined as a realized functional requirement of the system, more specifically, an observable behaviour that can be triggered by the user. As a result of the technique, the feature-unit map describes which computational units implement a given set of features. However, invoking a feature on a system can only be done by providing an adequate trigger. Therefore, a scenario is defined as a sequence of user inputs that triggers actions of a system with observable results. Thus, scenarios invoke features which are implemented by computational units.

The technique differentiates itself from related work mainly by allowing a many-to-many mapping between scenarios and features. Thus, a scenario can invoke many features, and a feature can be invoked by many scenarios. Consequently, using formal concept analysis, one can now differentiate between general and specific computational units with respect to a given feature. For example, a computational unit can be very specific to one scenario which invokes one feature, or it can be very generic, being executed by all scenarios and thus not specific to one feature. Similarly, one can also identify distinctly and jointly required computational units for a given set of features.

The process of feature location as described in [Eisenbarth et al., 2003] is depicted in Figure 6.1 (p.65). It starts with the creation of scenarios for a given set of relevant features. For this, we need a domain expert that knows which actions trigger a certain feature. This results in a set of scenarios and a scenario-feature map describing which scenarios invoke a certain feature and vice versa. In the dynamic analysis step, which is further refined in Figure 6.2 (p.66), the system is first instrumented and recompiled to obtain execution profiles, then the scenarios are executed by a user. The resulting execution profiles are the instantiation of a scenario-unit map describing which...
computational units are executed by each scenario. In step 4, the analyst uses these execution profiles to create a concept lattice. Informally speaking, this lattice representation maximally groups computational units executed by the same scenarios and vice versa. The analyst can then start interpreting the relation between scenarios and computational units, and (s)he can overlay the lattice with the scenario-feature map (resulting from step 1) to interpret the relation between features and computational units; thereby creating a feature-unit map.

However, this resulting feature-unit map is inherently hypothetical because it is based solely on the runtime information captured using only a small subset of a — potentially infinite — set of possible scenarios. Therefore, this information is validated and corrected by the analyst in a subsequent static analysis step (after a preparing step in which a static dependency graph is extracted from the system). Thus, the main goal of steps 3 and 4 is to narrow down the candidates for the manual inspection in step 5.

To allow for incremental analysis, feedback loops are foreseen in three key areas: (i) after initial interpretation (step 4) to adjust technical parameters of dynamic analysis, such as the granularity, (ii) within the concept lattice interpretation step, depicted in more detail in Figure 6.3 (p.66), to incrementally ‘grow’ the set of considered scenarios or the set of considered computational units when confronted with an unfamiliar system, and (iii) after analysis (step 4 or 5) when it becomes apparent that more discriminating scenarios are needed to clearly isolate the relevant features. This iterative approach, embedded in the process, is key to any good comprehension technique to resolve
6.1. REPLICATION EXPERIMENT

(A1) The technique is only suited for features that can be invoked from the outside.
(A2) The scenario-feature map is clear from the start.
(A3) The system can be recompiled or instrumented to allow recording of execution profiles.
(A4) The system can run in isolation.
(A5) No operator needed during dynamic analysis.
(A6) In principle, there is no limit to the granularity of a computational unit.

Table 6.1: Overview of Assumptions

both “known unknowns” and “unknown unknowns”. As such, it adheres to the requirements of a good reverse engineering technique as we stated them in Section 2.6 (p.13).

6.1.2 Assumptions

The technique makes some assumptions for its use, which are summarized in Table 6.1 (p.67). Some are stated explicitly in [Eisenbarth et al., 2003], others are implicit. First of all, some very explicit boundaries were stated for the applicability of the technique. One of them being that the technique is only suited for features that can be invoked from the outside (A1). This seems a logical prerequisite as it should be possible to create end-user scenarios for them. Then, for the first step of the process, it is explicitly assumed that the analyst knows in advance which features are invoked by a scenario, i.e., the scenario-feature map should be clear from the start (A2). This is important as all subsequent interpretation relies on this information. For the dynamic analysis step, it is assumed that one has access to the source code or the binaries in such a way that recompilation or instrumentation is possible to allow recording of execution profiles (A3). This implies that you either have access to the environment and are allowed to change it or that you can somehow lift the system from its ‘natural habitat’ and run it in isolation (A4). The latter also includes running it as “only user” to prevent noise from unrelated system usage. After having set-up the profiling infrastructure, the authors also assume that a user can perform the dynamic analysis step, implying that no operator is necessary for the technicalities of the tracing (A5). Furthermore, the authors regard the notion of computational unit as a generic parameter of the technique, i.e., in principle, there is no limit to the granularity of computational units (A6). This is important as the technique promotes refinement of granularity in combination with incremental analysis.

In this replication experiment, we will investigate whether or not these assumptions have a significant impact on the applicability of the technique.

6.1.3 Pilot Study

To dry run the technique and to verify the assumptions, we conducted a first small scale replication experiment on an ATM Simulation. This is an example system developed at Gordon College and used for teaching purposes as “a complete example of OO methodology applied to a single problem”. It consists of only 2418 lines of uncommented Java code including 43 classes and 167

1http://www.math-cs.gordon.edu/courses/cs211/ATMExample/
CHAPTER 6. FEATURE LOCATION

The system is a very simple implementation of an ATM. It provides features such as verifying a pincode and the ability to deposit, withdraw and transfer money (see columns table 6.2). These features are invokable as menu options in a GUI. Because the system is documented in great detail, we were able to use the functional test descriptions as a basis for our scenarios and for the scenario-feature map (table 6.2). Thus, for this case, assumptions A1 and A2 are valid.

As granularity for the computational units we chose the method level. We downloaded the sources and compiled them on a local server using IBM’s java SDK\(^2\). Thus, assumptions A3 and A4 are valid as well. To execute the scenarios we had to run the program with the \(-xtrace\) option. Depending on the technical skills of the user, this could invalidate assumption A5. But this is easily solved by preparing a start-up script to make the tracing transparent.

Figure 6.4 (p.68) shows the resulting concept lattice\(^3\), each circle represents a concept which is defined by a set of scenarios (a gray box) and a set of methods executed by those scenarios (a white box). In this representation, only the number of executed methods is shown. Methods of subconcepts are also part of their superconcepts, e.g., performing the withdraw \$20\) scenario has executed a total of 53 methods, of which 10 were only executed by that scenario, the other 43 (=12+11+20) are shared with other scenarios. Vice versa, scenarios of superconcepts are also part of their subconcepts, e.g., the turn ATM on/off scenario has executed 20 methods, but all other scenarios have also executed them. This is correct because we have restarted the ATM in each scenario. In general, more generic scenarios will trickle down in the concept lattice, while more specific scenarios will be found higher up. Note that there are no features in this representation. The relation between features and methods will be deduced in combination with the scenario-feature map depicted in Table 6.2.

When locating the verify pin feature, Table 6.2 shows that we should look for the methods executed by four scenarios. Figure 6.4 reveals that this would leave 31 (=11+20) possible methods.

---

\(^2\) http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/

\(^3\) created with CONCEPT EXPLORER — http://conexp.sourceforge.net/
6.1. REPLICATION EXPERIMENT

However, since we know that the turn ATM on/off scenario does not invoke this feature (also visible in table 6.2), we can discard the bottom concept and are left with 11 methods possibly implementing the verify pin feature.

For the more specific features deposit, withdraw and transfer, the analysis resulted in, 9, 10 and 7 methods specific to the respective features. Closer investigation revealed that most of these methods were part of a dedicated class implementing their feature, and that all methods of these classes were only part of their respective concept. This indicates that the implementations of these three features are clearly separated in different classes, something we would expect in an educational example of good OO analysis.

While this is by no means a full case study, it illustrates that all the assumptions for the use of the technique were valid, and indeed, the technique was applicable and gave satisfying results.

### 6.1.4 Study Set-up

The next step is to set up the technique in our ‘in vivo’ context the financial services industry. It is clear, however, that these large organizations using mainframes have some characteristics that were not taken into account by Eisenbarth et al. [2003]. Indeed, isolating one system from a centrally controlled environment with rigid procedures and responsibilities (A4) can prove to be difficult. Recompiling an entire system on another mainframe (A3) — assuming you have one available — is equally problematic. As such, the ability to extract an execution profile from a running system is endangered in our ‘in vivo’ context.

Luckily, there are ways to obtain an execution profile from a running system without completely isolating it, but this comes at a cost. In Chapter 9 (p.117), we explain some ways to obtain run-time information in the form of patterns. One could, for example, (Ab)use High-Level Monitoring Tools (pattern 9.1) to get coarse-grained run-time information or one could (Ab)use a Debugger (pattern 9.2) to obtain very detailed runtime information. Additionally, one could also try to Instrument Source Code (pattern 9.3) without isolating the system under study. Of course, one needs to balance several forces and make trade-offs when choosing a particular solution. Depending on the context, some factors will have a more prominent role than others, thereby favoring a certain solution.

One such factor is the choice of mainframe environment to obtain run-time information from. In Section 3.6 (p.24) we introduced the DEV (development), QA (quality assurance or preproduction) and PRD (production) environments. Because QA and PRD are live environments, the data quality of these environments will be very good, whereas DEV typically has very poor data quality. In
data processing systems this data quality reflects the quality of the scenarios one can create and, thus, it impacts the quality of the feature location results. On the other hand, the DEV environment will have more development and maintenance tools that can be abused to obtain run-time information, whereas QA and PRD have very strict quality of service requirements that prohibit too much overhead.

With these solutions and factors in mind, we started looking for a profiling infrastructure that would fit our needs.

As a first step, we created an inventory of existing debugging, profiling and tracing infrastructure available on mainframe (built-in or purchased and installed by the bank). As input we used scarcely available mainframe system documentation and knowledge of several expert technical operators. As a result, we obtained a diverse list of options ranging from very low-level system tracing, over hooks into language environments handling communication between different programming languages to high-level logging of module use for identifying dead applications over time. Afterwards, we added the option of instrumenting the code at several moments during the build process. Note that none of the options were ready to use ‘as is’.

Secondly, we organised a meeting with the technical experts to discuss the feasibility of each option and sorted them according to their impact given the following requirements.

(R1) The infrastructure should be usable on the QA environment to ensure sufficient data quality.
(R2) The granularity should be at least at module level.
(R3) System and organisational impact should be minimal.
(R4) The production environment should, under no circumstances, be endangered or modified.

The combination of requirements R1 and R4 ruled out all the options involving code instrumentation (pattern 9.3) because internal policy dictates that the QA environment is a mirror of the next production release. Therefore, all instrumented binaries would ripple through into production. While this is not a technical limitation, we did not have the political power to change this policy. The combination of requirements R3 and R2 made us abuse the high-level monitoring tool (pattern 9.1) DORANA4, as it provides a high-level, system-wide logging of load modules. A load module is the compiled binary of a COBOL program, henceforward called module. Using DORANA introduced no extra impact on the systems because it was already running in the background, whereas the impact of the more detailed system tracing tools was at that time unknown.

When stating requirement R2, we expected this would be sufficient for our feature location purposes because these banking systems are created using functional decomposition with entities like main functions and sub-functions. Since both are encapsulated in their own COBOL module, a profiling tool with module granularity is able to discriminate between the two. However, should we need finer granularity, e.g., during incremental analysis, DORANA will no longer be sufficient.

After having chosen the DORANA tool, we had to customize the set-up for it to provide correct input data for the dynamic analysis step. The main problem we encountered here was the isolation of scenarios on the mainframe. As mentioned in Section 3.6 (p.24), one of the strengths of mainframe is the ability to simultaneously support thousands of users. DORANA keeps track of

---

6.1. REPLICATION EXPERIMENT

how many times each mainframe user accesses each module. Thus, having a dedicated mainframe
user to execute scenarios would provide all the necessary information to filter out our execution
profiles. Unfortunately, most systems running on mainframe are calculation intensive back-ends to
various front-ends running on other systems. Interaction with mainframe is typically handled by a
transaction manager, such as IMS\(^5\) or CICS\(^6\), which runs as a dedicated transaction manager user.
Therefore, we lack all user information in the DORANA logs. Clearly, the assumption that one can
isolate herself or operate the system as only user (A4) is not valid in this context. Consequently,
we had to look for other ways to identify our execution profiles from the complete DORANA
log.

As a first countermeasure, we used transaction isolation. A transaction, used by the transaction
manager, is defined by a main function (essentially an entry-point into the back-end). All trans-
actions starting from such an entry-point can be technically isolated to run in a separate region,
which makes them identifiable in the logs. Furthermore, all calls to subsequent sub-functions
originating from this entry-point will execute in the same region because they are part of the same
transaction. However, this presumes that the domain expert knows in advance which entry-point(s)
will be used by each scenario. This is a risk, as we are limiting the search space of the dynamic
analysis using potentially incomplete information. Thus, we will have to take it into account when
interpreting the results. On the other hand, isolating the correct entry-point dramatically reduces
the noise in the DORANA logs, i.e., instead of receiving all transactional activity, we can now filter
to only obtain activity from the desired entry-points. However, it is still possible that other users
are using the same entry-point at the same time.

As a second countermeasure, we minimize the timeframe in which we execute our scenarios and
reset the log before and after. Clearly, this is not a waterproof countermeasure. Luckily, we can
detect whether or not the logs have been polluted by comparing the number of times each main
function has been accessed with the number of times it should have been accessed when executing
the scenarios. If the numbers do not match, the logs are polluted and the scenarios should be
executed again.

Similar to isolating our use of the system, we should also be able to distinguish between our
different scenarios. Currently, we can only do this by resetting the log between each scenario.
Unfortunately, this can only be done by technical operators with sufficient privileges who are not
even in the same geographical location. This clearly invalidates assumption (A5), making it a more
labour intensive and error-prone step due to the resulting synchronization overhead.

6.1.5 Portfolio Management System

After having set-up the profiling infrastructure, we were able to apply the feature location technique
on the Portfolio Management System as introduced in Section 4.2 (p.28). In short, the system
provides the back-end for keeping track of all financial possessions of a client (e.g., securities and

\(^5\)IMS is IBM’s premier transaction & hierarchical database management system. It simply stands for Information
Management System.

