Bit-to-board analysis for IT decision making

Łukasz Kwiatkowski

This research was supported by the Netherlands Organisation for Scientific Research (NWO) under the Jacquard project number:

638.004.405 *EQUITY: Exploring Quantifiable Information Technology Yields*

ISBN: 978-90-8659-624-9

VRIJE UNIVERSITEIT

Bit-to-board analysis for IT decision making

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op maandag 19 november 2012 om 15.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Łukasz Marcin Kwiatkowski

geboren te Kraków, Polen

promotor: prof.dr. C. Verhoef
copromotor: dr. R.J. Peters

Preface

Only my name appears on the cover of this book. However, there are many others whose support was invaluable in the process of carrying out the research described here. I would like to mention my promoter Chris Verhoef and co-promoter Rob Peters as they were key to delivering this book. My interaction with Chris was very rich in discussions that shaped this thesis. Frequent coffee and chocolate meetings with Rob passed in a nice atmosphere and were filled with plenty of insightful debates. I am very thankful to Rob for helping me come up with the Dutch summary of the thesis. I would also like to mention Steven Klusener here. Had Steven never introduced me into the EQUITY project my PhD journey probably would have never commenced.

I very much appreciate the efforts of the thesis committee members: Prof. Dr. Mark van den Brand, Prof. Dr. Willem Jan Fokkink, Prof. Dr. Bert Kersten, Prof. Dr. Paul Klint, and Harry Sneed. Primarily for their useful feedback at the last stage of my work. And, also for their exceptionally efficient collaboration in preparation of the defense.

My work on the PhD was set among great people who created lots of opportunities to enjoy the time. I want to mention: Bartek, Asia and Wojtek, Paweł and Kasia, Ela, Radek, Banan, Michał, Wojtek, Ania and Marcin, Ania and Jarek, Agnieszka and Przemek, Iwona and Janek, Sławek, Rafał and Kasia, Przemek, Tasos, Neuromancer, Laurenz, Erald, Peter, Vadim, Niels. My gratitude also goes to those whom I forgot to list here. Thank you for your company and support.

Moreover, I would like to express my gratitude to my family members. They provided constant support and cheered me up along the way despite the distance that separated us. My rare trips back home always allowed me to recharge my batteries. And, that was often indispensable.

Last but not least I would like to thank Agnieszka. It is hard to express how grateful I am for accompanying me in this journey and supporting in all sorts of ways. I certainly owe you an apology for the fact that even though most of the time I was physically present I often had my mind elsewhere. We will have to make it up.

Contents

Preface **i**

Contents **iii**

1 Introduction **1**
1.1 Excavating facts . . . 1
1.1.1 Locating the site . . . 2
1.2 IT in business . . . 2
1.2.1 IT management . . . 3
1.2.2 Decision making . . . 3
1.3 Popping the hood . . . 5
1.3.1 Bit level bits 6
1.3.2 ... and the board-level impact . . . 8
1.4 Management data source . . . 8
1.4.1 Tackling *big data* . . . 9
1.5 EQUITY research . . . 9
1.5.1 Context for this thesis . . . 10
1.6 Thesis outline and contributions . . . 11
1.6.1 Recovering management information from source code . . . 11
1.6.2 Reducing operational costs through MIPS management . . . 12
1.6.3 Samenvatting . . . 13

2 Recovering management information from source code **15**
2.1 Introduction . . . 15
2.2 Management treasury: codebase . . . 21
2.2.1 Management quandaries . . . 21
2.2.2 Information basis . . . 22
2.2.3 Case study . . . 23
2.2.4 Source code facts . . . 24

2.3 Codebase construction 28
2.3.1 Technology 28
2.3.2 Analysis facility 30
2.3.3 Separator 31
2.3.4 Metrics computation 34
2.3.5 Historical data 36
2.3.6 Code generators 38
2.4 Screening the data 43
2.4.1 Data normalization 44
2.4.2 Portfolio coverage 45
2.4.3 Plausibility checks 46
2.5 Information recovery 50
2.5.1 Function Points 51
2.5.2 IT benchmarks 52
2.5.3 Portfolio dynamics 55
2.6 Managerial insights 58
2.6.1 Size structure 58
2.6.2 Essential information 61
2.6.3 Vendor-locks 66
2.6.4 Nuts and bolts 71
2.7 Discussion 76
2.8 Conclusions 77

3 Reducing operational costs through MIPS management 79
3.1 Introduction 79
3.1.1 Targeting source code 81
3.1.2 Business reality 81
3.1.3 A light-weight approach 83
3.1.4 Related work 84
3.1.5 Organization of this chapter 85
3.2 MIPS: cost component 85
3.2.1 MIPS and MSU 86
3.2.2 Transactions 87
3.2.3 Database impact 87
3.2.4 CPU usage sources 91
3.3 DB2 bottlenecks 91
3.3.1 Performance 91
3.3.2 Playground 92
3.3.3 Potentially inefficient constructs 94
3.3.4 Getting the data 99
3.4 Reaching for code 105
3.4.1 Implementation of transactions 106
3.4.2 Portfolio exploration 107
3.4.3 Code properties 110
3.5 MIPS-reduction project 113

3.5.1 Project scope . . . 113
3.5.2 Selected modules . . . 114
3.5.3 Changes . . . 115
3.5.4 Impact analysis . . . 119
3.5.5 MSU reduction . . . 120
3.5.6 Summary . . . 123
3.6 Portfolio-wide control . . . 124
3.6.1 DB2 code . . . 125
3.6.2 Major MIPS consumers . . . 127
3.6.3 Low-hanging fruit . . . 130
3.6.4 Improvement scenarios . . . 134
3.7 Practical issues . . . 137
3.7.1 Phase I . . . 137
3.7.2 Phase II . . . 138
3.8 Discussion . . . 139
3.8.1 Vendor management . . . 139
3.8.2 CPU resource leaks . . . 141
3.9 Conclusions . . . 142

4 Samenvatting **145**
4.1 Broncode: een goudmijn van management informatie . . . 145
4.2 Case studie 1: strategisch IT management . . . 148
4.3 Case studie 2: verlaging operationele kosten door MIPS management . . . 150

Bibliography **153**

CHAPTER 1

Introduction

1.1 Excavating facts

We live in the present and make decisions which affect our future. Most of us crave for making decisions that lead to achieving the previously assumed goals. It is possible to make decisions *ad hoc* and in this way attain the desired results. Such an approach, however, is characterized by a great uncertainty and requires clairvoyance. Sadly, experts with such sense are rarely encountered in nature and therefore we need to be able to resort to some decision making process which offers the capability of getting sound outcomes. One way to achieve that is to incorporate in the decision making process relevant information; analyze it, draw conclusions, and based on these take actions. Clearly to follow this path it is crucial to have the relevant information at our disposal. Unfortunately, this approach is frequently hampered by one fundamental challenge: the needed information is not always readily available. Also, it is not straightforward how to obtain the information.

The drive to gratify the need for information is omnipresent in all sorts of domains. Take archaeology as an example. The main task of archaeologists is to study past human activity. This process involves mainly the recovery and analysis of the remaining material evidence which may include graves, buildings, tools, pottery of our ancestors, and other bits and pieces. The analysis delivers information that allows developing a picture of how people used to dwell, how they professed their religion, how they supplied food and water, why they inhabited specific areas, etc. The main objective of the archaeologists is to gain understanding of our history and provide, among other things, with grounds on how our contemporary civilization can make good decisions. That way we can, for instance, improve our existing processes with the know-how of the past or avoid repeating mistakes of our ancestors.

1.1.1 Locating the site

Archaeologists strive for basing their theses on evidence. That desire naturally spurs efforts to acquire the needed evidence. One of the popular approaches to gathering material evidence is the process of excavation. An archaeological excavation typically commences with locating the site. The visible sites do not always lie exposed on hilltops. The Angkor Wat temple complex in Cambodia, the Taj Mahal mausoleum in India, an ancient Inca fortress Machu Picchu in Peru, the pyramids in Egypt or the Colosseum in Rome are more of an exception than the rule. Very often excavation is a down to earth process which involves the removal of soil, sediment, or rock that covers artifacts. Finding the right location to begin extraction is a process that requires knowledge and experience. One may, for instance, use references from the ancient literature to locate lost cities. Regardless of the approach taken it is crucial that processing of the designated site delivers artifacts that help obtaining the sought after information.

Artifacts are crucial not only for archaeologists. In the process of obtaining reliable information in the domain of IT management one must also resort to analyzing artifacts. As it turns out the site to excavate the evidence needed here is reachable by many IT decision makers without much effort. There is no need to dig deep to reach for an abundance of bit level artifacts to support board level IT decision making.

1.2 IT in business

IT and business have become critically aligned. According to the Economist Intelligence survey on business resilience 47% of the respondents said that survival of their businesses could be seriously jeopardized if they experienced less than a day of downtime from their information systems [152]. And, according to another survey by US National Archives and Records Administration, 25% of companies that suffered from a failure relating to IT-systems lasting two to six days went bankrupt immediately [152]. To illustrate the order of the related financial implications in Table 1.1 we present the benchmarked hourly cost of downtime from IT for various business activities. The data was prepared by the Fibre Channel Industry Association and taken from [153].