\(^6\)CICS is a family of application servers from IBM which provides online transaction management and connectivity. It
stands for Customer Information Control System.
CHAPTER 6. FEATURE LOCATION

savings) in a so called portfolio and managing this portfolio based on the client’s profile. Clients can create, inspect, modify and regroup their own portfolios and even simulate buy and sell actions to see the effect on a certain portfolio. Important back-end features include calculating the real value of possessions (i.e., how much are my shares worth at this moment) and calculating how certain possessions are spread over a portfolio (e.g., to perform some risk analyses).

The system is coded in two different 3GL languages, namely APS and VAG, from which COBOL code is generated that runs on an IBM z/OS mainframe. Since there is a one-to-one mapping between APS/VAG modules and COBOL modules, the transformation preserves the module structure and our “granularity at module level” requirement (R2) remains valid. The system consists of more than 1.8M lines of VAG code and more than 1.2M lines of APS code, distributed over 2343 executable COBOL modules of which 1230 are interacting with other modules in the system. The other 1113 are stand-alone modules such as isolated maintenance modules and stand-alone batch programs.

Comparing these numbers to the industrial cases used by Eisenbarth et al. [2003], we may conclude that our system is larger, but that the scale of our concept analysis will be smaller due to coarser granularity of our computational units (COBOL modules versus C functions).

Features The bank is currently conducting an architectural prestudy for a Global Portfolio Management System, which should extend the scope of the current system to other European countries. This is not just a technical matter because different countries have different regulations and market conditions, which is why the entire Portfolio Management domain architecture is being reevaluated. One of the building blocks of such a domain architecture are Business Logic Services, i.e., functionalities that can be grasped and used by business people and are implemented as a feature with observable behaviour. Since this matches the definition given in Section 6.1.1 (p.64), the technique should be applicable.

For this case, the architects were particularly interested in the feature for calculating the real value of possessions (henceforward called value feature) and the feature for calculating the spread over a portfolio (henceforward called spread feature), because these features won’t change in this global context and, thus, could potentially be reused. As the domain expert of the system was confident that he could identify the modules implementing these features, it was a good opportunity for us to test the technique in the field for the first time. He stated the following expectations:

(E1) Both the value feature and the spread feature are located in a few modules (about 10), i.e., the implementation is not scattered all over the system.
(E2) Both features work closely together but are cleanly separated, i.e., they have their own modules.

Scenarios All scenarios we created could be performed via a web-interface, which constitutes one of the interactive front-ends for which the system under study provides the back-end. This web-interface allows for several ways to access a portfolio, such as inspecting it, asking for a report about it or simulating changes in it, all potentially invoking the value and spread feature.
As with most web-interfaces, much clicking is involved, which increases the chances of making mistakes while executing the scenarios.

Some specifics of the system were known up front, which helped in creating the scenarios and interpreting the results. For example, the back-end is steered by a meta-model, which allows for disabling certain features in the system. Furthermore, the interactive part of the system uses a back-end caching mechanism that prevents calculating and retrieving data more than once a day per user, except when a change occurs in the portfolio. Using all this information we have conducted three iterations, in each iteration correcting, refining and adding scenarios. For the first two iterations, the scenario-feature map is depicted in Table 6.3 (p.73).

<table>
<thead>
<tr>
<th>it</th>
<th>name</th>
<th>scenarios</th>
<th>description</th>
<th>features</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>value</td>
<td>spread</td>
</tr>
<tr>
<td>1st</td>
<td>ptf</td>
<td>inspect portfolio</td>
<td>x</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>ptf-sprd</td>
<td>inspect spread of portfolio</td>
<td>x</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>2nd</td>
<td>ptf</td>
<td>inspect portfolio</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ptf2</td>
<td>inspect portfolio again</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>report</td>
<td>generate report</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>report-sprd</td>
<td>generate report with spread</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

Table 6.3: Scenario-Feature Map Iteration 1 & 2

6.1.6 Iterations

For the first iteration, we restricted ourselves to two scenarios. The first to simply inspect a portfolio, which would trigger the calculation of the real values, and the second to inspect the spread of the portfolio, which would trigger both the value and the spread feature. It is not possible to create a scenario that only triggers the spread feature, because the real values are necessary to calculate the spread of possessions on a portfolio. Nonetheless, with this feature location technique, the modules should break down into a superconcept and a subconcept, thereby differentiating between the two features. However, as a result, we obtained only one concept (containing 51 modules), indicating that the execution profiles of both scenarios were exactly the same. This was due to the fact that the spread information was already visible on the main screen of the portfolio, making the proposed scenarios invalid. This is already a first indication that, in practice, the scenario-feature map is not clear from the start as assumed in (A2).

We have tried fixing the scenarios in two ways. First, we created a scenario that would simulate a change in a portfolio. This would trigger a recalculation of the real values, while no spread is shown on the simulation screen. Unfortunately, the result was the same, indicating that the back-end calculates both the value and spread at the same time, regardless of the front-end’s needs. Secondly, we disabled the spread option in the back-end meta-model. Unfortunately, the result was still the same, indicating that the meta-model was not working as expected. Consequently, the conclusion of the first iteration was that we could not separate the value and the spread feature from each other using interactive scenarios, therefore, we focussed the next iterations on refining the location of the combination of both features (henceforward called value+spread).
For the second iteration, we reduced the set of 51 potential candidates by adding scenarios that would not invoke the value+spread feature. We did this by exploiting the caching mechanism.

We executed four scenarios as listed in table 6.3: inspecting a portfolio for the first time (ptf), inspecting it again using the cache (ptf2), generating a standard report for the portfolio (report) and a report with spread information (report-sprd) both using the cache.

Figure 6.5 shows the resulting concept lattice where the modules for calculating value+spread should be located in the concept of the ptf scenario, because all other scenarios do not recalculate the cache. The number of modules has been reduced to 36, indicating that we have narrowed down our search space. As can be expected, the ptf2 concept is a subconcept of the ptf concept and the implementation for the cache reading feature will be in the modules of the bottom concept because all scenarios invoke it. Furthermore, the report scenarios concept is a top concept, which can also be expected as it should contain a lot of specific functionality for generating a report, but in the end it gets its information the same way as the other scenarios do. What is less expected, though, is the emergence of the concept containing five modules. It indicates functionality shared when generating reports from the cache and when inspecting a portfolio for the first time. Further investigation revealed that we, unintentionally, switched the natural language (from Dutch to German) while generating the report. This means that some language-dependent data can no longer be accessed from the cache and will have to be retrieved again. This sloppiness reminds us of the error-prone nature of executing scenarios and stresses the need for strict and structured documentation in the scenario creation phase.

Since 36 modules was still too much for the value+spread feature (E2), we conducted a third iteration in which we increased the number of scenarios. Significant variability we added was (i) the use of empty portfolios (which should not trigger a calculation), (ii) the difference among real and simulated possessions, (iii) a second inspection using the cache, and (iv) a third inspection with an explicit recalculation. Taking into account all possible combinations we executed a total of 39 scenarios. This led to two problems. First, not all scenarios were executed correctly. Some were incorrect because of errors with isolating a transaction. Others were incorrect due to synchronisation and mis-communication issues between the user executing the scenarios and the technical operator in charge of the profiling (induced by invalid assumption A5). Secondly, the resulting concept lattice exploded to a total of 63 concepts, which is difficult to read all at once. To cope with these problems, we started using the incremental approach for interpreting the concept lattice. We constructed different perspectives on the lattice by filtering out different sets of
6.1. REPLICATION EXPERIMENT

Figure 6.6: Partial Concept Lattice 3rd Iteration

scenarios and we gradually merged these perspectives to gain incremental understanding. At the same time we filtered out the incorrect scenarios.

To find the modules specific to the value+spread feature, we incrementally refined the perspective to include scenarios with empty and non-empty portfolios inspected for the first (calculation), second (cache) and third (recalculation) time. As depicted in Figure 6.6 (p.75), we expected the modules to be part of the concept on the crossing of the scenarios performing the first and third inspection with non-empty portfolios, which has now been reduced to five modules. As this was more in line with our expectations of ‘about 10’ modules (E1), we stopped iterating and started validating our results.

6.1.7 Informal Validation

To validate our results, we asked the domain expert to enumerate the modules he knew to be specific to both features. He named five modules for the value feature and two for the spread feature. We located those modules in our concept lattices for each iteration and could conclude that, for the first two iterations, the modules were all in the correct concept. In the third iteration, however, only one of the seven modules was found in the correct concept. As indicated on Figure 6.6 (p.75), all other modules were found in the concept on the crossing with the caching and no-caching scenarios. This implies that the scenarios using the empty portfolio did not have the expected results.

Further investigation revealed that, for the value calculation, the main module takes the empty portfolios as zero-positions and values the contents as zero. Oddly enough, all sub-functions are also executed for an empty portfolio, but they return doing nothing. This is because the check for doing something resides in the sub-function modules and not in the main module. This indicates
that our tracing granularity was too coarse to correctly locate the value feature.

For calculating the spread model, the domain expert identified only two modules. The first one creates an empty model, which we did not find as this also happens with an empty portfolio. The second module fetches the possessions that need to be added to the model, which we did find. In addition we found three other modules, which add the possessions to the spread model, in the concept we expected all relevant modules to be (see Figure 6.6). The domain expert confirmed that these were indeed also crucial modules for the spread feature, which he did not identify. Thus, from the domain experts’ perspective we uncovered (and solved) an “unknown unknown”, but for the prestudy leader this was part of solving her “known unknown”.

6.1.8 Results

Table 6.4 summarizes the number of modules that needed to be considered after each iteration. In total 1230 modules are statically reachable from any point within the system, but only 98 are reachable from the identified entry-points used by all scenarios. After each iteration, we have reduced this number of candidate modules. The number of relevant modules, as identified by the domain expert was seven, we discovered three more relevant modules with the technique. However, in the last iteration we encountered the limits of our chosen tracing granularity and, thus, miss-located some modules. Although the granularity is, in principle, a generic parameter of the technique (A6), in practice we were limited to this granularity by our profiling infrastructure, thus, we could not easily refine it in a subsequent iteration.

Therefore, we have not been able to completely confirm the first expectation stated at the beginning of the case study (E1), i.e., with the technique we have been able to narrow down the potential location of the value and the spread feature to 36 modules before having to report false negatives due to the granularity issue. However, in doing so, we uncovered 3 modules that were not identified by the domain expert. With regard to the clean separation of the implementation of both features (E2), we were unable to construct scenarios to separate the two features because access to both features was intertwined in the interactive front-ends. This is a consequence of the first assumption (A1), i.e., we were not able to (separately) invoke the features from outside, thereby confirming that this assumption is crucial to the applicability of the technique.

We have also discovered some “unknown unknowns” from the perspective of the domain expert. First, introducing new scenarios brings new features in focus. The concept lattice automatically breaks down their implementation as well. The location of the implementation of the caching mechanism, for example, was very clear in the second iteration. Secondly, the effect of scenarios on the system was not always known in advance by the domain expert (A2), but analysis of the results provided a better understanding of the inner-working of the system. Lastly, the malfunctioning
meta-model was a big “unknown” to the maintainers.

6.2 Lessons Learned

To summarize the experience gained through this replication experiment, we list some lessons learned in the form of good practices and trade-offs. In addition, we revisit the assumptions as listed in Table 6.1 (p.67) to highlight the difficulties we encountered when applying the promising feature location technique by Eisenbarth et al. [2003] in the context of mainframe enterprise applications.

6.2.1 The Power of Iteration

One of the assumptions made by Eisenbarth et al. [2003] is that “the analyst knows in advance which features are invoked by a scenario” (A2). While this is indeed a prerequisite for scenario creation, in practice, the outcome of this step is rarely correct from the start due to the complexity of (and lack of knowledge about) these legacy systems. However, these mistakes usually translate into a very unexpected breakdown in concepts during concept lattice creation. A good practice when confronted with such an unexpected event is to first question the validity of the scenarios and only afterwards question the implementation of the system. Thus, while in our experience assumption A2 does not hold, in practice, the scenario-feature map can be incrementally corrected by refining and adding scenarios and features. In addition, we have confirmed that incremental analysis of large concept lattices is very valuable. This shows that iteration really is the key characteristic and strength of the technique.

When iteration is embedded in a technique, one should also consider when to stop iterating, a so called stop criterion. This is a tricky thing during comprehension as there are always more “unknown unknowns”. Therefore, it is essential to have clear goals at the beginning of a case study, so you can verify at any time whether you have met the goals. For feature location, the goal is simply to identify the computational units specific to a certain feature. Validation should be done manually, therefore, a good stop criterion is when the expected time needed to validate the remaining candidates manually is smaller than the expected time needed to conduct another iteration.

6.2.2 Scenarios as Sensitivity Point

As with most techniques based on dynamic analysis, the results are very sensitive to the quality of the scenarios. At the same time, the scenarios themselves (both creating and executing them) are very sensitive to errors. Hence, we call the scenarios the sensitivity point of the technique, similar to the term used in architectural analysis when a certain quality of an architecture is very sensitive to a certain architectural decision. When something changes in the sensitivity point, the corresponding quality of the architecture should be reevaluated [Clements et al., 2002].
CHAPTER 6. FEATURE LOCATION

With this technique, a small error during scenario creation or execution can have a big impact on the resulting concept lattice. In our industrial context, invalid assumptions A2 and A5 makes these errors even more likely. To minimize this risk, it is good practice to carefully record your actions during scenario execution, including all technical and domain-specific variables, and update the scenario documentation as you incrementally refine the scenarios. We conclude that these invalid assumptions have a significant impact on the usability of the technique. However, these can be alleviated with careful planning and execution of scenarios.