Business activity	Industry	Hourly downtime costs
brokerage operations	finance	$6,450,000
CC authorizations	finance	$2,600,000
pay-per-view	media	$150,000
home shopping (TV)	retail	$113,000
catalog sales	retail	$90,000
airline reservations	transportation	$90,000
tele-ticket sales	media	$69,000
package shipping	transportation	$28,000
ATM fees	finance	$14,500

Table 1.1: Information systems hourly downtime cost per business activity.

Enterprises must adapt themselves to changing circumstances in order to survive. Changes to IT follow accordingly. From the business point of view every IT endeavour should be seen as an investment which bears risks for the organization. The risks include project failures, maintainability difficulties, time-to-market delays, reputation loss, or even market value destruction [55]. Moreover, depending on the type of business some organizations must make additional efforts to align their business management practice with legal requirements. For instance, in the Basel II accords on banking laws and regulations the concept of operational risk was introduced [116]. IT is listed as one of the risk components. That situation obviously puts IT in a special position since it demands from the affected organizations transparency with regard to how IT matters are handled.

The impact of IT on the business is unquestionable. The related costs are high. Risks involved have far reaching consequences. To contain these aspects organizations must act with adequate vigilance. The bottom line is that IT must be effectively managed.

1.2.1 IT management

Proper management of IT is paramount for a successful business. The way in which banks use and manage their IT is of crucial importance for their performance [9]. Research conducted at McKinsey suggests that spending more on IT does not guarantee better returns. Their research involving banking sector also suggests that those banks that are economical with IT make better profits [4]. Strassmann in his extensive research has shown that there is no clear correlation between profits and IT spending [145, 146, 147, 148]. An illustration of this phenomenon is presented in Figure 1.1. The relationship between IT spending and returns is based on data from the banking sector, the plot is taken from the IEEE EQUITY 2007 keynote presentation of Strassmann [150].

1.2.2 Decision making

Managing IT inherently involves decision making. Similarly as in management of other entities there are different layers at which the decisions are made. There exist three commonly distinguished management layers: strategic, tactical, and operational. Let us characterize them and put them in the IT management perspective.

Strategic Strategic decisions concern the general direction and involve, for instance, long-term goals, visions, philosophies or business values. They are made at the highest level and affect the long-term direction of the business. Due to their nature they are often characterized with high uncertainties. Within organizations the strategic decisions relating to IT matters are typically made by the top level executives such as CIOs or CFOs. And, involve answering questions like: what hardware to use for running software applications, what technologies to use in development, to develop in-house or outsource this process, should maintenance be outsourced, what to do to reduce spending, etc.

Tactical Tactical decisions are meant to support strategic decisions. They focus on intermediate-term issues. The ultimate purpose of these decisions is to let the business

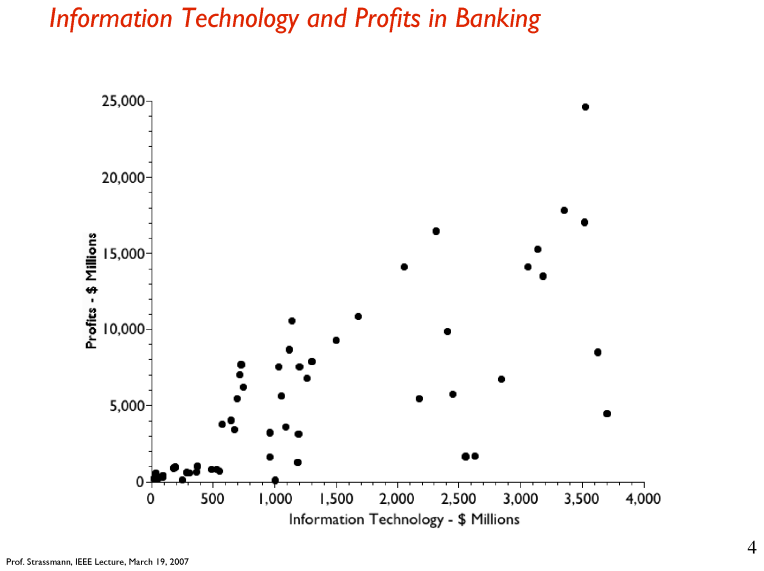

Figure 1.1: Relationship between IT spending and returns on the basis of data from the banking sector.

move closer to attaining the strategic goals. They are typically made by the middle level managers.

To give an example of a tactical IT decision let us consider the following scenario. Suppose that a strategic decision is to cut operational costs. Such a goal vows for tactical decisions which are likely to include choice of the area of cost cuts and the method in which the cuts will be realized. For an organization that runs its software on a cluster of mainframes the decision could be to concentrate on the reduction of the CPU resource consumption. In these environments the reality is such that the customers pay hardware usage fees based on the amount of workloads committed to their CPUs. Therefore one way to achieve savings is to optimize workloads. And, this could be achieved, for example, through adaptations to the source code of the executed applications. More specifically, the kind of adaptations that improve

runtime performance of the applications and lower the CPU usage as a consequence.

Operational Operational decisions are used to support tactical decisions. They are normally part of day-to-day activities. Frequently they can be preprogrammed or set out clearly in policy guidelines. Most employees are in charge of operational decisions.

To give an example of an operational decision in the IT context let us suppose that the reduction of the CPU resource consumption through source code improvements is the tactical decision. The related operational decision could involve, for instance, preparation of a report outlining the top CPU intensive software components and the associated source code modules.

As Tom deMarco puts it: *You can't control what you can't measure* [25]. Bit level data is the fundamental piece of input needed to incorporate control [85]. And, forms grounds for obtaining information required at each of the decision making layers. In real-world applications it turns out that obtaining the information is not always simple.

There appears to exist a difficulty with collecting relevant data [15]. In [132], a global survey of 1375 executives, 60% of the respondents points the lack of data integration as the barrier to the organization's ability to grow business. The realities of large organizations are such that the data is being widely spread among different people and databases. This fact alone suggests that data, if available, is not readily accessible. Moreover, corporate environments make the process of obtaining the data hectic. The omnipresent bureaucracy impedes effective access to data, especially when different business units or companies are involved. These are just some of the problems that exist within large contexts. Nonetheless, the dire demand for an effective decision making process spurs the search for novel approaches to obtaining data. And, in particular, for the kind of approaches that fit into the realities of large business environments. Just like archaeologists have their means for obtaining grounds to build knowledge about the human past the IT executives require suitable means to manage.

1.3 Popping the hood

One element of IT which is essential in data processing is software. For some types of businesses the software propelled data processing simply supplants the old fashioned (manual) procedures. This is certainly the reality for retail banks, insurance companies, hospitals, governmental institutions, etc. For other types of businesses data processing is the key driving force behind revenue generation. Here consider, for instance, the internet based businesses; frequently referred to as *dot-coms*. For companies like google.com, facebook.com, amazon.com, or ebay.com to name a few, the process of creating value proposition heavily depends on data processing. It is important to mention that these companies take on huge data sets. Data sets that are so large (petabytes) and complex that it becomes awkward to work on them using classical database management tools and simple infrastructures. A jargon has been even invented to characterize these abundant data sets as *big data*. This reality clearly indicates that *dot-coms* would not be able to exist without proper IT.

As another example take securities trading companies. These businesses have advanced to utilize computer technologies to give their participation in the financial markets a boost. Modern computer infrastructure made it possible to do trading electronically. While it used to be floor traders responsible for generating liquidity in the markets these days they have become superseded by computers. It is common to learn that it is software that is responsible for spotting the opportunities and executing transactions. For these businesses it is also software that is used to crawl through large sets of market data and aid in the process of designing various trading models. As reported in Forbes [51] Morgan Stanley bank tried to do some portfolio analysis using their traditional databases and grid computing and it found out that it was impossible to scale to the very large volumes of data that its data scientists wanted to use. The requirements were met when an adequate software framework was deployed.

1.3.1 Bit level bits ...

Behind the implementation of software applications there are a lot of artifacts involved. They include requirements documentation, design diagrams, project reports, test plans, performance test results, etc. There is also source code. The same software application can be written using various technologies and also in an innumerate number of ways. These development choices are not without meaning for the characteristics of the resulting software. To illustrate it let us consider source code level details of a software facility used for sorting numbers. To set the stage let us assume that we choose *C*, one of the mainstream programming languages, to implement this facility [89]. Also, to show how the characteristics of the software product may differ let us consider two possible implementations of the facility. The differentiating element will be the function used to sort the elements.

Let us consider the following two *C* functions. Each function implements the same functionality. Namely, given a zero-based array of integers it sorts the elements according to their natural ordering in an ascending order.

```
1  void insertSort(int* a, int size){
2      for(int i=1; i<=size; i++){
3          int j=i-1;
4          while(j>=0 && a[j]>a[j+1]){
5              int tmp=a[j];
6              a[j]=a[j+1];
7              a[j+1]=tmp;
8              j--;
9          }
10     }
11 }
```

```
1  void quickSort(int* a, int first, int last){
2      if(first<last){
3          int pivot=a[first];
4          int left=first;
5          int right=last;
6
7          while(left<right){
8              while(left<=right && a[left]<=pivot) left++;
```

```
9               while(left<=right && a[right]>pivot) right--;
10
11              if(left<right){
12                  int tmp=a[left];
13                  a[left]=a[right];
14                  a[right]=tmp;
15              }
16          }
17          a[first]=a[right];
18          a[right]=pivot;
19
20          quickSort(a,first,right-1);
21          quickSort(a,right+1,last);
22      }
23  }
```

The first code fragment implements the sorting function which follows the so-called *Insertion sort* algorithm. The other lists the implementation of a function that accomplishes the same task, however, there the sorting process follows another method, known as *Quicksort*. For details on how these algorithms work the interested reader is referred to literature [20, 21]. The two code snippets clearly differ in terms of the number of lines of code. Another major difference lays in the runtime performance of the code. For the purpose of our illustration we only focus on this aspect.

In computer science theory the Big-O notation is a commonly used tool to classify runtime performance of the algorithms [20, 21]. In essence the tool allows for associating with an algorithm a class of functions that best models its runtime given the size of the input. Using this notation it has already been shown that the *Insertion sort* algorithm is expected to sort numbers in a polynomial time ($O(n^2)$ with n being the number of elements to sort), and the *Quicksort* is likely to deliver results in $O(n\log(n))$ [20]. So, in principle we should expect that for a sufficiently large input the sorting facility equipped with the *Quicksort* would perform faster. Naturally, the theory finds its reflection in practice.

	time (seconds)	
no. of elements (n)	*Insertion sort*	*Quicksort*
1,000	0.001	0.000
5,000	0.022	0.001
10,000	0.086	0.002
50,000	2.177	0.008
100,000	8.695	0.020
500,000	217.207	0.269
1,000,000	899.863	0.826

Table 1.2: Comparison of the runtime performance of the two sorting functions.

In Table 1.2 we present the results of applying the two sorting *C* functions to a number of arrays of varying length, each of which initially contained randomly distributed integer numbers. The experiment was executed on a unix-based machine with Intel(R) Core i7 CPU (Q720) 1.60 Ghz with 4 GB of RAM. The first column contains the size of the array given for sorting. The size is expressed in the number of integers. The second column gives the measured times of sorting the array using *Insertion sort* function, and the third column the times obtained with the *Quicksort* method. Times are given in seconds. Not

surprisingly, as the number of integers to sort increases the time requirement changes for both functions. Clearly, *Quicksort* outperforms *Insertion sort* when the size becomes significantly larger.

1.3.2 ... and the board-level impact

Let us now put the above illustration into business perspective. Sorting is one of the many elementary operations that business software performs on a daily basis. Each time we create a record for a new client, generate a report, explore the graph of connections between users in a social network, build a list of recommended products to buy, or price financial instruments in a split of a second, there are dozens of this and other low level operations involved. Whether software is written in modern programming languages like Java, C#, C++, Objective C, Ruby, Perl, Python, PHP, Visual Basic, etc., or with legacy ones like Ada83, mumps, Jovial, CHILL, Cobol, Fortran, PL/I, etc., the way the source code is written will have its effect on the software's performance. This will affect the overall business performance, and costs of the IT infrastructure. Ideally, the software is optimized so that this infrastructure costs are as low as possible and yet accommodates the business requirements.

Take as an example the social networking giant, *facebook.com*, of which the 2012 IPO turned out to be the biggest in history of technology firms. This company made at certain point a decision regarding its website's codebase written in PHP programming language and translated it into a heavily optimized C++ codebase [176]. This transformation, which was done automatically by means of an internally developed tool (*HipHop*), allowed the company to improve performance of the website and also reduce operational costs. The company's engineers reported that the conversion allowed for a reduced average CPU consumption on Facebook by roughly 50 percent [119]. In plain financial terms this means millions of dollars in savings.

Not only implementing software in an efficient manner is important for the business. Decisions regarding programming language have its impact for the business as well. For instance, choice of a programming language has its consequences for costs relating to development and maintenance. Take as an example the availability of programmers specialized in certain technologies. The availability varies from technology to technology. Where there is a shortage of specialists the costs of development, and further maintenance, can be high. The bottom line is that the decisions made at this very technical bit-level bring about concerns at the very top of the business, the board-level.

1.4 Management data source

Warren Buffett, a famous investor, once said [2]:

> *Always read source (primary) data rather than secondary data.*

Although his advice is addressed to the financial markets investors the message it carries is very relevant from the IT management point of view. It is vital that information used in IT decision making originates from unfiltered data. This property is not necessarily

satisfied when information is obtained from the secondary IT data such as project or software documentation, high-level overviews, consultancy firm reports, outsourcing vendor reports, etc. The nature of secondary data is that it is prone to imprecision. The natural consequence of it is that the information obtained may not reflect the true condition of the IT.

We follow Buffett's advice in the process of obtaining information useful for IT management. Namely, we choose to reach after the primary data. Locating the source of the primary data resembles the task of locating the site by the archaeologists. In our context we chose to rely on the fact that software is one of the key components comprising the IT infrastructures. And, the essential element behind its implementation is source code. So, similarly to what the archaeologists do when excavating artifacts from the ground we reach for artifacts which lay deep under the hood of the IT-infrastructure; the source code.

Source code due to its nature can be seen as a repository of textual data. This makes it suitable for analysis and extraction of human interpretable data. We treat source code as the primary data source and analyze it to retrieve bit-level data for board-level purposes (bit-to-board). Our research is carried out in the context of organizations which are large, international, employ many people, and thus are complex. As a consequence the related IT infrastructures are also non-trivial. We demonstrate that the data we extract from source code is useful in supporting the decision making process in such a context.

1.4.1 Tackling *big data*

Archaeological excavation is a daunting and laborious task. Source code analysis of a large industrial IT-portfolio also looms as a repelling task. Although analysis of source code may indeed present hardships, it is possible to avoid the well known meanders. We strive for being pragmatic so that our research is reusable in an industrial setting. Working with a large IT-portfolio inevitably boils down to having to deal with large amounts of data. To get to data hidden in source code and allow for simple scalability we approach source code analysis purely in a lexical manner.

In our work we employed widely available technologies to implement our data extraction and analysis facilities. For the major part we relied on the programming language *Perl* and numerous auxiliary tools provided by the unix shell such as sed, awk, grep, etc. [173, 172, 48]. To process the large amounts of data extracted from the source code and other bit-level data sources we used the MySQL database engine [54]. Analysis of the data, which included obtaining statistical characteristics, graphs plotting, and use of the Latent Semantics Indexing was carried out by means of the generally available software tools for mathematical and statistical analysis: R and Matlab [74, 164]. We show that the development of our facilities does not pose challenges and that they are suitable for usage within the context of large organizations.

1.5 EQUITY research

This thesis is a product of the EQUITY research project. The name EQUITY stands for Exploring QUantifiable IT Yields. The objective of the project is to study relationships between yields and information technology. The underlying assumption is that decision

making with regard to IT must be done in a calculated manner on the basis of information derived from facts [169, 170, 171]. The exploration areas are driven by the real-life problems encountered in the IT dependent organizations, and based on real-world case studies.

As part of the EQUITY project a number of extensive articles has been produced [35, 98, 99, 104, 121]. The issues they cover include the quantification of the effects of requirements volatility, quantification of the quality of IT forecasts, or a method to quantify the yield of an IT investment portfolio in an environment of uncertainty and risk.

1.5.1 Context for this thesis

The research presented in this thesis is 100% empirical. We concentrate on information retrieval from source code implementing information systems comprising IT-portfolios of large organizations. Particularly, we focus on the kind of information which is useful in supporting decision making processes at all three management levels: strategic, tactical, and operational. Although for the major part we focus on addressing issues at the strategic and tactical level we also discuss in-depth the process which led us to obtaining the results. That way operational decision making is covered as well.

In the domain of IT management we concentrate on the main areas: cost reduction and risk mitigation. The particular questions that drove our explorations include the following. How do the low level source code properties translate into risks for the software portfolio maintainability? What are the code characteristics of the business critical information systems? What is the relationship between source code improvements and the costs of running applications in the mainframe environment? What are characteristics of the top CPU resource consuming applications in a large software portfolio? In the presented explorations we rely on the data which originates from the analyzed industrial source code. We also consulted about our findings with domain experts, available manuals, and any available internal documentation.

Acknowledgments This research received partial support by the Dutch *Joint Academic and Commercial Quality Research & Development (Jacquard)* program on Software Engineering Research via contract 638.004.405 *EQUITY: Exploring Quantifiable Information Technology Yields* and via contract 638.003.611 *Symbiosis: Synergy of managing business-IT-alignment, IT-sourcing and off-shoring success in society*. We would like to thank the organization that will remain anonymous for kindly sharing their data with us. We are also indebted to the employees of the IT department for providing us with their support. Patrick Bruinsma from IBM for offering his knowledge on the mainframe environments. Jean Kleijnen and Paul Robb for sharing with us their expertise on IT portfolio management. Last but not least, we are very grateful to Capers Jones and the anonymous reviewers for commenting on the content of this thesis.

1.6 Thesis outline and contributions

The remainder of the thesis contains three chapters. Each chapter is self-contained and can be read independently. We now summarize each chapter. In particular, we discuss the content, outline contributions, and explain the origin of the chapter.