6.2.3 Realistic Scenarios versus Flexible Data Collection

Eisenbarth et al. [2003] discuss the trade-off between the space required to store runtime information and the desired level of detail, additionally, they argue that information overload of the analyst should also be taken into account. Therefore, based on assumption A6, they promote incremental analysis with the option of refining the runtime information for a subset of the system during incremental analysis. However, in practice, the granularity of the computational units is limited by other factors.

In our industrial context, one such limiting factor stems from the choice of environment. This poses a trade-off between the representativity of the data and the technical flexibility of the environment. As discussed in Section 6.1.4 (p.69), the closer you are to using a live environment, the better the data quality. In data-oriented environments, this is crucial for the quality of the scenarios. On the other hand, the closer you are to using a live environment, the more restricted your options will be for retrieving execution profiles.

Similar trade-offs are discussed in more detail in Chapter 9 (p.117), where we discuss several ways to obtain run-time information. For now, we conclude that, while assumption A6 is valid in principle, it has limitations in practice which can be avoided with proper planning but do have a significant impact on the applicability of the technique.

6.2.4 The Problem of Isolation

Assumptions A3 and A4 state that the system under study can somehow be lifted from its ‘natural habitat’ and can run in isolation. This is usually not the case with mainframe systems. Therefore, other techniques should be used to obtain a clean execution profile, potentially endangering the validity of the results.

We have reduced the noise by identifying and isolating the entry-points of each scenario. However, since this identification was done by the domain expert, it poses a threat to the internal validity of the results. Indeed, we are looking for all possibly relevant computational units for a certain feature, but we are telling the technique in advance where to look. Therefore, we might miss some computational units. We can work around this issue by identifying all possible entry-points either statically, via the front-end code, or dynamically, by monitoring the link between the front-end and the back-end.
6.3. ATTEMPTED ADJUSTMENTS

In addition, we have reset the log after each scenario and found a way to identify polluted profiles, so we can either discard the corrupted scenario or perform it again. However, this invalidates assumption A5 because resetting the log has to be done by a technical operator due to the strict division boundaries in large mechanistic organizations which is also reflected in the privileges.

We conclude that assumptions A3 and A4 were invalid in our industrial context, which posed a significant impact on the applicability of the technique. However, using some countermeasures, we were able to make the technique applicable in our set-up. Unfortunately, these countermeasures result in a significant overhead, thereby invalidating A5 and jeopardizing the ability to perform rapid incremental and iterative analysis.

6.3 Attempted Adjustments

After our experiences with the replication experiment on the Portfolio Management System, we have tried to apply feature location on other systems and within other organizations. Taking into account (i) the shortcomings of our set-up to extract run-time information in the replication experiment, (ii) the different analysis needs for different systems, and (iii) the different tools available in other organizations, we had to find new ways to gather the necessary data to successfully perform feature location. In this section, we list several failed attempts to do so, because the causes of these failures provide valuable information for practitioners interested in conducting similar analyses.

6.3.1 STROBE as Tracing Facility

When attempting to apply this feature location technique in another organization, the first thing we had to do was find a way to capture an execution profile from a running system. During this process we also looked for tools available within the organization which we could abuse to achieve our goals. One such tool is STROBE.\footnote{http://www.compuware.com/solutions/strobe.asp}

In short, STROBE is a performance monitoring tool from compuware that reports on how application programs use resources in a mainframe environment. STROBE employs a sampling technique that periodically takes snapshots of an applications execution and stores this data in a sample data set. From this sample data set, several reports can be generated regarding the performance of the application under study. For example, it can show which lines of code, among perhaps thousands, are using significant amounts of system resources.

By using this sampling technique, compuware has found a way to do performance monitoring with very little impact, i.e., no changes to the code or the system are necessary and performance impact is linear to the sampling rate, not to the characteristics of the application under study. As such, STROBE can be used in a production environment without disrupting normal operations. However, for our needs this sampling technique seems to be a bad fit. Sampling provides no guarantee on
CHAPTER 6. FEATURE LOCATION

recovering all computational units that have been executed. Only those computational units in which ‘enough time’ is spent will be sampled. Here, the meaning of ‘enough time’ mostly depends on the sampling rate, i.e., how many times per minute STROBE will look at the application and log which computational unit is being executed.

Clearly, this approach is insufficient if one would, for example, want to trace the execution of each program statement. However, in practice we might be able to increase the sampling rate to a point where we would be able to trace all modules being executed.

To test this idea, we set up a quick feasibility study (pattern 10.1) to see whether the resulting sample data set could be useful in practice. We created a scenario which involved using the web interface of an application and set the sampling rate of STROBE to its maximum, namely 150,000 samples per minute or 2,500 samples per second. At the same time, we traced the transactions submitted from the web interface to the mainframe back-end. Remember from Section 6.1.4 (p.69) that a transaction is characterized by one main function compiled into a module and exposed to the outside, which subsequently uses various sub-functions also compiled into separate modules. After executing the scenario, we extracted a report from the sample data set containing all the modules that had been sampled. We compared this list of modules to the list of transactions we monitored. The comparison showed us that the sample data set — not even taking into account sub-functions — missed some main functions.

Thus, using STROBE to obtain module level run-time information was not feasible on the application under study because the maximal sampling rate was too low to identify all transactions, let alone all modules. However, performing this Quick Feasibility Study (pattern 10.1) only cost us one man-day: not a high price to pay for the potential benefits of finding a built-in tool that can provide the necessary data.

6.3.2 Dynamic Feature Location at Statement Level

After conducting the experiments on the Portfolio Management System, we decided to investigate another system within the same organization, namely OSIRIS. One characteristic of the system is that it has several very large programs. Taking into account our experience with the granularity issues on the Portfolio Management System, it was clear that finer granularity was needed. Hence, we set out to perform dynamic feature location on OSIRIS with statement-level granularity by Instrumenting the Source Code (pattern 9.3).

**Solution in the small** As a first step, we set out to build a proof of concept that would be able to instrument a small part of the system, capture its execution and process the results in a meaningful way. This feasibility study (pattern 10.1) had to show us quickly whether it was possible to get to this fine-grained level of granularity before making a serious investment to apply it on the entire system.

Recall that OSIRIS is a system created with the CA GEN CASE TOOL, which generates COBOL code that is compiled and run on the mainframe. As one of the few systems within the organization, it still runs on an old technical platform. Because of its dependence on CA GEN it has not yet
6.3. ATTEMPTED ADJUSTMENTS

been migrated to the new standard platform. As a drawback, this results in even less built-in tools to use for tracing. On the positive side, it gives us a bit more flexibility as how to manipulate the environment since we don’t have to take into account impact on other systems. Furthermore, due to the nature of the system, the data is of similar quality both in the DEV environment and the QA environment. This means we can perform our experiments on the DEV environment.

Taking into account all the above characteristics, we decided our best option would be to instrument the source code (pattern 9.3) of the generated COBOL using DISPLAY statements. More specifically, each executed line of CA GEN code will be logged. This is feasible because the CA GEN CASE TOOL keeps track of the CA GEN line numbers when generating the COBOL sources. Listing 6.1 (p.82) shows two CA GEN statements with original line numbers 8 and 9, which are transformed by the CA GEN CASE TOOL to the COBOL code in Listing 6.2 (p.82). Notice the compute-statements on lines 1 and 3 of Listing 6.2 that clearly refer to line 8 and 9 of the CA GEN code by setting the value of a line counter variable. Thus, lines 4 to 17 are the COBOL counterpart of line 9 of CA GEN code. Listing 6.3 (p.82) shows how we have simply added a DISPLAY statement on lines 2 and 5, that is, after each COBOL line that sets the CA GEN line counter variable, to capture the necessary run-time information.

To perform the instrumentation of the generated COBOL source code, we use a very simple PERL script listed in Listing 6.4 (p.84). It searches for all lines containing this COMPUTE LAST-STATEMENT-NUM-statement using a regular expression and adds a line to display the necessary information. We download all generated COBOL source files from the mainframe to a UNIX machine, run the PERL script for each file and upload the instrumented source files back to the mainframe where they are recompiled. Hence, when we run the system with the recompiled binaries, output will be written to the SYSOUT on mainframe which we can collect after each scenario.

We tested our technique on a handful of source files implementing some mainframe screens with which users can set and adjust archival settings of the system, i.e., when will data be archived. It was chosen as a technical proof of concept because it was small, self-contained and nobody was working on that part of the system so no ongoing maintenance activities would be disturbed.

As execution scenarios, we started the application and simply followed the options on the screens: adding a setting, modifying or removing an existing setting, inspecting (non-)existing settings and stopping the application. Figure 6.7 (p.83) shows the resulting concept lattice. The gray boxes represent the executed scenarios, the white boxes represent the number of statements executed over all COBOL source files that have been instrumented. We did not relate features to the scenarios, because this was merely a technical proof of concept.

Now that we are working at the level of code statements, more effort should be invested in the presentation of the results. Clearly, a list of line numbers is insufficient for comfortable analysis. We envision an integrated approach in which a concept lattice as in Figure 6.7 (p.83) is used as a navigation tool through the code. The user would be able to select one or more concepts that represent a feature which will result in a list of affected modules. In each module, the affected statements would be highlighted. For example, Figure 6.8 (p.83) shows the ten (=2+8) lines of CA GEN code which have been executed during the ‘stop application’ scenario.
CHAPTER 6. FEATURE LOCATION

Listing 6.1: Two CA GEN statements with their original line numbers.

Listing 6.2: The COBOL code generated from the CA GEN code as listed in Listing 6.1.

Listing 6.3: The generated COBOL code after instrumentation.
6.3. ATTEMPTED ADJUSTMENTS

Figure 6.7: Concept lattice representing several scenarios for managing archival settings.

Figure 6.8: Highlighting the executed lines when the application is being stopped.
CHAPTER 6. FEATURE LOCATION

```
#!/usr/bin/perl

my $prefix = @ARGV[0];

while(<STDIN>) {
    if ($._ =~ /\*(.*)\*/COMPUTE LAST-STATEMENT-NUM = (\1)\{)
        print $._;
        print "$DISPLAY \$ prefix: \$2 \r\n";
    } else {
        print $._;
    }
}
```

Listing 6.4: Instrumentation PERL script.

This proof of concept, i.e., the source code instrumentation, scenario execution tracing and representation of results (without integration), has been implemented in no more than five man-days. It shows that it is technically possible to implement this statement-level feature location and that it is feasible at least on a small scale. Using the insights from this proof of concept, we listed several expected scalability issues in Listing 6.5 (p.85) which could have been clarified with other quick feasibility studies (pattern 10.1). However, using the currently available information, a go/no-go decision had to be made as to whether we would try to implement this approach for the entire system.

Why it was a no-go  The project was cancelled for two reasons, which were not related to the expected scalability issues. Firstly, the impact on the maintenance activities would be too high. During the experiment, the instrumented source files would be active on the development environment. This means that maintainers can still work on the code in separate packages, however, they cannot test their changes as this requires them to promote their version to the runtime environment, which will remove some of the instrumented binaries. Running the experiments at night was also not an option because (i) maintenance is being done both in Belgium and in India, which are in significantly different time zones, and (ii) compiling the entire system would probably take more than a day anyway.

Secondly, apart from all these technical and organizational hurdles, applying this feature location technique still requires a significant upfront investment of creating and preparing execution scenarios for the experiment. This requires domain knowledge that is only available in a very select group of people. At that time, there were no resources available to provide us with that knowledge.

Potential for the future  To circumvent the technical issue, the most feasible solution to apply dynamic feature location is probably to use a hybrid approach. That is, use high-level monitoring (e.g., DORANA) to have a rough estimate of load modules involved and, subsequently, instrument only those modules for a more fine-grained analysis. This minimizes the impact on the maintenance team as only small parts of the system will be unavailable for testing for shorter periods of time. Nonetheless, the investment in creating the scenarios remains.

84
6.3. ATTEMPTED ADJUSTMENTS

(1) Mass Upload
   issue — How easy is it to upload a batch of 10,000 files to mainframe?
   alternative — Instrument the source code on the mainframe itself.

(2) Single Changeman Package
   issue — A changeman package gets exponentially slower wrt the number of members in the package. It could be impossible in practice to commit all changes in one package
   alternative — One package per subsystem, resulting in more work on dependency management.

(3) Mass Compilation
   issue — How long will it take to recompile the entire system. Rough estimates say a day or longer.
   issue — What will this recompilation cost in terms of CPU cycles, which are typically also charged on mainframe?
   alternative — n/a

(4) Trace Data Size
   issue — Simply displaying each statement every time it is executed could lead to very large traces.
   alternative — Implement a smarter tracing mechanism than COBOL DISPLAY statements. For example, represent each statement as one bit that can be true or false. This will produce execution profiles which are linear wrt to the size of the system.

(5) Scenario Isolation
   issue — We need to be able to indicate start and end of a scenario in the execution trace. Currently we manually keep track of positions in the execution trace.
   alternative — Redirect and clear SYSOUT after each scenario if possible.
   alternative — Implement a program that does nothing but print a divider line to SYSOUT.

Listing 6.5: Expected technical scalability issues
6.3.3 Static Feature Location

After the cancelled dynamic feature location experiment from the previous section, we briefly looked into a pure static approach to locate features. That is, only using the source code without actually having to execute the system. This would circumvent all issues with mainframe environments and impact on current development and maintenance teams.

The idea was quite simply to perform data and control flow analysis. Instead of actually compiling and running the system, we would simply simulate the execution by providing some input data and partially evaluating conditions in the code to see which branches will, for sure, not be executed.

A lot of research has been done in the area of symbolic execution and techniques exist to accomplish this. A recent survey is provided by Pasareanu and Visser [2009]. However, the major bottleneck in implementing this in practice is the level of detail of the facts you need to extract from the source code. Not only do you need information about the functional dependencies between modules, you also need to have access to the variables and conditions within one module. Lexical Fact Extraction (pattern 8.2) will not be sufficient to obtain this level of detail, thus, Commercial Fact Extraction (pattern 8.3) (in this case for CA GEN) will have to be used to perform this kind of analysis.