1.6.1 Recovering management information from source code

In this chapter we show how to employ source code analysis techniques and recover management information. In our approach we exploit the potential of the concealed data which resides in the source code statements, source comments, and also compiler listings. We show how to depart from the raw sources, extract data, organize it, and utilize it so that the bit-level data provides IT executives with support at the portfolio level. Our approach is pragmatic as we rely on real management questions, best practices in software engineering, and also IT market specifics. We enable, for instance, an assessment of the IT-portfolio market value, support for carrying out what-if scenarios, or identification and evaluation of the hidden risks for IT-portfolio maintainability. Our approach was deployed in an industrial setting. The study is based on a real-life IT-portfolio which supports business functions of an organization operating in the financial sector. The IT-portfolio comprises Cobol applications run on a mainframe with the total number of lines of code amounting to over 18 million. The approach we propose is suited for facilitation within a large organization. It provides for a fact-based support for strategic decision making at the portfolio level.

Contributions of this chapter

- We share with the scientific community the realities of a large industrial software portfolio in which Cobol is the dominant programming language and some of the actively used code modules date back to 1960s.
- By following the Exploratory Data Analysis approach we exhibit an abundance of data hidden inside source code which can be automatically extracted and used to obtain management information.
- We discuss in detail how we implemented a light-weight facility to aid the extraction process from both structured (e.g. source code) and unstructured (e.g. source code comments) data sources.
- We show that by using our approach it is possible to analyze costs and risks, carry out what-if scenarios, etc., and that way obtain meaningful information that IT decision makers can use.

Parts of this chapter are based on the following article.

Ł.M. Kwiatkowski and C. Verhoef, *Recovering management information from source code*. Article accepted for publication on July 27, 2012. To appear in Science of Computer Programming.

1.6.2 Reducing operational costs through MIPS management

In this chapter we focus on an approach to reducing costs of running applications. MIPS, which is a traditional acronym for millions of instructions per second, have evolved to become a measurement of processing power and the CPU resource consumption. The need for controlling MIPS attributed costs is indispensable given the high percentage they involve in the operational IT costs. In this paper we investigate a large mainframe production environment running 246 Cobol applications of an organization operating in the financial sector. We found that the large majority of the top CPU intensive operations in the production environment involve the use of DB2. We propose portfolio-wide efforts to reduce CPU resource consumption from the source code perspective. Our technique is low-risk, low-cost and involves SQL code improvements of small scale. We show how to employ, in an industrial setting, analysis of a mainframe environment to locate the most promising source code for optimization. Our approach relies on the mainframe usage data, facts extracted from source code, and is supported by a real-world SQL tuning project. After applying our technique to a portfolio of Cobol applications running on the mainframe our estimates suggest a possible drop in the attributed monthly CPU usage by as much as 16.8%. The approach we present is suited for facilitation within a mainframe environment of a large organization.

Contributions of this chapter

- We present results from the examination of a large portfolio of IMS-DB2 software applications executed in the mainframe environment in which costs attributed to CPU usage constitute a significant cost component for the organization. Using our observations we motivate SQL code improvements as a way to lower platform usage costs.
- We examine a past SQL-tuning project carried out in this portfolio. That project was limited to a small number of selected IMS-DB2 applications from the environment and allowed us to study the nature of code changes that resulted in reduction of CPU resource consumption. We present details of our analysis of this real-world project.
- We outline the complexities and risks involved in the implementation of a source code improvement project encountered in the studied environment and address these by proposing an adequate approach.
- We propose an approach to plan CPU resource consumption reduction projects in a large codebase by identifying low-risk high-yield SQL-tuning opportunities using source code analysis and code heuristics.

Parts of this chapter are based on the following article.

Ł.M. Kwiatkowski and C. Verhoef, *Reducing operational costs through MIPS management*. The article has been submitted for publication to a peer-reviewed journal and is under revision.

1.6.3 Samenvatting

This chapter provides a summary of the thesis in Dutch.

CHAPTER 2

Recovering management information from source code

2.1 Introduction

Information technology plays an important role in many organizations. Worldwide IT related expenditure has been on the rise. Gartner reports that in 2008 spending on information technology was at $3.4 trillion [73]. This means a threefold growth since 1996 when the spending was estimated at $1.076 trillion [149]. For IT dependent organizations IT related costs constitute a significant cost component. For Dutch banks it was estimated that total operational IT costs oscillate at around 20%–22% of total operational costs [12]. IT investments of government regulated organizations require transparency. When in 1996 the Clinger-Cohen Act was enacted by the US Congress the CIOs in government institutions were compelled to adopt a portfolio approach to IT investments [12]. Later the Sarbanes–Oxley (SOX) Act of 2002 aligned IT governance with corporate governance hence requiring from IT investment decisions transparency [77]. The soaring dependency of organizations on IT, high costs and risks of the IT investments, or constant demand for changes imply that IT-management must be equipped with means to support decision making. META Group, the benchmarking research company, pointed out that IT-portfolio management is the only method by which organizations can manage IT from an investment perspective [109]. One aspect of mature decision making is access to management information [25, 85]. It turns out that IT executives are commonly faced with the problem of its omission.

Management information Gathering IT knowledge is an economically rational activity as it develops organizational intelligence [93]. Many IT-intensive organizations lack even the slightest insight into their IT-portfolios. In fact, the average IT-executive manages a business-critical IT-portfolio without knowing important aspects of its IT costs or risk drivers. INSEAD's survey shows that 60% of CFOs and CIOs do not know the size of their core software assets [32]. According to Jones' data 75% of IT organizations world-

wide are at the lowest capability maturity model level (CMM), which is a five point scale for ranking the maturity of an organization's software process [120]. Level 1 means that no repeatable process is in place, in particular, there is no overall metrics and measurement program. In a 2001 survey 200 CIOs from Global 2000 companies were asked about their measurement program. More than half (56%) said they did high-level reporting on IT financials and key initiatives. Only 11% said to have a full program of metrics used to represent IT efficiency and effectiveness, the rest (33%) said they had no measurement program at all [124]. Whereas the introduction of a metrics program looms as a simple remedy for the problem the reality shows a different picture. Since 1988, the ratio of starts to successes has remained remarkably consistent at about four in five metrics programs failing to succeed [122]. One reason for the lack of IT measurement programs is the difficulty with the collection of relevant data [15]. The sheer size of the organizations results in data being widely spread among different people and databases, and hence difficult to obtain [132]. In some cases the data is not just distributed, it is plainly missing and one must resort to surrogate sources to obtain it.

The business environment demands answers from IT-executives to questions which are not easy to tackle. Consider, for instance, the assessment of asset value in mergers and acquisitions scenarios. To capture the market value of an IT-portfolio one requires information which enables first embracing the IT-assets and then deriving pricing figures. These tasks are not straightforward especially when information systems built long ago are at stake. If they lack functional documentation it is a challenge to size them and estimate their market value. Another IT management challenge is to keep the portfolio maintainable. Most of the long-ago built information systems share one property; they are business-critical. Without them organizations cannot survive, yet their haphazard decades-long evolution makes them difficult objects in their own right [13, 127, 159]. Successful maintenance of these systems requires proper mitigation of risks entrenched inside technology. Operations such as compiler upgrades, code generators license renewals, or code optimizations are needed from time to time. However, to carry out these operations so that the related IT risks are under control one requires adequate insight. In order to obtain information which is up-to-date and unfiltered we propose to reach for source code [168].

Source code The term IT-portfolio embraces various entities relating to IT such as projects, computer systems, hardware, licenses, people (knowledge), networks, and more. Information needs are likely to differ for each of these entities. Let us consider computer systems as an example. Nowadays computer systems are composed of COTS, cloud computing services, legacy software, etc. Clearly to capture relevant information that is of value for managers it is necessary to take a more fine-grained focus. The focus in this chapter is on the portfolio of software applications used to support business operations of an organizations. And, for this entity we propose to reach for source code and recover from it management information.

Source code constitutes the nuts and bolts for software. On its own it forms a rich textual repository which is suitable for analysis and data retrieval. There are numerous reasons to use source code as a proxy for recovering management information. First, unlike other sources of data this one is almost always available. Second, by studying the

source code it is possible to collect technology level characteristics of the IT-portfolio that help justifying maintenance costs and expose maintenance risks. For instance, it has been observed that modules with a rich history are candidates for either a volatile part of the business or for error-prone corners of the portfolio [50]. This is important to executives since appropriate action is needed; error-prone modules are the top-ranked factor degrading maintenance productivity (with 50%) and can cost as much as five times more than high-quality modules [81, pp. 400–402]. Third, source code analysis provides insights that cannot be obtained by scrutinizing typical project data or even functional documentation of the systems. For instance, code complexity or presence of obsolete constructs are not normally reported as part of software documentation. And finally, with the source code in place there are also source comments available. As we will show in this chapter the data hidden there allows, for instance, getting insight into volatility in the portfolio.