However, the biggest problem we encountered had to do with the way OSIRIS processes its data. This technique assumes that a function is modeled according to the IPO model\(^8\) (Input-Process-Output), namely it (i) receives input data, (ii) processes this data, and (iii) returns the results. A small variant is the IPO+S model, depicted in Figure 6.9 (p.86), which also includes persistent storage in the processing step. In OSIRIS, the input data is mostly restricted to one identifier which is used to retrieve all the necessary data from persistent storage. Thus, to get any sensible results from such a symbolic execution technique, we will need live access to a database to retrieve data, otherwise we will have nothing to evaluate the conditions in the program.

As such, a purely static approach will not work in this case. Incorporating a custom-built link to the database in the simulation framework to retrieve the real data could be a sufficient solution, but creating this database link could be more expensive than overcoming the problems with implementing dynamic feature location. However, we did not pursue this solution any further, therefore, we cannot make any claims about its feasibility.

\(^8\)http://en.wikipedia.org/wiki/IPO_Model
6.4 Summary

In this chapter we have applied the dynamic feature location technique by Eisenbarth et al. [2003] in the context of mainframe enterprise applications. Most of the difficulties we have encountered are rooted in the assumption that one can lift the system under study from its natural habitat to investigate in isolation. In our context, this assumption is unrealistic at best. As such, the biggest issue with applying dynamic feature location is setting up sufficient infrastructure to capture the necessary run-time information from the system under study with as little impact as possible to its environment. The setup in our replication experiment accomplished this, but triggered a number of other issue regarding isolation of scenarios and execution profiles and accuracy of results. Furthermore, although we confirmed that rapid iterations are the key to successful feature location, our setup endangers the ability to explore freely using iterative and incremental analysis.

Looking for other ways to obtain sufficient run-time information, we experienced that abusing existing tools is not that easy and that both technical implications and organizational impact play an import role. However, Quick Feasibility Studies (pattern 10.1) can help to discard certain possibilities more quickly with more confidence, thereby decreasing the barrier to at least trying to (ab)use existing infrastructure. Furthermore, the patterns in Chapter 9 (p.117) give an overview of possible ways to capture run-time information, including a discussing of the trade-offs between the different forces in play. As such, practitioners can use them to find a suitable mechanism in their context.
CHAPTER 6. FEATURE LOCATION
Chapter 7

Redocumentation

Plans get you into things, but you’ve got to work your way out.
— Will Rogers

Knowledge about a software system can be explicit or implicit. Explicit knowledge takes the form of documentation, which is tangible and reusable by other people. Implicit knowledge, on the other hand, resides in the heads of the experts; it is less tangible and more difficult to reuse. Mechanistic organizations, trying to industrialize the software development process as best as possible, strive to have as much explicit knowledge as possible. However, as stated before, the availability of this knowledge is often unsatisfactory, especially in legacy systems as experts retire, documentation gets outdated or is not there at all [Demeyer et al., 2008].

Redocumentation is the process of recovering (hence, the ‘re-’ prefix) documentation that existed or should have existed [Chikofsky and Cross, 1990]. It can be done manually or automatically. Manual redocumentation requires labour from the experts (original developers, maintainers, architects, users or others). It is tedious, error-prone, not intellectually stimulating and can only be applied if the experts are available. Automatic redocumentation seems much more promising, as it does not require manual labour [van Deursen and Kuipers, 1999]. Nonetheless, some things cannot be automated.

As a reverse engineering technique, redocumentation is very much focussed on the data gathering and knowledge organization activity, but not on the information exploration activity. As such, redocumentation is different from the previous two techniques discussed in this dissertation because it is all about recovering structural facts about the software system and explicitizing them in a more human-readable and accessible documentation format to enable subsequent information exploration. Thus, real comprehension, which is inevitably a human process, is not part of this reverse engineering technique.

Thus, documentation enables comprehension, which is essential to efficiently evolve and maintain
a software system. Since the documentation is typically lacking or out of date, we set out to study the applicability of redocumentation in the context of mainframe enterprise applications. As such, we report about a redocumentation experience from the field which provides insights on what to (and what not to) automate and how to accomplish this. The chapter is structured as follows. In Section 7.1 (p.90) we present more details about the context in which this study was performed. Afterwards, in Section 7.2 (p.92), we sketch the redocumentation process we applied and we finish with an overview of all lessons we learned in Section 7.3 (p.97). The purpose of these lessons is twofold: first, practitioners (e.g., technical designers and maintainers) can use them to improve the usability, quality and maintainability of the documentation they create; second, researchers can use them to improve their (re)documentation tools and methods.

7.1 Project Context

This section establishes a context in which the study was performed. It elaborates on the organization and its typical development methods, illustrating the need for documentation. Furthermore, it explains the documentation standards within the organization, some specifics about the system under study and the actual goals of the redocumentation effort.

7.1.1 Organization

As explained before organizations either have a more flexible organic culture or a more structured mechanistic culture [Burns and Stalker, 1994]. This dissertation focusses on organizations that adopt a mechanistic culture thereby favoring a more structured, industrial approach to the software development lifecycle. This results in strict rules and standard operating procedures in which all employees have clearly defined responsibilities. Communication between these units of labour is typically facilitated by comprehensive documentation [Vinekar and Huntley, 2010].

The maintenance of the system on which we report, will soon be outsourced in order to cut expenses [Tas and Sunder, 2004]. In general, it is more and more common to outsource or off-shore parts of the development process, such as the maintenance [Ahmed, 2006]. Clearly, comprehensive documentation is a prerequisite here, because the subcontractor cannot fall back on the implicit knowledge which is present in the heads of the experts.

7.1.2 Documentation Standard

The organization uses an enterprise-wide documentation standard which is largely based on Component Based UML [Cheesman and Daniels, 2000], tailored to the specific structure of typical banking applications. Furthermore, this standard is supported by MEGA, a general purpose documentation tool which is highly customizable. At its core it has only two notions: objects with certain characteristics and links between objects with certain characteristics. To support
7.1. PROJECT CONTEXT

the rigorous documentation process, the MEGA tool has been customized to support this tailored version of UML.

There are documentation guidelines for all phases of the software development life-cycle. On the domain level, for example, diagrams are defined to document the business processes within that domain, indicate which parts of these processes are automated and which systems are responsible for it. On system-level, functional documentation is required in the form of use cases and conceptual data models. In turn, these have to be linked to technical documentation containing all functions, database tables, screens, batch jobs and the dependencies among them in several UML class diagrams. Furthermore, the purpose of each function is documented as a comment and the logical steps within that function are documented in UML sequence diagrams.

Some parts of this information are easy to retrieve automatically, other parts are not. For example, it is usually possible to retrieve technical data dependencies from the source code, but one cannot automatically retrieve the logical steps of a program from the source code because this requires human interpretation.

7.1.3 System under Study

The system under study is the Custody Services Back-end as introduced in Section 4.3. It is largely implemented in CA GEN, a CASE tool which stores its information not in plain text source code but in a dedicated, proprietary repository.

The system is currently undergoing a technical conversion for two reasons. First, it aims to move away from the CA GEN environment, which will no longer be supported, and replace it with EGL, a platform-independent, business-oriented language that is being introduced throughout the entire organization. Second, it aims to move away from an outdated mainframe environment which is currently only being supported to run this one system created in CA GEN.

When the technical conversion project started, there was no documentation about OSIRIS available in MEGA. However, since the maintenance being done on the current version of OSIRIS (in parallel with the technical conversion) is being outsourced, it is necessary to document the required changes very well. Therefore, the MEGA documentation is created on an as-needed basis, i.e., if a function needs change (1) the function is documented as-is, (2) the change is documented, and (3) the documentation is sent to the subcontractors for implementation. As such, some documentation is already available in the MEGA tool.

Although the documentation standard itself is very well documented and clearly articulates which things should be documented and how, there are two ways in which OSIRIS cannot adhere to this standard. Firstly, the documentation standard has been created under the assumption that mainframe systems will only be used on the back-end, i.e., without direct interaction with the end-user. This is indeed a policy being pushed within the organization, however, our legacy system still uses mainframe screens to interact with end-users. These screens are not provided in the documentation standard. Secondly, a subsystem is defined as a set of data and a set of functions working on that data. Functions can be exposed to other systems but the data will always be encapsulated within the subsystem (much like with component based development). However,
CHAPTER 7. REDOCUMENTATION

OSIRIS is not structured like this. It is one big system but due to technical scalability issues it has been divided into logical subsystems all working on the same data. This is likely to cause issues when documenting the system.

7.1.4 Project Goal

Knowing that it is not possible nor desirable to automate all documentation, the goal of this project is to automatically generate parts of the technical documentation (so nothing on domain or functional level) to provide a basis for a complete redocumentation effort.

More specifically, we decide to (i) reduce laborious, repetitive and boring manual effort, (ii) focus on quick wins, i.e., reduce a significant amount of effort with minimal investments, and (iii) take into account reuse of the redocumentation process for other legacy systems in the future.

7.1.5 Why not buy a tool?

Given this goal, one could argue to buy a commercial tool that can collect the necessary information from the source system and transform it into correct documentation. However, in Section 5.5 (p.58) we have already argued that the biggest cost for applying fact extraction on mainframe is in the various languages and dialects used in these legacy systems. Furthermore, van Deursen and Kuipers attest that they have yet to see a legacy system that does not use compiler-specific extensions or locally developed constructs [van Deursen and Kuipers, 1999]. Moreover, the target of the redocumentation effort is the documentation tool MEGA, which is specifically tailored to the organization’s needs. To the best of our knowledge, there is no commercial tool available matching these specific demands. However, if there is a commercial tool available that, after the necessary customization, can collect the necessary information, it would not be worth the cost as the next system within the organization will be written in yet another language. Consequently, a customized redocumentation process is the most cost-effective, especially because it can be tailored to some organizational standards making reuse within the organization more likely.

7.2 Redocumentation Process

From a high-level perspective, two steps are necessary to redocument a system. Firstly, one needs to extract facts about the system. That can be done starting from a static representation of the system (e.g., the source code), from already available documentation (text documents, spreadsheets, ...) or from the people working with the system. Secondly, these facts need to be combined and transformed into the correct documentation format (e.g., UML diagrams or hyperdocuments).

Although not strictly necessary, we choose to add an extra step in between. Namely, the representation of the extracted facts in a language-independent model, i.e., independent from the source system characteristics and independent from the target documentation format. This way, part of
the approach can easily be reused for redocumenting other systems within the organization using the same documentation format. Furthermore, when the documentation format changes (which is currently happening in the organization), our redocumentation approach can be easily adopted as well.

Figure 7.1 (p.93) depicts the details of these three steps, which we explain in the next three sections.

7.2.1 Fact Extraction

On a technical level, there are two kinds of facts which can be gathered. On the one hand, structural facts can be extracted from the system; for example, program names, functional dependencies, data dependencies, etc. On the other hand, there is also more semantic information; like the purpose of a function or the logical steps within a function. The latter are notoriously harder to extract, mainly because they are not provided (consistently) as they are not necessary to compile and run the system.

We gather both structural and semantic facts from three different sources.

The CA GEN Environment Since CA GEN is a CASE tool it does not store its information as plain-text source code. Rather, it stores its information in the CA GEN Host Encyclopedia, the master data repository containing screen definitions, interaction models, pieces of code,
etc. A public interface to this complex host encyclopedia is provided in the form of about 100 database tables. Although limited detail is provided (no information about individual statements), the most important facts for information systems are available. Structural facts such as action step usage, data usage and data definitions, but also semantic facts such as descriptions of action blocks and data definitions. We extract this information via several relatively straightforward SQL queries. We download the generated SQL reports from the mainframe for later processing.

**Spreadsheets** Some information is not available in the CA GEN environment. For example, OSIRIS is created as one big system, without the notion of subsystems. During its lifetime, it has been split up rather arbitrarily in several CA GEN models because of technical scalability issues. Because of the technical conversion to the standard platform, a more functional separation has been semi-automatically created and documented in a spreadsheet. We use this input to categorize all functional entities in the correct subsystem, which we need to create separate documentation repositories for each subsystem.

**Existing MEGA Repositories** As explained before, some documentation is already available in the MEGA tool due to outsourced maintenance. The structural parts can just be replaced by what we will generate. However, some more semantic parts have required significant manual effort and cannot be automated. For example, documenting the logical steps within one function is an inherently manual process as it requires human interpretation. To make sure we take this documentation into account we extract it from the existing MEGA repositories.

### 7.2.2 Fact Representation

All the facts extracted in the previous step are stored in a plain-text format, be it table-like SQL reports, comma separated values or the MEGA export format. This makes them relatively easy to process using nothing but regular expressions and string manipulations implemented in Perl.

As intermediate format we use RSF, the Rigi Standard Format, conceived as part of the Rigi Reverse Engineering Environment. RSF is a lightweight, text-based exchange format that uses sequences of tuples to encode graphs. As such, a tuple can (1) define a node and its type, (2) define an edge between two nodes, or (3) bind a value to an attribute. Using this format, one can specify anything that can be represented as a typed, directed, attributed graph. Software systems are commonly expressed as such graphs [Kienle and Müller, 2008].

We choose RSF for its simplicity and for the potential reuse of tools and expertise from the FETCH² tool chain [Du Bois et al., 2007].

Figure 7.2 (p.95) depicts the basis of our RSF meta-model. It reflects the common structure of information systems in the organization under study, which boils down to data processing. Functions are basically steps within an automated process which, in our case, are implemented in COBOL programs generated from CA GEN. A data entity is simply a table in a database. Functions can manipulate data using CRUD operations (creating, reading, updating or deleting rows) on a

---

²FETCH stands for Fact Extraction ToolChain, info and tools via [http://lore.ua.ac.be/fetchWiki/](http://lore.ua.ac.be/fetchWiki/)
7.2. REDOCUMENTATION PROCESS

Table and invoke other functions to delegate some of its work. Both functions and data entities belong to a module, which in our case represents a part of a subsystem.