IT portfolios have evolved on top of products and services offered on the IT market. The structure of the IT market has made organizations dependent on hardware and software vendors. Over the past decades software development in the domain of management information systems (MIS) has been dominated by Cobol. Studies show that Cobol is used to process 75% of all production transactions on mainframes. In the financial industry it is used to process over 95% of all data [8]. A 2009 Micro Focus survey claimed that the average American interacts with a Cobol program 13 times a day and this includes ATM transactions, ticket purchases and telephone calls [41]. Cobol constantly evolves and adapts itself to the latest technology trends [126, 76]. This is especially visible in case of web enabled mainframe applications, where the modern web front-ends cooperate with the legacy back-end systems. Cobol development is associated with all sorts of products such as compilers or code generators. With respect to that the IT market dictates its own rules which businesses must abide to. And it means, for instance, upgrading code generation tools or assuring that the source code syntax complies with the standards supported by the compilers available on the market. Managing all that is not easy and requires reliable data, and source code analysis is the means to provide it.

Goal of the chapter In this chapter we present an approach to recover management information from source code. We show how to depart from the raw sources, extract data, organize it, and eventually utilize so that the bit level data provides IT executives with support at the portfolio level (bit-to-board). To assure that our approach is pragmatic we rely on real management questions, best practices in software engineering, constitutional knowledge about the corporate IT environment, management level reports, feedback from experts, and public IT benchmarks.

Scientific contributions of this chapter are two-fold. First, recovering management information. Second, applying source code analysis techniques to extract data required to reach our goal: recover management information. Naturally, the first cannot be done without the other. The novelty of our approach lays in the application of source code analysis techniques to recovery of management information in the context when a large and business critical software portfolio is at stake. In the chapter we show how we reach our goal in a manner that makes it possible, without much effort, to implement our approach for other portfolios. We discuss the process step by step to encourage others (IT managers

in large organizations) to adapt it for their purposes.

This chapter extends the work presented in a series of many articles [169, 170, 171, 35, 98, 99, 104, 121, 100] where IT-governance issues are continually discussed with executives in many companies. Research presented in this chapter was carried out within the EQUITY project (Exploring QUantifiable IT Yields) where industrial partners discuss their particular questions with the research team. For the sake of confidentiality we cannot mention these questions explicitly. Nonetheless, from many of these discussion sessions we extracted the information needs of the executives. Subsequently we dove into the literature to related work, and we found an interesting column by Zvegintzov [178] in which these major information needs turned out to be discussed under the common theme of *frequently begged questions*. We embark on these in detail later on in the chapter. The managers responsible for the software portfolio used in our work confirmed that the questions are of interest for them. We therefore used the questions outlined by Zvegintzov as a framework of reference.

Case study In this work we analyze a Cobol software portfolio of a large organization operating in the financial sector. The Cobol sources are a mixture of code written manually and generated with Computer-Aided Software Engineering (CASE) tools, such as TELON, COOL:Gen, CANAM, and others. Normally, obtaining the sources of the entire production environment is not a routine job. Some organizations lack adequate version control, configuration, and release management practice and tool support. In this case it was not a problem to receive code since this was under strict version, configuration and release management. We took as input for our analyses compiler listings of the latest production version of the portfolio. The portfolio is decades-old and large in many dimensions; for example, in terms of lines of code, number of systems, or number of modules. To give an idea, the portfolio contains more than 18.2 million physical lines of code (LOC) partitioned over 47 information systems.

Data analysis In this chapter we discuss an empirical study. Using Dibbern et al classification we can characterize our work as empirical descriptive [27]. Namely, we did not set any theoretical grounding for the studied phenomenon, but we focused on presenting factual accounts of the event and used it to illustrate how to derive from it information of interest. There are a number of ways to approach data analysis [154, 111, 113]. Typically data analysts employ classical and exploratory approaches [114]. In both of them one departs from a question and arrives at a conclusion using data. The core difference is in the sequence and focus on the data analysis steps. The classical approach imposes models on the data, for instance, regression models or analysis of variance models (ANOVA). On the contrary, the exploratory data analysis (EDA) approach does not impose models, but instead focuses on the data itself; its structure, outliers, and models suggested by the data. Also, EDA approach makes little or no assumptions on the data. This is unlike in the classical approach where models can be applied after the required assumptions have been validated. While classical techniques are very quantitative in nature the EDA techniques are generally graphical, and involve tools like scatter plots, histograms, probability plots, and tables.

We chose EDA to analyze software portfolios since very often in the process of deriving measurement goals from information needs and metrics from measurement goals reveals that a lot of data is missing and many context-specific adaptations are necessary. For instance, software maintenance has many aspects and depending on the purpose (such as monitoring or prediction), different data is necessary. With EDA we freely look into available data, make no assumptions, take measurements, report on observations and based on these make recommendations. Our insights into the software portfolio are expressed primarily through graphical means.

We exploit the potential of the concealed data which resides in the source code statements, source comments, and also compiler listings. We want our approach to be easily reusable by others. Therefore, we elaborate on the extraction process in detail, in particular, the non-trivial task of analyzing the loosely structured textual content of the source comments. The source code analyses we employed rely on lexical approaches. We used the widely available technologies to implement data extraction. For the major part we relied on the programming language *Perl* and numerous auxiliary tools provided by the Unix shell such as sed, awk, grep, etc [173, 172, 48]. To handle the large amounts of data extracted from the source code we used MySQL database engine [54]. Analysis of the data, which included obtaining statistical characteristics, graphs plotting, and use of the Latent Semantic Indexing (LSI); was carried out by means of the software tools for mathematical and statistical analysis: R and Matlab [74, 164]. On the basis of a number of examples we demonstrate how deployment of our approach helps in embracing various IT risks and costs. Our technical framework was used also in delivering an approach to achieving reduction in CPU resource consumption in another large industrial case study. We reported on that effort elsewhere [100].

Portfolio insights Our approach enabled us to provide various managerial insights into the IT-portfolio. We partitioned the portfolio into information systems and approximated systems' size. We were also able to obtain the indication of the growth rate of the portfolio. For instance, as it turned out the amount of source code in the studied portfolio expands annually by as much as 8.7%. We show how we estimated the portfolio market value, assessed the cost of operations, and the staff assignment scope. Using the recovered information we conducted a number of what-if scenarios for the portfolio to project future managerial indicators. We exposed various technology related challenges for the top executives. For instance, as it turned out maintenance of almost 60% of the portfolio source modules depends on expensive CASE tools. Almost 40% of the modules rely on the no longer supported IBM OS/VS COBOL compiler. Approximately 15% of the modules suffer from excessive code complexity. By analyzing various technology migration scenarios we found that the top critical business applications are impacted in a relatively high degree. Furthermore, the complexity of the migrations is non-trivial. For instance, the Cobol implementation of the top critical systems involves as many as 4 different code generators. Throughout the chapter we elaborate on how we arrived at these and other findings. All the presented insights are discussed in the context of the organization which operates on the studied IT-portfolio.

Related work Source code analysis is omnipresent in many IT contexts. There is an extensive amount of work done in the area of software architecture reconstruction [7, 96, 115, 142, 143, 144]. Much work is devoted to automated software modifications [95, 162, 155, 156, 131, 163, 92]. The related work also focuses on the cost aspect which in the industrial setting plays a paramount role [38, 129, 166]. For instance, in [94] the authors emphasize on the cost reduction factor that code analysis adds to a software transformation project. We already presented elsewhere [100] how to employ source code analysis to support reduction of operational costs relating to MIPS, and how to attain it at the portfolio level. In that paper we elaborated in detail on the code analysis process and illustrated it on a large industrial IT-portfolio. In this chapter we also analyze code of a large IT-portfolio. However, here the goal is to recover information that helps making costs and risks more transparent, and allows for fact based strategic decision making.

Code analysis in the industrial setting is strongly dependent on adequate tooling [130, 157, 101]. In case of IT-portfolios containing legacy Cobol applications the code analysis is often a daunting task due to the large multitude of Cobol language dialects that are around [128]. Analysis involving grammar based parsing is especially difficult. We propose in this chapter to rely on lexical analysis so that the recovery of management information is inexpensive and less cumbersome. In that respect our approach is similar to the one taken towards rapid-system understanding presented in [160]. In that paper only the source *statements* are analyzed. We additionally propose to analyze source *comments* to obtain meta information which is normally available in external software documentations. To allow for generic and effective analyses we developed a set of lexical tools which are accurate enough to satisfy our purposes.

Work related to the analysis of Cobol code commonly relies on case studies consisting of individual systems. For instance, in [160] the authors analyzed two systems of about 200k lines of code (LOC) from a portfolio of banking systems, or in [46] the analysis was based on a single Cobol system of 200k LOC from a Belgian insurance company. While analysis of real-life code samples gives valuable insights and commonly provides answers to the stated problems when trying to address problems encountered by the executives at the board level it is important to reach for the portfolio of systems as a whole to obtain an organization-wide view. Similarly as in [100, 136, 161] we also analyze a large IT-portfolio. One of the potential reasons that related work deals with smaller portfolios is the reluctance of companies to disclose all the sources of their core assets, that are considered their trade secrets. We were fortunate to have access to a large industrial portfolio and were thus able to use it as our case study.