Figure 7.3 (p.95) shows the different kinds of functions that are available. A function can be (1) a batch job used in bulk data processing, (2) an online step represented by a mainframe screen for interactive processing by a user, or (3) a reusable piece of functionality used by all three kinds of functions. A reusable function can either be public or private indicating whether or not it is an exposed functionality of the subsystem. A batch job typically reads and writes files in bulk and generates a report. These batch characteristics are denoted in gray because they are not yet incorporated into our redocumentation process.

7.2.3 Documentation Generation

After collecting all facts of OSIRIS in one RSF file, we process it to generate the desired documentation format. For this, we use CROCOPAT, a graph query engine by Dirk Beyer [Beyer et al., 2005]. It supports the querying of large software models in RSF using PROLOG-like Relation Manipulation Language (RML) scripts. This way we can, for example, easily extract all the different interactions of a specific function, or an entire call-graph starting from an online screen.

Of course, for documentation purposes, this information has to be loaded into a documentation tool. For this, we use the textual MEGA input format. In essence, this is no more than a sequence of .Create and .Connect statements, to add certain objects to the MEGA repository and connect
them in a meaningful way. What kind of objects can be created and how they can be connected depends on the meta-model used, in our case it is a meta-model based on UML. However, how this meta-model translates to specific MEGA statements is not documented at all. As such, we had to reverse engineer the MEGA format itself. Because the import and export formats of MEGA are very similar, we modeled some toy examples in MEGA and exported them to see how it would translate to a combination of objects and connections.

Listing 7.1 (p.96) gives a simple example of the MEGA import format where two modules are created on lines 1–4, which translate to UML packages. Note how we can attach characteristics of objects like the name in line 2, for instance. Furthermore, two functions are created in lines 6–14, which translate into UML classes. The first function is marked as a public reusable function by means of a pre-existing UML stereotype and attached to the first package using the OwnedClass relation. The second function is marked as a batch job and attached to the second package. Lastly, on line 16, the reusable function is defined to be required for the batch job, essentially documenting an invoke relation.

As mentioned before, we generate this output directly from our RSF file using CROCOPAT scripts. However, most of the already existing MEGA documentation from the parallel maintenance projects is directly integrated with the output format, without being stored in RSF. The main reason for this is that the documentation mostly involves rich-text comments which are (1) not supported by RSF and (2) stored in an obscure hexadecimal format proprietary to the MEGA tool. By feeding it directly back into the input format, we circumvent all translation issues. Concretely, we use CROCOPAT scripts to generate a template MEGA input file for the comments and then use some PERL scripts to merge this template with the extracted comments from MEGA.

Unfortunately, some technical limitations to the MEGA import format prevent us from automating the generation of all diagrams defined in the organization’s documentation standard. For example, you can disable specific operations from a UML class in a specific view, essentially making them invisible in that view. This approach is used to indicate whether or not a function performs a specific CRUD operation on a data entity. Because the visibility of operations are cosmetic properties of a diagram, they are not accessible via the import format. Hence, they can only be disabled manually. We did provide annotations with the correct information in each of the
7.3 Lessons Learned

In this section, we enumerate and explain the lessons we learned when applying the redocumentation process of Section 7.2 (p. 92) on the software system as explained in Section 7.1 (p. 90). They are all based on observations we made in this real-life case. The purpose is twofold: first, practitioners can use them to improve the usability, quality and maintainability of the documentation they create; second, researchers can use them to improve their (re)documentation tools and methods. We present the lessons in an arbitrary order.

7.3.1 Only document relevant parts

While collaborating with the banking company, it became clear that they had only very limited resources to spend on the creation and maintenance of documentation. This was quite surprising, given that the banking company has a more mechanistic culture and adopts industrialized approaches in which documentation is essential to the software development process.

Nevertheless, we were hard-pressed for time to automatically generate a basis which can be completed manually (there is a strict deadline on which the manual documentation starts). Therefore, we had to prioritize the kind of views and documentation we would tackle first. Our initial thought was to take the documentation standard of the organization and start with those things that are the easiest to automate and the most mundane, repetitive and error-prone to do manually. However, this assumes that all views are equally important and should be there in the end anyway. In practice this is not the case, especially because the documentation standards are not necessarily constructed by the people who actually create and maintain the documentation. Rather, the standard reflects a nice to have view on documentation, but as long as the necessary resources are not available to actually produce the documentation, those parts with the most practical value should be tackled first.

The lesson we learn from these observations are twofold. Not only the effort of generating documentation should be considered, but also the effort for maintaining the generated documentation has to be taken into account when performing a (re)documentation effort. For those reasons, it seems that one should only document the parts that will be relevant and that will be maintained and kept up to date. A prioritization of what to document should be a part of the (re)documentation process.

7.3.2 Less is more

We observed that when performing automated redocumentation, it is very easy to get carried away and generate documentation just because you have access to the facts. More documentation,
CHAPTER 7. REDOCUMENTATION

However, is not always better. Feedback from the maintenance team learned us that there are two motivations for avoiding an over-production of documentation. First, a huge amount of documentation might lead to an information overload for the users of the documentation, which makes the documentation itself becomes less usable. Second, the trustworthiness of the documentation decreases as huge documentation repositories have a reputation of not being maintained and consequently are suspected of containing outdated information. This suspicion results in a further lack of commitment to keep the documentation up to date: a typical example of a self-fulfilling prophecy.

The lesson we learn from this observation is that we should not only ask whether we can generate the documentation, but also whether we should. The main criteria should be whether or not the generated documentation will be used and whether or not it will be maintained. If the answer to any of those questions is no, the redocumentation effort will have been in vain resulting in documentation that will soon be outdated again. We conclude that one should only create documentation that will be used and maintained. It is best to have little documentation that is easy to understand and that can be fully trusted.

7.3.3 Make documentation usable

The system under study has a rather artificial structure in which one subsystem contains all the data that is used by the other subsystems. As the documentation policy states that the description of the data is only included in the subsystem defining the data and not in the subsystem using the data, all the documentation that explains the data is grouped into one repository. This is rather annoying from a documentation point of view, where maintainers claim they would benefit more from having the documentation available where they use or plan to use the data. Indeed, this hinders the maintenance process and makes the maintainers sometimes guess about the data documentation instead of looking it up.

The lesson we learn here is that the structure of the documentation has to support the way it will be used. The available information has to be easily accessible there where it is required.

7.3.4 Do not duplicate documentation

During redocumentation, we observed that the maintainers were not always satisfied with how the documentation was structured throughout the system. In Subsection 7.3.3 we already elaborated on the situation in which the data documentation is only included in the subsystem where it is defined, and not where it is used. To help the maintainers, we suggested to include the description also in the subsystems using the data. However, this suggestion was refused because this means that every time something changes about the data, the documentation potentially has to be updated in nine different locations. Something even the maintainers admitted would not happen.

We learn that it is not a good approach to duplicate documentation, as this would increase the maintenance costs of the documentation. Moreover, it might bring along inconsistencies within the information specified in the multiple copies. Instead, we suggest to use references to the
7.3. LESSONS LEARNED

![Diagram](image1)

**Figure 7.4:** Point-to-point versus indirect transformation for the system under study

![Diagram](image2)

**Figure 7.5:** Point-to-point versus indirect transformation within the system under study

documentation. References can be inserted throughout the entire software system. They point to the relevant parts of documentation and support developers to obtain the relevant documentation information, wherever it is required.

### 7.3.5 Use a generic meta-model

The most straightforward way to perform a redocumentation effort is to implement a point-to-point transformation from the sources to the documentation. However, we observed that a lot of the transformation work is redundant both from the organizational as from the system perspective. Therefore, we opt to add an indirection in the redocumentation process, representing all the extracted facts in a generic way using RSF.

Consider Figure 7.4 (p.99), in which the transformation is depicted from OSIRIS to the MEGA UML format. The left hand side shows a point-to-point transformation, the right hand side includes an indirection via RSF. Both representations have a dotted arrow for extracting facts and a full arrow for generating the documentation from these facts. At first sight, this indirection increases the total amount of work, because the amount of arrows remains stable but a box is added.

However, when we study the system in detail we see a different picture altogether. Consider Figure 7.5 (p.99) in which we depict the same transformation from OSIRIS to the MEGA repository, but taking into account the different fact extraction sources and the different views to be documented. The Screen Flow Views could just require facts from CA Gen to be transformed. The Job Flow Views also require information from the spreadsheets as jobs can crosscut subsystems. The Basic Flow Views also require semantic information, some of which was already documented in MEGA. The combinations between these different sources will have to be performed ad-hoc when no
intermediate model is used. The example in Figure 7.5 (p.99) shows a point-to-point transformation with nine steps (arrows), whereas using RSF as intermediate model reduces the amount of steps to six. Clearly, the necessary point-to-point transformations will explode when fact extraction sources and, more likely, documentation views increase.

Furthermore, the indirection allows us to reuse all the work on transforming to MEGA, independent from the implementation language of the source system. This is very relevant in our context because MEGA is used throughout the entire organization and CA GEN is only used for the Custody Services Back-end. On top of that, most banking systems have a very typical structure, an example of which we described earlier in Section 7.2.2 (p.94), making it easy to define and reuse a generic meta-model. In Figure 7.6 we show how the indirection allows reuse of the transformations to MEGA. If another banking system (such as portfolio or order management) has to be documented, we can reuse the documentation generation. For three systems this results in four arrows instead of six. Similarly, if another output format is chosen by the organization (e.g., UML implemented in another tool), we can reuse all the fact extraction work.

Concretely, we learn that it is good to use an indirection that splits up the fact extraction and the documentation generation phases because it brings along two concrete advantages. First, the extra effort it takes to implement such indirection pays off fast as the indirect architecture increases the reuse opportunities both within one system and among several systems. Second, it allows for using a generic meta-model (such as RSF) in which the extracted facts can be expressed. As this meta-model is supported by existing analysis tools (like CROCOPAT), one can take advantages of those tools for different purposes: analysis, system comprehension or redocumentation itself.

### 7.3.6 Query engines are not reporting tools

In our redocumentation effort, we make use of the CROCOPAT tool. We experienced that it allows one to write very concise queries using first-order predicate calculus which execute blazingly fast. Transforming these results into the MEGA output format, however, is a tedious task and requires a lot of effort. Furthermore, CROCOPAT lacks any kind of modularization for the RML scripts. For example, it is impossible to call a script from another script. As a result you either get one very big script doing everything, or you get a lot of duplication in your queries. In practice, the latter often results both in duplicated scripting code and duplicated calculation time.

The lessons we learn is that a good query engine is a big plus during redocumentation, but that query engines alone are not enough. To comfortably generate the correct output format a good
7.3. LESSONS LEARNED

reporting tool is necessary to complement a query engine.

7.3.7 Be opportunistic when extracting facts

We choose to keep the fact extraction very lightweight. We extract most of the CA GEN facts using SQL queries, limiting the information to what is available in the public interface of the host encyclopedia. All other necessary facts we extract using lexical techniques [Murphy and Notkin, 1996], for example, when processing already existing documentation. Performing a full syntactic analysis is not cost-effective, especially because this is a one-time operation, and no other systems use the CA GEN environment.

In general, we promote Opportunistic Fact Extraction (pattern 8.1), i.e., looking for tools in the development and build process that already do much of the hard work, and merge all these bits and pieces of information into a coherent model.

7.3.8 Not everything can/should be automated

We started out with the premise that as much as possible should be automated. Of course, some factors prevent full automation. Firstly, some type of information might be needed in the documentation but not available as facts to be extracted. For example, semantic information on the purpose of some functions or how they represent real-world objects is not necessarily available in the code or any other documentation. This kind of information requires human interpretation and is, thus, by default not automatable. Secondly, some technical limitations could prevent automation that, in principle, should not pose a problem. For example, at the end of Section 7.2.3 (p.95) we explain how the MEGA import format does not cover all features of the MEGA tool, necessitating manual corrections.

Besides the fact that some things are impossible to automate, it is not necessarily desirable to automate all things that are possible. For example, the diagrams we import in MEGA lack a layout. They are drawn on the screen using a built-in layout algorithm which is not very human readable. It is possible to define the position of elements on a diagram, which we can use to create our own layout algorithm that would, to some extent, be more human readable. However, the investment of creating such an algorithm is bigger than manually layouting the diagrams on an as-needed basis. Especially because manual intervention will be necessary anyway.

We learn that full automatic redocumentation is not possible. Technical barriers and lacking semantic information necessitate a human in the loop. Furthermore, full automation is not desirable as the cost of automation can exceed the cost of doing it manually: only automate what is cost-effective.

7.3.9 Get regular user feedback

Because we base ourselves on the documentation standard which is used throughout the entire organization, it is not necessary to involve the users too much. In this case, users are those who
CHAPTER 7. REDOCUMENTATION

create, use and maintain the documentation. Nonetheless, we opted to include regular user feedback because we made the following observations: (1) not all parts of the documentation standard are documented equally well, (2) mapping the legacy system constructs to the documentation standard is not always straightforward, especially because the legacy system does not adhere to all organizational standards, (3) not all parts of the standard are equally maintainable in practice, so we had to prioritize based on usefulness, and (4) not all constructs, as defined in MEGA, are possible to import automatically into MEGA, therefore some issues had to be flagged for manual fixing.

We learn that, although documentation standards are available in these kinds of organizations, it is still best to work iteratively and include regular user feedback. This way, one can quickly focus on the most relevant parts, resolve system-specific oddities in consensus and omit irrelevant or harmful parts early in the process.

7.4 Summary

The availability and quality of software documentation is often unsatisfactory, especially for legacy systems. To overcome this problem one can recover documentation about the subject system using automatic and manual redocumentation. We report about a redocumentation experience from the field which provides insights on what to automate (and what not to) and how to accomplish this.

We applied redocumentation on the Custody Services Back-end of a large Belgian bank, which uses an enterprise-wide documentation standard based on UML implemented in MEGA (a general purpose documentation tool which is highly customizable). As the maintenance of this system is in the process of being outsourced, the organization needs to make sure that the documentation is up to date. Currently, there is very little documentation available about the system, and most of it is contained within the CA GEN environment which will become obsolete. Therefore, we were asked to provide documentation within the MEGA tool. More specifically, we aim to (i) reduce tedious manual effort, (ii) focus on quick wins, and (iii) take into account possible reuse of the technique on other systems.

Our redocumentation process consists of two phases. First, we extract facts about the system. For that, we use existing artifacts such as the source code, already available documentation and the people working with the system. Second, we transform the extracted facts into the correct documentation format. We adopt an iterative approach, in which we frequently present our findings to the experts. In turn, the experts provide feedback that we use to improve our redocumentation efforts.

Finally, we report on our experience with applying redocumentation techniques in this industrial setting. Most notably, we learned that (1) not everything can be automated, nor should it be, (2) “less is more” also applies to (re)documentation, and (3) using a generic meta-model is not only advantageous for reuse in later projects, but it also pays off for an isolated redocumentation effort. Both academics and practitioners can take advantage of the best practices represented by these lessons. While the latter can use them to improve the usability, quality and maintainability of the
documentation they generate, the former can use them to improve their (re)documentation tools and methods.
Part III

Patterns
Chapter 8

Fact Extraction

— Winston Churchill

Documentation — if available at all — is unreliable at best. System experts are scarce, overworked and tend to be close to retirement. Therefore, the system itself (and source code in particular) is the only true source of information that will always be available\(^1\). However, plain-text source code contains an abundance of information and can be tedious to interpret manually. Furthermore, most analysis techniques and tools require a more structured format as input. Therefore, facts need to be extracted from the source code to be more easily interpreted by humans and/or tools.

We define a fact as any piece of structural information about a software system that can be extracted or derived from the source code. This can range from the name of a system to the types of the local variables in a function; from the name of the author of a piece of code to a complexity measure of a module; from database dependencies to the control flow within a routine.

The problem we tackle in this chapter is how to obtain these facts from your system. This problem is difficult for several reasons.

**Parsing is difficult** Parsing and compiling a piece of code in a specific language is difficult, as evidenced by an entire research field on compiler technology. However, on the mainframe this is made more difficult due to the several vendor-specific dialects of, for example, COBOL that have organically grown over the years. This makes it very difficult to create standard, reusable tools to parse one language.

**Legacy comes with many languages** Due to the long-living nature of the legacy systems, it is not uncommon for one system to use a mix of languages. Some are used for dedicated

\(^1\)We assume the source code itself has not been lost. Although even then information can be extracted from either executables or a running system.
purposes, such as JCL for job control or SQL for querying databases and will be used in almost all systems. Others are more general purpose of nature and are used to implement the business logic. A wide variety of these 3GL and 4GL languages are used throughout a typical organization with no apparent consistency. Furthermore, these languages interact with each other, resulting in cross-language dependencies which adds another level of complexity to the problem.

Forces

Furthermore, several forces are at play that need to be balanced.

Granularity Do you just need a list of all your functions, or do you also need the dependencies between them? Is a list of identifiers enough or do you also need the function arguments and return type? What about the data? Will it suffice to have an overview of all the persistent data or do you need information on all variables, their scope and how they interact with the functions? Is every little statement important or are you actually just looking for a broad overview of your application portfolio? Facts occur on all levels of abstraction, depending on the detail required your fact extraction methods can vary a lot.

Precision How important is the precision of the extracted facts for you? Will the world end when your automatic fact extraction misses two of your 5000 function because of an exotic language construct. If you just need the facts to monitor trends in quality indicators, then these two skipped functions won’t make that much of a difference, especially when you consistently ignore these functions over time. On the other hand, if you need it for very fine grained impact or dead code analysis, these two imprecisions might even inhibit the execution of the analysis.

Resources How much time, effort and money are you willing to put into it? Is this a one-time deal because the facts will never change? In that case you might even consider gathering facts manually instead of buying a package you will use once. On the other hand, the enormous size of the system might make manual extraction prohibitively expensive and buying a package might be the more economic option. Or maybe it concerns a critical yet volatile part of your business that needs constant monitoring, for which a serious investment is more than warranted.

Overview

Figure 9.1 (p.119) shows an overview of the Fact Extraction patterns and how they can be applied. When in need of certain facts, one should always try to Gather Available Facts (pattern 8.1) first. If this is not feasible or sufficient, one can complement the available facts by Performing Text-Based Fact Extraction (pattern 8.2) using lexical techniques on the source code. Only when this is does not provide sufficient information, one should consider Buying a Commercial Fact Extractor (pattern 8.3). Furthermore, to evaluate the feasibility of Gathering Available Facts (pattern 8.1) and Performing Text-Based Fact Extraction (pattern 8.2), one can use Quick Feasibility Studies (pattern 10.1).
Figure 8.1: Overview of the Fact Extraction Patterns.
CHAPTER 8. FACT EXTRACTION

8.1 Gather Available Facts

also known as ‘Opportunistic Fact Extraction’

Problem
How can you obtain the necessary facts about a software system without having to fully analyze the source code yourself?

This problem is difficult because

- the source code provides a lot of detailed information requiring a lot of work to fully extract,
- one software system can use many implementation languages.

Yet, solving this problem is feasible because

- you don’t need all the details, a limited amount of information is enough to answer your question,
- build and management tools already parse and interpret the source code. Some of that information is available to you.

Solution
Gather already available facts about a software system that have been obtained by other tools in your standard development process.

For example:

- Your change management environment can give you a listing of all your compilation units.
- During the build process, some cross-reference information will most likely be collected by the change management environment. Although access to this information is usually through an application with limited functionality, a customized SQL query directly on the correct database tables will get you the bare data.
- Some parsers have an option to dump an internal representation of the code (a parse tree). In that case, the parser has done all the hard work, you just need to interpret the dump. For example, most COBOL compilers accept the XREF compiler option to dump a cross-reference report.

Hint  Make sure you establish a good relationship with the supporting departments in your organization. For some of these options you will have to cooperate with technical operators to get the necessary clearance. They might also be able to point you to other options you don’t know about.

Trade-offs

+ Since these tools are used to build your system, you know the information they provide is correct, i.e., there will be no imprecisions.
8.1. GATHER AVAILABLE FACTS

+ These tools are available for all languages you use in the build process (although options may vary).
+ Depending on what is available, you might gain access to many facts with little or no investment.
  – Not all tools provide this internal information. Even if they do, it is usually just partial information. You are limited to what they expose.
  – You will still need to glue together all the information you gather from different tools.
  – Information about the tools is usually very limited, especially when you want to use them in an exotic way.

Known Uses

In Section 5.5.1 (p.59) we explain how we gathered available facts from the mainframe change management environment for the Portfolio Management System. This change management environment keeps track of language-independent cross-reference information which was important because the Portfolio Management System was implemented with a mix of APS (1.2M lines) and VAG (1.8M lines). This saved us the cost of extracting facts for each language separately.

In Chapter 7 (p.89) we gathered facts from various sources to redocument OSIRIS. Most notably, we created SQL queries to extract facts directly from the exposed representation of the CASE tool CA GEN in which OSIRIS is mainly implemented. Although it did not contain all the information we needed, it set us well on our way.

FETCH² is an open source fact extraction tool chain that is built on the philosophy of reusing and abusing existing tools and glueing them together to obtain the necessary information and analyze it appropriately. For fact extraction, it mainly relies on SOURCENAVIGATOR³, a multi-language IDE project initiated by Red Hat. The main advantage of the IDE is its robustness against various language constructs and dialects. Furthermore, it also incorporates various metrics calculation tools to complement the information from SOURCENAVIGATOR. As such, FETCH has been successfully used in industry to analyze various C and C++ systems [Du Bois et al., 2007].

Related Patterns

Consider conducting a Quick Feasibility Study (pattern 10.1) to see whether a certain facility can get you the required facts. If not all facts are available you can Perform Text-Based Fact Extraction (pattern 8.2) or Buy a Commercial Fact Extractor (pattern 8.3) to complete your set of necessary facts.

²FETCH stands for Fact Extraction ToolChain, info and tools via http://lore.ua.ac.be/fetchWiki/
³http://sourcenav.sourceforge.net/
CHAPTER 8. FACT EXTRACTION

8.2 Perform Text-Based Fact Extraction

also known as ‘Lexical Fact Extraction’

Problem
How can you obtain the necessary facts from the source code of a software system without having to fully analyze the source code yourself?

This problem is difficult because

- the source code provides a lot of detailed information requiring a lot of work to fully extract,
- one software system can use many implementation languages.

Yet, solving this problem is feasible because

- you don’t need all the details, a limited amount of information is enough to answer your question,
- most programming languages use some constructs and keywords that are easily recognizable.

Solution
Use simple text processing tools to extract facts from the free-form source code.

The most simple example of this are text-based search tools such as GREP\(^4\) for UNIX like systems or the SCAN command for mainframe. To use this for fact extraction, a certain degree of automation is desirable. Lightweight scripting languages with support for regular expressions (or anything else that matches textual patterns) are perfect for this.

Support for regular expressions is commonplace in UNIX based system, but not on mainframe systems. Nonetheless, we know of some powerful string manipulation commands in REXX that could be sufficient.

Hint To process the source code with UNIX based tools, you will have to download the sources from mainframe. Most emulators provide a way to download files separately. However, this is infeasible to download the sources of an entire software system. FTP access to download files in bulk is usually possible. Ask the supporting departments how to set it up.

Trade-offs

+ More robust against syntactic variations and exotic language constructs. Therefore, more likely to be reusable in other systems as well.
+ Requires significantly less resources and commitment than parsing an entire language.
- There is a limit to the detail of the facts extracted like this. You will, for example, not always have enough information to perform reliable data or control flow analysis.
- This will most likely result in an incomplete model with many imprecisions.

\(^4\)http://www.gnu.org/software/grep/
8.2. PERFORM TEXT-BASED FACT EXTRACTION

**Known Uses**

For the Document Generation and Management System in Section 5.2.1 (p.37), we have extracted functional dependencies between COBOL programs essentially by searching for the string ‘CALL programName’ in all the source files. Afterwards we related the names of the COBOL files to the program names in the CALL statements. While this was not perfect (some filtering had to be done) it was blazingly fast and served our purpose.

[van Deursen and Kuipers, 1998] have used similar lexical analysis techniques to perform rapid system understanding for two COBOL systems from the financial services industry. The main benefit they articulate is the flexibility of lexical analysis. They have also experienced this flexibility when extracting functional dependencies. One system used an indirection via an ASSEMBLER program to invoke other programs. Therefore, the CALL statements had to be interpreted differently. The other system used CICS constructs which essentially provides two other ways to invoke programs. Their fact extraction method based on lexical analysis was easily adapted to incorporate these exotic constructs. They attest that this would not have been so easy to do with the two commercial solutions they were investigating simultaneously.

[Murphy and Notkin, 1996] have created a lexical source model extractor which allows for easy, hierarchical pattern matching based on regular expressions. They have used it to extract implicit invokes in the FIELD programming environment. These implicit invokes are used to elegantly combine a combination of tools within the environment. But they were not extracted by any tool on the market. They the advantages over a parse-tree based tool as more robust, more flexible, easier to implement and faster to execute. On the downside, these lexical techniques are, of course, less precise and less complete.

**Related Patterns**

This pattern works particularly well in combination with Gathering Available Facts (pattern 8.1) to complement what was not available. Moreover, Gathering Available Facts (pattern 8.1) will most likely also require some text-based post processing. That is, you can also Perform Text-Based Fact Extraction (pattern 8.2) on other listings than source code. Furthermore, consider conducting a Quick Feasibility Study (pattern 10.1) to see whether it is feasible to Perform Text-Based Fact Extraction (pattern 8.2) for your needs.

---

5 http://www.cs.brown.edu/~spr/research/envfield.html
CHAPTER 8. FACT EXTRACTION

8.3 Buy a Commercial Fact Extractor

also known as ‘Commercial Fact Extraction’

Problem

How can you obtain very detailed facts for fine-grained analysis?

This problem is difficult because

- detailed facts require detailed parsing for which all language and compiler peculiarities will have to be taken into account,
- this kind of expertise is most likely too expensive to have in-house.

Yet, solving this problem is feasible because

- someone else has done it before you.

Solution

Buy a fact extractor for your particular language (dialect). Pay a consultant to tweak, deploy and integrate their tools into your environment. Pay the consultants some more to educate you and your colleagues about the tool.

Trade-offs

+ A commercial fact extractor usually comes with an analysis framework, putting you well on your way for solving current and future problems.
+ Although expensive, it is most likely cheaper than building and maintaining a fact extractor in-house.
- A commercial fact extractor will not work out-of-the-box. Not only will integration be necessary, some exotic constructs you use will not be supported. Make sure they are willing to adjust their tools.
- Besides money, it will also cost time and resources from your part to help with the integration and education.

Known Uses

We did not find any published case material supporting the use of commercial fact extractors. However, we know of several companies offering such services and tools. Since they have been around for many years, some organizations use their services and tools enough for them to stay in business.
Related Patterns
Consider Gathering Available Facts (pattern 8.1) and Performing Text-Based Fact Extraction (pattern 8.2) before Buying a Commercial Fact Extractor (pattern 8.3).
Chapter 9

Trace Capturing

In God we trust; all others must bring data.
— W. Edwards Deming

Where Chapter 8 (p.107) shows how fact extraction can provide the basis for studying the structure of a software system, this chapter explores different ways to obtain run-time information to study the behavior of a software system.

Capturing behavior is inherently more difficult than extracting structure from a system. First of all, to study behavior you need a running system. This implies that the system is complete and correct (you need to be able to build it), and that all runtime dependencies are satisfied (you need to be able to execute it). Secondly, a typical system does not just process things by itself, it responds to external stimuli. As such, scenarios need to be available to provide predefined stimuli. Possibly, a human is necessary to execute these scenarios as well. Lastly, it is not enough to make a system do something, at the same time, you need to be able to monitor what the system is actually doing.