Organization of this chapter This chapter is organized as follows. In Section 2.2 we provide details on the IT-portfolio we used as a case study, outline the commonly encountered management questions, and show what data we extract from the portfolio sources to fuel our analyses. Section 2.3 elaborates on how we approached automated extraction of data in an industrial portfolio. We show results of applying our tooling to the studied portfolio and characterize the extracted data in Section 2.4. In Section 2.5 we present how to recover information from the code extracted data bits. In Section 2.6 we illustrate how to use the bit-level data to obtain board-level insights. In Section 3.8 we discuss how our approach fits into portfolios comprising programming languages other than those en-

countered in the case study. Finally, in Section 3.9 we conclude our work and summarize findings.

2.2 Management treasury: codebase

In this section we discuss the data potential that source code exhibits. First, we present the common quandaries that IT-executives face and explain how by reaching for the source code these become resolvable. Next, we present the studied real-world portfolio that we will use throughout this chapter. Finally, we discuss which source code parts we use to fuel the process of management information recovery.

2.2.1 Management quandaries

The fact that IT has become strongly embedded in the business processes makes it a dissociative component of strategic business management. Often IT related decisions have far reaching consequences for the business. Managing IT requires finding answers to questions that typically revolve around the problems of optimal budget allocation, business reputation protection, mitigation of risks relating to software asset maintainability, information systems up-time, performance improvement, operational cost reduction, etc. In all of the encountered dilemmas one requires information which, if not leads directly to a solution, at least supports the process of attaining it. To reach for the information one must collect the relevant data, analyze them and properly interpret.

IT executives tend to have relatively simple sounding questions. However, most of the time the solid answers are missing. In [178] Zvegintzov discusses the most frequently begged questions by the decision makers. Here are some of them:

- How much software is there?
- Which languages are used to write software?
- How many software professionals are there?
- How old is software?
- How good is software?

As he points out the answers to those questions only seem to be well-known. They are commonly based on few sources which often turn out to be thin and unreliable. As Zvegintzov suggests the most reliable answers to empirical questions originate from a representative sample of real organizations or systems [178].

The demand expressed in the question *How much software is there?* has clear costs related motivation. Costs relating to, for instance, software development or maintenance, are directly linked to the amount of software involved. Given an existing portfolio of software applications it is useful to know: how much money is expected to be paid to keep the portfolio operational or how much staff is required to keep it operational. Naturally, estimates for the future expected costs are also valuable. To do this one must have some

basis for estimating the growth rate of a portfolio. In some situations it is required to capture the market value of the software that company owns. To arrive at this figure one might consider estimating the rebuild costs of the portfolio. In some cases executives are required to supplant the existing systems with newer versions. Here not only the information on how much it might cost to rebuild systems is handy, but also how many people should be involved, and how likely it is that the rebuild fails. Information of that sort are of particular importance to people like CTOs or CFOs who allocate budgets.

To help CTOs and CFOs obtain the information they need we analyze the available data. We follow the EDA technique to obtain information that characterize software size. To measure software size it is possible to resort to source code and count lines of code. Obtaining the growth rate of a portfolio would typically involve reaching for some historical data on software changes. Here source code version management systems come as the most natural data source. We will show that when this data source is not available it is still possible to get surrogate data useful in estimating growth rate of the portfolio by analyzing source code. With EDA we look at general questions, analyze the available data, and propose the possible answers that contribute to decision making.

What is also desired to know is *Which languages are used to write software?*. That is important since the more different technologies there are in the portfolio the more complex software maintenance is. For instance, there is more knowledge required to update software. The question *How old is software?* is motivated by the will to assure IT continuity. Those applications which were written long ago and are still in use are likely to require supplanting to make them maintainable using contemporary workforce and technologies. The question *How good is software?* is motivated by the need to be able to swiftly respond with software adaptations to changes in the business environment. Having the information that allows portfolio level executives find possible answers for their generic questions is key in order to knowledgeably make decisions.

2.2.2 Information basis

To support IT executives in challenging questions like those posed above and other we derive input data from source code. We argue that source code is the prime unbiased source of data that enables gaining lucidity in the condition of IT. In its nature source code is a repository of various textual data. Therefore, by reaching for it we are able to derive characteristics of the IT-portfolios' software applications from their actual state. This way we assure that any information recovered from source code is based on facts. As we will show the obtained facts are well suited to form insights useful in IT-management.

Let us recall that we chose EDA technique to analyze the software related data. Following EDA we freely look into available source code extracted data, make no assumptions, take measurements, report on observations and based on these make recommendations. Our insights into the software portfolio are expressed primarily through graphical means. To illustrate the mechanics behind our approach we obtained numerous insights into the portfolio which we formulated through histograms, density plots, tables, etc., and interpreted them in managerial terms with Zvegintzov's questions in mind. We want to emphasize that the insights we present in the remainder of the chapter do not exhaust all possible views that IT decision makers may opt for. For instance, in [100] we presented

on the basis of a large-scale portfolio how to obtain other insights and provide support in reducing costs linked to running software on mainframes. With our approach we aim at presenting how source code analysis can be incorporated into IT-management practice.

2.2.3 Case study

We investigated the IT-portfolio of a large financial organization. The portfolio is business-critical as many business processes are implemented in it and millions of clients are served worldwide through it. The entire portfolio comprises systems written in Cobol and all of them run on a mainframe. We reached for compiler listings rather than the original source code since they are more informative. They not only contain the original source code but also carry auxiliary data such as the compilation date, the name of the compiler, or its version, just to name a few. The listings were produced by various compilers. The code they contain originates from preprocessors, CASE tools or is hand-written. Each listing corresponds to some Cobol module in the portfolio. To obtain a first impression of the IT-portfolio in Table 2.1 we present some of its characteristics.

Portfolio property	Count
Modules (Cobol programs)	8,198
Lines of code (LOC)	18,075,013
Lines of comments (CLOC)	4,464,177
Oldest module	1967
Information systems	47

Table 2.1: Characteristics of the IT-portfolio.

Let us explain the content of Table 2.1. All 8,198 Cobol files provide together over 18 million of lines of code. Among the lines of code we distinguished comments. Their count is denoted by CLOC. Comments in natural language contain various technical and contextual information reported by the programmers. It is often like an evolution history but then not in a version management system but as a comment header within the source modules. For instance, it contains dates of modules creation, modification, names of programmers or companies modifying the code, or reasons for modification. In principle, the more lines of comments we have the larger the chance to learn more about the code. In this case almost 25% of the lines of code turned out to be comments. It is our experience that usually this is about 10% or less but then the code is often of mediocre quality as well. From the comments we found out, for instance, that the oldest module dates back to 1967. Using source code only we were also able to determine the number of information systems. By adequately interpreting the source files naming convention we found in total 47 systems in the portfolio. Throughout the remainder of this chapter we will elaborate on how we processed this huge amount of source code and recovered more than the basics listed above.

2.2.4 Source code facts

To structurally present our approach towards analysis of the portfolio we introduce two concepts that we will use throughout this chapter: source code fact and codebase. A source code fact is any piece of data that is extractable from the source code and useful in characterizing some of its properties. Examples of source code facts are the total number of lines of code of a module, the date of the module's inception into the portfolio, or the name of the code generator used to produce it. For our analysis we distinguish two types of source code facts: metadata and metrics. A metadata is a source code fact that provides context information. For instance, the code modification date, characteristics of the development or production environments, etc. A metric is a property of the actual source code. It can be, for instance, the total number of lines of code or the number of occurrences of a particular keyword. We will refer to the collection of source code facts associated with all the modules comprising the portfolio as the *codebase*.

Metadata To learn about the portfolio's history, its evolution, or the technology context in which it operates we used metadata. This way we were able to equip ourselves with surrogates for the data which is typically present in the project databases, version control systems, project descriptions, etc. To retrieve metadata we used the compiler appended auxiliary data, source code comments, and the source statements. With each module m in the portfolio we associated a set of metadata. We first summarize the metadata that is both extractable and relevant for our analysis, and then treat the less obvious ones in greater detail. To show the origins of the metadata values we provide anonymized code fragments taken from the studied portfolio.

- Name (*N(m)*): filename of the compiler listing which corresponds to the module m.
- Module creation (*DC(m)*): date of creation of the module m.
- Changes (*CHG(m)*): set of dates characterizing historical changes done to the module m.
- Last compilation (*LC(m)*): date of the last compilation of the module m.
- Compiler (*CN(m)*): name of the compiler last used to compile the module m.
- Code generator (*GEN(m)*): name of the code generator used to produce the module m, if the code was auto-generated.

Module creation In Cobol there exist special language constructs which enable programmers to structurally record certain information in the code of the programs. The `IDENTIFICATION DIVISION` of a Cobol program permits inclusion of the following optional statements: `DATE-WRITTEN`, `AUTHOR`, `INSTALLATION`, `DATE-COMPILED`, `REMARKS` and `SECURITY` [103, 76, 140], by some referred to as program *documentation statements*. Each of these statements serves as a label which precedes a comment. For instance, the `DATE-WRITTEN` statement is

meant to label the date of a module's creation. Figure 2.1 illustrates the usage of the documentation statements.

```
00001  ID DIVISION.
00002  PROGRAM-ID.    AA123.
00003  AUTHOR.        JACK FLIST.
00004  INSTALLATION.  COMPANY X.
00005  DATE-WRITTEN.  FEB 23,1972.
00006  DATE-COMPILED. APR  6,2000.
00007  REMARKS. THE PROGRAM FETCHES A USER-DATA RECORD FROM THE MAIN FILE
```

Figure 2.1: Fragment of a Cobol program containing the *documentation statements*.