The latter is what this chapter focusses on: facilities to help capture run-time behavioral information when providing external stimuli to a running system.

Forces

Several forces need to be balanced when choosing a particular technique to capture run-time information.

**Structural Granularity** Which level of granularity is necessary to perform your analysis. Do you just need a list of subsystems used within a specific scenario? Are you interested in the full run-time interaction of all different functions. Do you need an overview of all executed statements,
for example, to study the intricacies of one function in detail. In short, what is the granularity of the code units that need to be monitored.

**Run-time Detail** Apart from the granularity of the code units, what amount of run-time detail is necessary for each unit? One could, for example, obtain a plain list of executed units (an execution profile) or a complete call tree (an execution trace) in which the order of execution is preserved. In addition, both of them can be augmented with, e.g., execution time information. Depending on the analysis needs, more or less detail is required.

**Scale** What is the scale of the analysis you want to perform? Are you investigating the usage of an entire system or do you just need to investigate one function? To keep the approach scalable, one will have to balance the structural granularity and the run-time detail.

**Technical Impact** All of the above forces will have to take into account the technical impact on the system. The most obvious technical impact is *performance*. What is the overhead of monitoring a system in terms of memory footprint and cpu-cycles? A similar one is *storage*. How large will the trace data get? Can we harbor it physically? Depending on the environment, how much overhead can we afford and will it destabilize or overload a (perhaps business-critical) system?

**Organizational Impact** On top of the technical issues, the organization can be impacted as well. By organizational impact, we mean things like the impact on change management, the release cycle and day to day development and maintenance. For example, if profiling requires recompilation, their might be a huge impact on change management: keeping two version of an entire system and preventing the wrong version from going into production. On the other hand, structural changes to build and deploy procedures can greatly ease the capturing of run-time information, but these changes are not easily accomplished in a mechanistic organization. Furthermore, hijacking an environment to capture run-time information can put an entire development or maintenance team out of work as they cannot use their development environment without interfering the trace capturing.

**Mainframe Environment** There are typically three to four different environments which run different versions of the system in different stages of one release-cycle. Two important factors vary from stage to stage. Earlier stages usually have more and better tool-support for debugging and more flexibility towards tailoring the environment. Later stages are a more realistic reflection of the actual production system, better integrated with other systems using more realistic data, but also more sensible to both technical and organizational impact.

**Overview**

Figure 9.1 (p.119) shows an overview of the Trace Capturing patterns and how they can be applied. The main concern when trying to capture run-time information is the scalability of the approach, influencing all forces above. Therefore, we split up the patterns in two categories, those that can be applied on a large scale for locating areas of interest, and those that can be used on a small scale for subsequent detailed investigation of the areas of interest.
For locating areas of interest using run-time information, one should first try to (Ab)use High-level Monitoring Tools (pattern 9.1). If it is not feasible, one can try to perform coarse-grained Source Code Instrumentation (pattern 9.3). When having identified the areas of interest, one can investigate them in more detail by capturing more detailed run-time information. To do this, one should first try to (Ab)use a Debugger (pattern 9.2) or other profiling tools in your development environment. If it is not feasible, one can try to perform fine-grained Source Code Instrumentation (pattern 9.3). Investigating the feasibility of abusing existing infrastructure can be supported by Quick Feasibility Studies (pattern 10.1).
CHAPTER 9. TRACE CAPTURING

9.1 (Ab)use a High-Level Monitoring Tool

Problem
How to obtain sufficient run-time information to locate areas of interest in a large system?

This problem is difficult because

- adjusting the system under study to provide run-time information is not always possible or allowed.

Yet, solving this problem is feasible because

- you don’t need very fine-grained and detailed information to find an area of interest,
- mainframes typically have some high-level monitoring tools in place in production and pre-production.

Solution
Abuse the output of a high-level monitoring tool, such as DORANA\(^1\) or OMEGAMON\(^2\), to obtain a very coarse-grained execution profile.

Hint
Make sure you establish a good relationship with the supporting departments in your organization. For all these options you will have to cooperate with technical operators to get the necessary clearance. They might also be able to point you to other options you don’t know about.

Trade-offs

+ Organizational and technical impact are minimal as these tools typically run 24/7 on (pre)production systems for ensuring accountability, generating statistics or enabling post-mortem problem resolution.
+ Since impact is low, you can use the pre-production environment and, thus, have access to realistic data and a stable, reliable version of the system under study.
+ This approach scales very well to entire systems as these tools typically run enterprise-wide.
  - Structural granularity is usually very coarse-grained, it is not suited for detailed analysis.
  - You have little to no customization options. You will have to take the structural granularity and run-time detail ‘as is’.

Known Uses
In Section 6.1.4 (p.69) we explain how we abuse DORANA— a tool that provides system-wide logging of load modules — to obtain the necessary run-time information to locate features in the Portfolio Management System. Due to its small technical and organizational impact, this approach allowed us to perform our experiments on the pre-production environment with realistic data. On

\(^2\)http://en.wikipedia.org/wiki/IBM_OMEGAMON
9.1. (AB)USE A HIGH-LEVEL MONITORING TOOL

the other hand, we ran into trouble pin-pointing the exact location of some features due to the coarse granularity.

Arias et al. [2008] gather information from the processes and threads controlled by the operating system (OS) to uncover execution dependencies of the Philips MRI software at an architectural level. Concretely they monitor (i) libraries representing subsystems being loaded by the OS, (ii) data accessed in the form of file accesses or database calls, (iii) CPU and memory usage by the OS. The main advantage of the approach is its non-intrusive nature, they collect up-to-date run-time information without touching the source code or creating overhead.

Related Patterns

You can (Ab)use a High-Level Monitoring Tool (pattern 9.1) to uncover areas of interest within the system under study. Capturing more fine-grained run-time information can be accomplished by (Ab)using a Debugger (pattern 9.2) or Instrumenting Source Code (pattern 9.3) which can monitor the uncovered areas of interest in more detail while discarding other areas. Furthermore, consider conducting a Quick Feasibility Study (pattern 10.1) to see whether it is feasible to (Ab)use a High-Level Monitoring Tool (pattern 9.1) for your needs.
CHAPTER 9. TRACE CAPTURING

9.2 (Ab)use a Debugger

Problem
How to obtain fine-grained and detailed run-time information to investigate area of interest?

This problem is difficult because

• capturing fine-grained and detailed run-time information has a major impact on the system,
• you need hooks into the existing system to capture the run-time information.

Yet, solving this problem is feasible because

• you already know where to look, so you don’t have to monitor the entire system,
• you are actively maintaining the system, hence, you have debug and profiling tools available.

Solution
Abuse a debugger or other profiling tools in your development environment to generate trace output which you can process afterwards to fit your analysis needs.

Hint You might need to alter some parameters of your debugger that you normally don’t use. Ask a supporting department for a course or a manual on the debugger you use. They might also be able to point you to other options you don’t know about.

Trade-offs
+ You are familiar with the infrastructure as you use it often during development and maintenance activities.
+ Structural granularity and run-time detail is usually very fine-grained, including monitoring the value of variables.
  − To provide this fine-grained and detailed information, it usually eats up a lot of resources, especially the memory foot-print can cause problems.
  − It does not scale to full system tracing. As such, you need to make sure you have located the areas of interest in advance.
  − Debuggers are only available on the development environment. If reliable data is necessary to obtain useful run-time information, it will have to be produced from scratch or copied over from the (pre-)production environment.

Known Uses
Before abusing DORANA to obtain the necessary run-time information to locate features in the Portfolio Management System in Section 6.1.4 (p.69), we tried using XPEDITER, the standard debugging tool for mainframe in the organization. We found it very easy to tweak the profiling and obtain the information we needed. We soon realized, however, that we could not monitor more than a dozen modules simultaneously before filling up the memory available to our mainframe profile. After increasing our dedicated memory to its maximum, we could still only monitor about
30 modules simultaneously. Furthermore, using XPEDITER we can only trigger execution in an interactive manner, thus, monitoring batch jobs is impossible. Clearly, this approach was infeasible to be used on an entire system, nonetheless, the potential for focussed fine-grained tracing is enormous.

**Related Patterns**

First (Ab)use a High-Level Monitoring Tool (pattern 9.1) to better delineate the area of interest, because you cannot debug an entire system at once. Conduct a Quick Feasibility Study (pattern 10.1) to see whether it is feasible to (Ab)using a Debugger (pattern 9.2) for your needs. If not, consider Instrumenting Source Code (pattern 9.3). When (Ab)using a Debugger (pattern 9.2), consider Performing Text-Based Fact Extraction (pattern 8.2) to transform the debug output to a format usable for your analysis needs.
CHAPTER 9. TRACE CAPTURING

9.3 Instrument Source Code

Problem
How to capture run-time information from a software system without tool support.

This problem is difficult because

- you need hooks into the existing system to capture the run-time information,
- you cannot rely on tools to do the dirty work.

Yet, solving this problem is feasible because

- you are actively maintaining the system, hence, you have the ability to change and rebuild the system.

Solution
Instrument the source code by adding monitoring statements to all the places of interest.

Instrumenting source code for very fine-grained run-time information is typically not that hard to do: just provide a DISPLAY statement before each original statement. Similarly, instrumentation for very coarse-grained information is usually doable as well, provide a logging statement when entering or when invoking a module, which typically have clear boundaries. However, everything in between can be more difficult, especially when facing less structured programming languages. In particular, GOTO statements and other structure-breaking constructs can cause a lot of issues.

Furthermore, one should take into account the intent of capturing run-time information. When looking for areas of interest on a large scale, use coarse-grained instrumentation to limit technical impact. Nonetheless, organizational impact remains a bottleneck as you have to modify and recompile all source code, as such, it affects the change management environment and concurrent development and maintenance activities. When investigating certain areas in more detail, use fine-grained instrumentation. The small scale of the investigation will limit technical and organizational impact, however, one will most likely be limited to the development environment for such experiment.

Hint Adding these extra statements is labour-intensive when instrumenting more than a source code files. Therefore, this instrumentation is best automated with some kind of script that changes the source code. Consider using lexical techniques, like we propose when Performing Text-Based Fact Extraction (pattern 8.2), for quick and low-cost automated instrumentation.

Trade-offs

+ You do not need to rely on exotic tools or infrastructure, which never perfectly fit your needs.
+ Structural granularity is not an issue: you have full control over what is monitored.
+ Relatively easy to implement on a small scale.
9.3. INSTRUMENT SOURCE CODE

- Instrumenting source code requires recompilation of that code. Depending on the scale, this can have a significant technical and organizational impact.
- Manual instrumentation is tedious, but automating instrumentation can be difficult depending on the programming language used and the granularity and run-time detail needed.
- Brute force approaches can have a huge technical impact on system resources when actually monitoring the system.
- Reliable data might have to be migrated to the development environment.

**Known Uses**

In Section 6.3.2 (p.80) we performed source code instrumentation on a small part of OSIRIS. The implementation was very straightforward and allowed us to obtain run-time information with statement-level granularity. However, we were not able to use this approach on the entire system due to excessive organizational impact. In particular, the fact that the entire environment would be unavailable for normal maintenance activities was unacceptable.

Eisenbarth et al. [2003] used source code instrumentation on an industry-scale C system (not on mainframe). They were able to recompile the system in a separate environment. As such, they had no organizational issues. Using source code instrumentation, they were able to tailor the solution to their monitoring needs with little effort.

**Related Patterns**

Consider (Ab)using a High-Level Monitoring tool (pattern 9.1) to better delineate the area of interest, as such, you only have to instrument a small part of the source code. Consider (Ab)using a Debugger (pattern 9.2) for obtaining fine-grained run-time information when you are not able to change the source code.
Chapter 10

Process Patterns
10.1 Quick Feasibility Study

Context
Several approaches are possible to tackle a certain reverse engineering problem. Some approaches are intuitively more feasible than others, other could be more beneficial.

Problem
An available tool or method might be useful to solve a problem at hand, but you lack experience and the expertise to make the decision.

This problem is difficult because

- you lack sufficient experience to make a judgement call,
- documentation of the potential tool or method is usually not conclusive regarding custom problems.

Yet, solving this problem is feasible because

- someone in the organization will have at least ‘some’ experience with the tool or method, otherwise it would not be available,
- well-designed experiments can quickly invalidate an option.

Solution
Setup a simple experiment to invalidate the use of a certain tool or method in your context. If ‘successful’ you can quickly discard the option and look for alternatives. If not, you either will have gained experience to pose a new invalidation question or you will have gained enough confidence to take the risk and use it.

Hint Keep the experiment as small as possible and set up new experiments in an iterative and incremental fashion.

Trade-offs
+ You can discard options with more confidence.
+ When lucky, you can get a quick win.
  - Setting up such experiments takes time, you might be better off spending your resources on implementing something that will work for sure.

Known Uses
In Section 6.3.1 (p.79) we have conducted such a small feasibility study to investigate the usability of STROBE, a sample-based monitoring tool, to capture a full system trace on module level. We increased the sampling rate to its maximum and compared the results (monitored statements aggregated to their containing function) with all transactions (main functions accessible via
transaction manager) that had been executed. As a result, we observed that some main functions had not been monitored by STROBE, thus, it would surely not be useful to monitor all functions. This feasibility study has been conducted in one man-day, after which we could safely discard this option from all future discussions and start looking for viable alternatives.

In Section 6.3.2 (p.80) we tested the feasibility of statement-level source code instrumentation for capturing a full execution trace. We have built a prototype to instrument a handful of modules, recompile them on mainframe, execute them and process the results offline. It took us five man-days to implement this prototype and it showed us that it was technically feasible to do source code instrumentation on a small scale. Of course, this does not prove this approach will work on a realistic scale. However, we did pass the first few hurdles and uncovered new unknowns, which we listed in Listing 6.5 (p.85), that can be tackled with subsequent feasibility studies.