The first line in Figure 2.1 opens the Cobol program with the obligatory `ID DIVISION` statement followed by a dot. In the second line the name of the program is specified in the Cobol's `PROGRAM-ID` paragraph. The lines 3 through 7 contain the documentation statements: `DATE-WRITTEN`, `INSTALLATION`, `AUTHOR`, `DATE-COMPILED`, and `REMARKS`. Each of which is followed by a comment provided by a programmer. Even though these statements are already obsolete they are still frequently encountered in the code [76, 140]. For the purpose of our analyses we utilize the comments labeled with the `DATE-WRITTEN` statement to obtain the creation dates for the portfolio modules.

Changes Source code version history is typically found in version control systems, for example, CVS or SVN. We found that an abundance of data of this kind is also present in the source comments. In Figure 2.2 we present an example of a sequence of commented lines of code in which history of changes is reported.

```
00015 * * * * * * * * * VERSION 63 J. NILE MAY 1975 * * * * * * * * * *
00016 * * * * * * * * * VERSION 64 A.J.SINATRA AUGUST 1976 * * * * * * *
00017 * * * * * * * * * VERSION 66 BILLY BOLT * NOV 1979 * * * * * * *
00018 * * * * * * * * * VERSION 67 JOHN DICKSON * OCT 1980 * * * * * * *
00019 * * * * * * * * * VERSION 68 M.A. SMITH   * JAN 1985 * * * * * * *
00020 * * * * * * * * * VERSION 69 XG500 * NOV 1991 * * * * * * *
00021 * * * * * * * * * VERSION 70 F. TRIMMER * DEC 1992 * * * * * * *
00022 * * * * * * * * * VERSION 71 C.COMB    * MRT 1999 * * * * * * *
00023 * * * * * * * * * VERSION 72 A.B. CREST   * JUL 2000 * * * * * * *
```

Figure 2.2: Fragment of the version history log extracted from a Cobol module.

In the example presented in Figure 2.2 each line consists of the Cobol editor line count, a sequence of stars in which the first indicates that the line contains a comment, a version number, name of a programmer, and finally a modification date. The fragment presented here is one of the few cases of very neatly formatted code documentation. The fact that comment embedded entries are not restricted to follow any particular convention results in having to analyze a free format text. This obviously complicates data extraction. However, by applying our analysis approach it

turned out to be possible to extract variations and normalize them into a single format so that the data is appropriate for statistical analysis. Hence making it possible to extract the comments written evolution history, much like doing archeology. For our purpose we only extract the dates as we found them particularly useful in constructing managerial insights. Given a module m we form content of the $CHG(m)$ set by extracting from the module's comments as many dates as our tools determine to be valid version history entries.

Compiler and last compilation Even though the syntax of the Cobol programming language provides a documentation statement for reporting the compilation dates in the code of a module, through an optional keyword `DATE-COMPILED`, we extract this date from the time stamp generated by the compiler in the listing. For the studied portfolio this is the true date of the last compilation prior to migrating a module into the production environment. In addition, we also extract the name of the compiler.

```
1 1PP 5648-A25 IBM COBOL for OS/390 & VM  2.2.2                 PROG02    Date 26/07/2004  Time 13:52:01   Page     3
2    LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8  Map and Cross Reference
3 0  000001         000100 IDENTIFICATION DIVISION.                                        00010004
4    000002         000200 PROGRAM-ID.                                                     00020001
```

Figure 2.3: Fragment of a compiler listing containing compiler appended metadata.

To illustrate how the data is expressed in the studied portfolio we present in Figure 2.3 a fragment of one of the compiler listings. The first line contains the aforementioned compiler appended auxiliary data. In the remaining lines we find the source layout description (line 2) and a portion of the Cobol code (lines 3 and 4). Let us analyze the first line in more depth. After the first two blocks, 1PP and 5648-A25, we find a name of the compiler used for the compilation of the Cobol module. In this case it is IBM COBOL for OS/390 & VM version 2.2.2 . The name and version are followed by the name of the compiled module, PROG02, and a detailed time of the compilation, Date 26/07/2004 Time 13:52:01. The remainder of the line provides the compiler listing page label.

Code generator Apart from hand-written code, there also exists generated code. Traces of the use of CASE tools are typically present in the source code.

```
00001 *TELON-------------------------------------------------------------
00002 *DS: B02                                          | COPY REMARKS
00003 *-----------------------------------------------------------------
00004 *    PROGRAMMA : AA123
00005 ******************************************************************

00006 *  Source code generated by CANAM Report Composer V4.1.20
```

Figure 2.4: Examples of signatures of Cobol code generators.

In Figure 2.4 we present two examples of CASE tools produced comment entries. A comment fragment generated by the CA TELON code generator is shown in lines

1–5. Line 6 was produced by the CANAM Report Composer V4.1.20. Comment lines like those presented here are used to obtain values for the GEN metadata. Those modules for which no CASE tools specific comments are found are classified as hand-written.

Due to the nature of the source code elements from which we retrieve the metadata their availability cannot be guaranteed for every source file in the portfolio. One exception here is the metadata dealing with the compilation process. Nevertheless, we found that we were able to extract metadata for a significant portion of the portfolio, and this sufficed for our analysis purposes.

Metrics To measure the quantitative aspects and capture the technology related portfolio risks and costs we collected source code metrics. We restricted ourselves to a set of metrics which suffice to encompass the aspects studied in this chapter. The metrics were computed on the basis of the Cobol source code after it was separated from the corresponding compiler listing. With each module m in the portfolio we associated a set of metrics. We first summarize the metrics and then treat them in greater detail.

- Lines of code (*LOC(m)*, *SLOC(m)*, *CLOC(m)*, *BLOC(m)*): The total count of the lines of code of particular types in the module m.
- Obsolete language constructs (*OBS_x(m)*): The total number of occurrences of an obsolete language construct x in the module m.
- Code complexity (*MC(m)*): The cyclomatic number for the module m.

Lines of code The simplest way to measure the amount of code is to count lines of code in the source files. Given a source file we consider all of its lines of code and obtain totals for the specific types of lines; namely the source lines, the comment lines, and the blank lines. We denote them by $SLOC$, $CLOC$, $BLOC$, respectively. The total number of physical lines of code of a source module m is denoted by $LOC(m)$, and it is the sum of the $SLOC(m)$, $CLOC(m)$, and $BLOC(m)$ metrics. Whereas in our analyses we will rely mainly on the $SLOC$ metric the remaining ones are used as auxiliary characteristics of the code.

Obsolete language constructs In order to capture risks dealing with maintainability of the IT-systems we screen the code for the presence of the obsolete programming language constructs. To give an idea why they are undesired let us consider as an example the `ALTER` statement. In Cobol 85 [140] we find the following point:

> "[..] Use of the `ALTER` statement results in a program that is difficult to understand and maintain. It provides no unique functionality since the `GO TO DEPENDING` statement can serve the same purpose."

The statement was declared obsolete in ANSI standard Cobol 85, and eventually removed from the latest standard of Cobol 2002 [140, 76]. In Cobol there are several constructs which have been dubbed obsolete. To guide the process of screening the code we rely on the list of the obsolete elements of Cobol 85 taken from [140, p. 327]. We identified 15 such constructs and labeled them with integers 1 through 15. The integers served as labels (x) for the OBS_x metrics.

Code complexity A common approach to capturing code complexity is to measure the McCabe's complexity metric [107, 108]. The value of McCabe is also referred to as the cyclomatic number. The metric expresses the number of linearly independent execution paths through a program. Despite the fact it does not embrace code complexity aspects comprehensively the metric remains popular within the industry [39, 133], mainly for the simplicity in which it can be computed. We use $MC(m)$ to denote the cyclomatic number of a module m.

Of course, by no means the set of metrics presented here exhausts all the possible measurements of the code aspects. This particular set supports us in resolving the quandaries posed in this chapter. Whenever analysis of the source code is deemed to contribute to unraveling additional dilemmas other metrics should be considered. With our framework in place this is not a large investment in terms of time and/or effort.

2.3 Codebase construction

In this section we present the approach we took to analyze the IT-portfolio. Our first step in the process of management information recovery is the construction of the codebase. Because of the sheer size of the IT-portfolio it was close to impossible to grasp data hidden in the code without the help of automated support. Due to the fact that the compiler listings were our point of departure we designed our tooling in a manner which enabled us to use the listings as input for the analysis. Of course, for other IT-portfolios the approach may differ, and it will depend primarily on the form in which the sources are available (e.g. compiler listings, original source modules, etc.), but also on the programming languages used, or availability of the contextual information. Nevertheless, the principles behind the design of the tooling presented here remain unchanged for other case studies. We begin with presenting an overview of the analysis facility we developed. We then discuss implementation details so that others can reproduce our results for their portfolios. In particular, we elaborate on the separation of the source code from the compiler listings, extraction of metadata from both the listings and the source code comments, and also the collection of source code metrics.