**Related Patterns**

This pattern is very similar to the Build Prototypes (pattern A.4) pattern which is extensively used in Agile Software Development [Coplien and Harrison, 2005, page 49].
Part IV

Conclusions
Chapter 11

Conclusions and Recommendations

Learn to pause . . .
or nothing worthwhile will catch up to you.
— Doug King

11.1 Research Questions Revisited

Reverse Engineering

What is the goal of reverse engineering?

What are the characteristics of state-of-the-art reverse engineering techniques?

The ultimate goal of reverse engineering is to support a human to acquire knowledge about an existing software system. We have identified three characteristics that will enable a reverse engineering technique to effectively help reach this goal: (i) it can be used as a starting point for knowledge acquisition, even without prior knowledge available; (ii) it must support rapid incremental and iterative analysis to verify or reject hypotheses, and (iii) it must reveal unknowns that facilitate the formulation of new hypotheses. The latter two characteristics focus on supporting the human learning process which is iterative and incremental by nature.
CHAPTER 11. CONCLUSIONS AND RECOMMENDATIONS

The Financial Services Industry

What are the characteristics of organizations in the financial services industry?

What are the characteristics of mainframes used in the financial services industry?

Organizations from the financial services industry typically have a mechanistic culture favoring plan-based development processes. As such they promote labor specialization resulting in clearly delineated tasks and impose rigorous processes supported by standardized tools to maintain centralized control.

Mainframes nicely reflect the urge to maintain centralized control. They have been originally invented to fulfill the automation needs of these mechanistic organizations. As such, the organizational structures and rigorous processes are clearly visible in the mainframe, allowing for little flexibility. Nonetheless, there is no system better suited to perform commercial data processing, the prime concern for organizations in the financial services industry.

Who can benefit from reverse engineering in the financial services industry?

From our observations — but without any empirical evidence — we have identified two areas in which we believe these organizations can benefit from reverse engineering techniques: during prestudies for evolutionary development and during routine maintenance. As such, the following practitioners can benefit the most from reverse engineering: (i) prestudy leaders when studying the feasibility or estimating the cost of a proposed project, (ii) technical designers when confronted with technical questions from a prestudy leader or when deciding whether a change request is to be considered routine maintenance, (iii) maintainers when asked to implement a change to a part of the system they do not know very well.

Resulting Difficulties

Which factors — both technical and organizational — make it difficult to apply reverse engineering in the financial services industry?

Many Languages and Variants  All forms of reverse engineering have a data gathering activity. Crucial to many techniques is the extraction of structural facts about the source code. However, in the context of mainframes a lot of different languages and language variants are used. Obtaining standardized tooling to do this fact extraction is not possible, extracting facts for all these languages is not feasible, buying commercial fact extractors for all these languages is too costly. Usually, the latter is even impossible because not all languages and variants are commercially supported. Furthermore, none of these commercial fact extractors will work out-of-the-box, customization is always required.

Mainframes are not Visual  We have identified software views as a promising technique for a human to explore a software system in a more tangible way. However, this is not possible on mainframe systems as they do not support a visual environment. Disconnecting the visualizations from the development environment will hold back the full potential of the technique, especially
for maintainers who need a strong link with the code. However, more and more the development environment itself is disconnected from the mainframe, only leaving compilation and execution to the mainframe. This opens up new possibilities to integrate software views into the development environment. Apart from routine maintenance, software views can also be very useful in the setting of a prestudy. Here, the disconnection from mainframe is less of an issue as prestudy leaders are less concerned with the actual code.

**Isolation is a Pain**    When applying dynamic analyses, which we did when performing feature location, the main concern is isolation. The assumption that you can lift a system from its natural habitat in such a rigorously structured environment is unrealistic at best. Applying dynamic analyses on a live environment poses new isolation issues. How can one distinguish between own usage and unrelated system usage? How can one distinguish between different execution scenarios? All these issues can be solved by introducing an extra mainframe environment with the necessary infrastructure, however, the excessively high operational costs of mainframes make this prohibitively expensive.

**Lacking Tool Support**    Although tool support for the development process and the supporting processes is typically pretty advanced and commonly used on mainframe, tools specifically supporting reverse engineering are seldom available. For example, obtaining reliable execution traces on a live environment, taking into account all the isolation issues, should be possible with the necessary supporting infrastructure.

**Mechanistic Culture**    The mechanistic culture and plan-based approach is at odds with the iterative and exploratory nature of reverse engineering. For example, applying one iteration of the feature location technique requires a lot of pre-planning due to the technical operators from supporting departments involved. As such, rapid turn-around cycles were not possible, thereby severely hampering the usefulness of these techniques for efficient knowledge acquisition.

**Rules Not Rooted in Practice**    All these rigorous rules and procedures are in place to optimize and maintain control over an industrialized process. However, not all these rules are rooted in practice. For example, when conducting a redocumentation effort, it became clear that the imposed documentation formats were not always considered useful or beneficial for the people working with the documentation. As such, there is a lack of commitment to create and — more importantly — maintain the documentation. While reverse engineering techniques can help create mandatory documentation, it is of little use to create it to adhere to rules that have lost touch with reality. Even more, it is counterproductive to do so because lack of maintenance to the documentation will make it unreliable and, thus, unusable for acquiring knowledge.
11.2 Recommendations

What Can Organizations Do?

What can organizations do to better support reverse engineering?

The cost of applying reverse engineering techniques on an ad-hoc project-per-project basis is currently excessively high for organizations in the financial services industry, mainly due to their mechanistic culture and lack of tool support. The only way to really tackle this problem is to streamline the mainframe infrastructure and to incorporate reverse engineering in the organizational processes. In order to do so, we recommend two gradual transformations.

Industrialize Reverse Engineering Although we have portrayed reverse engineering as an organic process, stressing the need for exploration, iteration and human interaction, some activities within the reverse engineering process can be industrialized. For example, in many cases the data gathering activity is a prime candidate for such mechanistic automation. Just like the very mechanistic build and deploy process in forward engineering, one can have a similar mechanistic fact extraction process building a knowledge base that can be used afterwards. A next step could be the automation of documentation generation or — more useful — automatic verification of documentation conformance. The general idea is to automate and incorporate as much as feasible into the current industrial development practices.

Agilize Organizational Processes It is imperative that organizations recognize knowledge acquisition as a human process best supported by tools. It is a natural learning process which is iterative and incremental by nature. Less rigorous processes and procedures can help facilitate faster turn-around cycles. Furthermore, one should minimize documentation that needs to be kept up to date manually and maximize tool support for system comprehension. Being able to use these tools to their full potential also implies providing a secure work environment that enables and promotes exploration and interaction.

Reverse engineering needs to be taken into account from the start, instead of being considered as an after-thought. It should become a natural part of the software development life-cycle, getting equal attention and tool support as forward engineering.

What Can The Research Community Do?

What can the research community do to better support reverse engineering in the financial services industry?

Cross-Language Dependencies We have identified the extraction and analysis of cross-language dependencies as an interesting research domain. Results from that domain can be of great importance within the financial services industry. As such, we would like to encourage the current researchers to continue their efforts and urge them not to forget the typical mainframe languages.
Empirical and Industrial Research  The focus of the reverse engineering community is still very much on creating new techniques in vitro that look fancy and work under laboratory conditions. However, for some reason these techniques are not adopted by industry. Therefore, we urge the entire reverse engineering community to do more empirical research to show the benefit of applying reverse engineering. This way, the industry might be more inclined to apply it as well. Furthermore, we urge the community to do more industrial research to see how these techniques hold up under realistic circumstances, also taking into account organizational intricacies. This way, we can investigate which difficulties reduce or eliminate the benefit of applying reverse engineering in industry and start addressing them.

Keep it Simple!  In his keynote on the 2010 Symposium On Applied Computing\(^1\), Willy Zwaenepoel attributed the lack of adoption of research ideas to the following contradiction:

In order to have your idea accepted at a major conference, it needs to be complex. In order for your idea to have any impact in industry, it must be simple and comprehensible to the above-average programmer.

He has observed that simple papers tend to get rejected and complex papers tend to get accepted, without any regard for the applicability of the idea in realistic circumstances. Although we do not intend to start a controversy on the academic reviewing and publishing system, we do agree that research should be more rooted in practice. There is no use in adding complexity to a technique to solve problems that do not really exist. Thus, we urge the research community to focus more on solving problems that exist in real life with techniques that are as simple as possible.

What Can Practitioners Do?

What can practitioners do to work around these difficulties while waiting for their organization and the research community to catch up?

Be Opportunistic  While waiting for decent tooling and adapted procedures, practitioners should be opportunistic when trying to acquire knowledge. Many scattered bits and pieces of information about a software system are already available in mainframe environments, thus practitioners can Gathering Available Facts (pattern 8.1) to reason about. Furthermore, practitioners should build a good relationship with the supporting departments and get an overview of the facilities that are available. Do not be afraid to start Quick Feasibility Studies (pattern 10.1) to check the potential of available tools for different purposes.

Be Lazy  A nice motto in life is to get the most done with the least possible effort. As such, practitioners should avoid repetitive tasks: when you have done something twice, consider automating it. Performing Text-Based Fact Extraction (pattern 8.2) and combining available facts are nice examples of this. Again, a good relationship with the supporting departments is desirable.

Be Courageous  Maximize the use of your flexible development environment, explore the possibilities. Spend some time trying to patch together a bit of ‘tool’ support for acquiring

\(^1\)http://www.acm.org/conferences/sac/sac2010/
knowledge about the software system you are working on. Share your knowledge with others. Show your increase in productivity to management. Constructively apply bottom-up pressure to get better tool support and organizational support.

Be a Student Most importantly, one should never stop learning. Learn from fellow practitioners by reading books, studying patterns and sharing experiences. Contribute (as this the best way to learn) by teaching newcomers and documenting useful patterns as you discover them. Never stop being a student.

11.3 Concluding Thoughts

In the book, ‘Artful Making: what managers need to know about how artists work’, Austin and Devin explore the differences between industrial making and artful making. Where industrial making focusses on replication of products and optimizing the process to do that, artful making is all about reconceiving, innovating and creating a workspace that allows it. By drawing a parallel with how a theatre company creates a play, they argue that software development — being subject to so much unpredictable change — should be considered a form of artful making and should be managed accordingly [Austin and Devin, 2003].

The financial services industry, operating in a mechanistic culture, takes the industrial approach of software engineering, which is not necessarily bad because IT is not the core business of the financial services industry. They focus on replicating and optimizing the process to build software systems that support their core business. They have no need to be innovative or revolutionary in that respect. But of course, the existing systems should not hamper the ability of business to innovate. On the other hand, reverse engineering — and system comprehension in particular — is an inherently creative process. It is the artful (re)making of an existing system in the human mind, it is a natural learning process that cannot be replicated as it will be different each time.

We believe that this mismatch between the artful (re)making of reverse engineering and the industrial making of software development on the mainframe in the financial services industry is the main cause for the issues identified in this dissertation.

Nonetheless, we firmly believe that the financial services industry can greatly benefit from incorporating reverse engineering in their processes. Therefore, we hope to have given the organizations some useful pointers to accomplish this, to have persuaded more researchers to do in vivo research, and to have encouraged practitioners to start or continue to use reverse engineering.
Part V

Appendices
Appendix A

Thumbnail Patterns

A.1 Chat with the Maintainers

**Intent**  Learn about the historical and political context of your project through discussion with the people maintaining the system.

**Source**  [Demeyer et al., 2008, chapter 3]

A.2 Read All the Code in One Hour

**Intent**  Assess the state of a software system by means of a brief but intensive code review.

**Source**  [Demeyer et al., 2008, chapter 3]

A.3 Interview During Demo

**Intent**  Obtain an initial feeling for the appreciated functionality of a software system by seeing a demo and interviewing the person giving the demo.

**Source**  [Demeyer et al., 2008, chapter 3]
APPENDIX A. THUMBNAIL PATTERNS

A.4 Build Prototypes

**Intent**  Test requirements ad design decisions to reduce risk of wasted cost and missed expectations by building a prototype.

**Source**  [Coplien and Harrison, 2005, page 49]
Appendix B

Pattern Format

This appendix describes the format of the patterns listed in Part III. The format is inspired by the patterns from Demeyer et al. [2008].

B.1 PatternName

Also known as ‘another name for the pattern’

Problem
The problem is phrased as a simple question. Sometimes the context is explicitly described as well.

This problem is difficult because

- here we discuss the forces because they tell us why the problem is difficult and interesting.

Yet, solving this problem is feasible because

- here we pinpoint the key to solving the problem.

Solution
The solution is a compact statement explaining how to solve the problem. It sometimes includes steps, examples or hints that can help you apply the solution.

Trade-offs

+ We list the positive trade-offs.

- But also the negative trade-offs.
APPENDIX B. PATTERN FORMAT

Known Uses
We list some known uses of the pattern, either from our own experience by referencing a part of the dissertation, or from literature by referencing other work.

Related Patterns
We list some related patterns, mostly to show the interaction between the different patterns from this dissertation.
Appendix C

Pattern Catalog

This appendix lists the patterns from Part III and their intent.

**Gather Available Facts** *(pattern 8.1)*
Obtain facts from a software system without having to extract them yourself.

**Perform Text-Based Fact Extraction** *(pattern 8.2)*
Obtain facts from the source code of a software system without having to fully parse it.

**Buy a Commercial Fact Extractor** *(pattern 8.3)*
Obtain detailed facts without having to write your own tool support.

**(Ab)use a High-Level Monitoring Tool** *(pattern 9.1)*
Obtain coarse-grained run-time information with little impact.

**(Ab)use a Debugger** *(pattern 9.2)*
Obtain fine-grained and detailed run-time information on a small scale.

**Instrument Source Code** *(pattern 9.3)*
Obtain run-time information in the absence of sufficient tools.

**Quick Feasibility Study** *(pattern 10.1)*
Quickly decide whether a proposed approach is feasible.
Bibliography


BIBLIOGRAPHY


[Demeyer et al., 2008] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns.* Square Bracket Associates, 2008. [4, 9, 12, 14, 35, 50, 51, 89, 141, 143]


BIBLIOGRAPHY