2.3.1 Technology

Typically, there are two routes in which analysis of source code is approached: grammar-based and lexical. The first approach enables expression of the analyzed source code in terms of its language syntactical elements and involves grammar-based parsing of the

source code. The latter operates at the level of regular expressions that model sequences of tokens or characters. Grammar-based approach can be quite involving, especially when legacy Cobol sources are involved. Let us recall, in our case we deal with the source code which has been developed since 1960s. Repositories containing Cobol source code, or sources written in other mature programming languages, are commonly a mixture of innumerate programming language dialects. This fact alone makes the grammar-based parsing route a very daunting process which has a potential to result in having portions of the portfolio code unparsed. Our approach to recovery of management information does not require code to be parsed in order to extract the data we need. Furthermore, we want to take advantage of the data hidden inside the source comments. This requires reaching for custom approaches to text analysis which cannot be accomplished by means of the publicly available parsers. To facilitate our requirements we needed to opt for lexical approach to code analysis.

To instantiate the process of data extraction we sought for a suitable technology. Software application portfolios of large organizations comprise millions of lines of source code. In our case study, we had to analyze over 18 millions of lines of code. It is well observed that when industrial projects are in place the following attributes are required from the tools utilized in software analysis.

- Scalability and robustness: the technology must be suited for IT-portfolios containing systems built of millions of lines of code.
- Reconfigurability and tolerance: the technology used for tools creation must allow for reasonably simple adaptations.
- Support for pretty printing: the technology must enable generation of output in a customizable manner.

We took a uniform approach to data extraction. Developing a code analyzer capable of handling possibly multiple ways of extracting the same type of data requires experimentation and therefore technology which is suited for rapid prototyping is a necessity. In fact, some of the implemented data extraction algorithms went through a number of adjustments at the design level. We also needed to create a suitable representation of the extracted metadata and computed metrics to be able to provide them as input to the data analysis tools. Here, the feature of pretty printing was vital.

Given the stated requirements we opted for *Perl* as our main implementation vehicle [172, 173]. *Perl* contains a strong regular expressions processing engine and therefore it is adequate to accomplish our tasks. Data analyses were accommodated using R and Matlab [74, 36]. In order to instantiate the codebase for the studied portfolio we used the MySQL database engine [54]. We found this open-source product sufficient to accommodate our needs. Furthermore, it provided us with a powerful database engine.

Our technology choices pose certain limitations. Lexical code analysis is in principle characterized with less precision than full parsing approach. Prone to imprecision are typically those analyses which are context sensitive. Nonetheless, the kind of metrics we extract from source to answer our questions do not require delving deeply into the syntactic structure of source statements. Apart from obtaining metrics from source code we

also committed to extracting code change history from source code comments. Here we must accept imprecision. After all we parse natural language texts which not necessarily follow any formatting standards unlike data stored in history version control support systems like CVS/SVN. In fact, with our approach we recover data which might have never been stored in any database.

2.3.2 Analysis facility

We automated the process of building the codebase by developing a source code analysis facility. The facility considered the compiler listings as input, and generated as output SQL scripts which contained the extracted source code facts. In order to simplify the analysis process we split it into two phases. In the first phase the preprocessed code of the Cobol modules was separated and the compiler added metadata extracted. In the second phase the analysis of the separated Cobol code was performed. This included extraction of the additional metadata and collection of the source code metrics. In Figure 2.5 we visualize the analysis process.

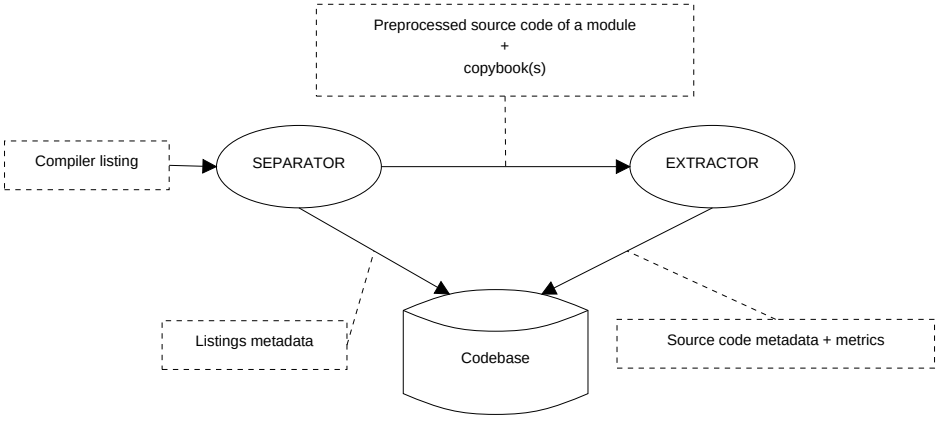

Figure 2.5: Compiler listing analysis process.

Initially each compiler listing is processed by the *Separator*, indicated in Figure 2.5 by a tagged ellipse. The *Separator* isolates the preprocessed Cobol source code, extracts the embedded copybooks, and collects the compiler added metadata. As a result for each listing the *Separator* generates the corresponding Cobol module which includes the preprocessed code of the originally compiled Cobol program, any copybooks referenced in the Cobol program, and the related metadata originating from the compiler listing. The metadata is recorded in a form of the SQL insert statements. The recovered Cobol module along with its copybooks are stored for later analysis by the *Extractor*. The *Extractor*, indicated in Figure 2.5 by the second tagged ellipse, analyzes the source files obtained by the *Separator*. The tool extracts the remaining metadata and collects the metrics. Eventually, it produces a script with the SQL insert statements containing the extracted data.

From the implementation point of view both the *Extractor* and the *Separator* were written as *Perl* scripts. The codebase was modeled as a relational database. To allow for

reproduction of our approach we now present implementation details behind the analysis facility. In certain cases we use regular expressions to facilitate our explanations. Readers not familiar with these constructs yet interested in them are referred to the literature [172, 173].

2.3.3 Separator

The role of the *Separator* is twofold. First, it is responsible for separation of the preprocessed source code from the compiler listings. Second, it carries out extraction of the metadata. We now explain the two operations in detail.

Compiler listing We begin with the presentation of the structure of the compiler listings we dealt with in our study. In Figure 2.7 we present a fragment of a compiler listing originating from the IBM COBOL for OS/390 & VM compiler. As we see apart from the preprocessed source code the compiler listing also contains an abundance of auxiliary data. Lines 1–3, 6–19, and 22 apart from the preprocessed lines of Cobol code also contain compiler added prefixes and suffixes. Of course, to obtain the source code which is suited for computation of the metrics these lines must be properly trimmed and separated. Lines 4 and 20 contain detailed characteristics of the compilation process. For example, it is possible to identify the name of the compiler, its version, date and time of the compilation. These lines are important since they serve the processes of detection of the compiler listing type, more specifically the layout of the compiler listing, and provide input for metadata extraction.

Layout detection Prior to code separation or metadata extraction it is important to determine the compiler listing type. This step is indispensable since layouts of the listings differ from compiler to compiler. This implies that different code separation rules need to be applied. To recognize the type of the compiler listing we used the auxiliary data added by the compilers during the compilation process. In Figure 2.7 lines 4 and 20 give examples of such data. The data of this kind was present in all of the compiler listings. By manually browsing through randomly selected compiler listings we identified four different types of listings. On their basis we designed regular expressions to match the particular types of the compiler listing. In Figure 2.6 we present a *Perl* code fragment from the separator's script containing definitions of the regular expressions.

Figure 2.6 shows four *Perl* variables to which the regular expressions are assigned. These regular expressions sufficed to successfully identify the type of each of the compiler listing we had at our disposal. Of course, the way in which we arrived at the regular expressions did not guarantee that all possible compiler listing types get covered. For other portfolios it may be the case that the manual inspection of the compiler listings does not immediately reveal enough examples to obtain all of the needed regular expressions. This is likely to happen when the portfolio is highly versatile in terms of compilers used. It is obviously possible to iteratively browse the unidentified portion of the compiler listings. Each time a new compiler listing type is found and the corresponding regular expression formulated the matching listings can be filtered out. The process can be repeated until

```
$lst_type1 = '^1PP 5740\-CB1 RELEASE 2\.4\s+(IBM OS\/VS COBOL\s+JULY  1\, 1982)\s+\d{1,2}\.\d{1,2}\.\d{1,2}  DATE (\w{3})\s+(\d{1,2})\,(\d{4})$';
$lst_type2 = '^1PP 5648\-A25 (IBM COBOL for OS\/390 \& VM  2\.\d\.\d)\s+Date (\d{2})\/(\d{2})/(\d{4})  Time [\d\:]{8}   Page[\s\d]+$';
$lst_type3 = '^1PP 5655\-G53 (IBM Enterprise COBOL for z\/OS  3\.3\.1)\s+[\w\d]*\s+Date (\d{2})\/(\d{2})/(\d{4})  Time [\d\:]{8}   Page[\s\d]+$';
$lst_type4 = '^1PP 5688\-197 (IBM COBOL FOR MVS & VM   \d\.\d\.\d)[\s\w\d]+DATE (\d{1,2})\/(\d{1,2})\/(\d{4})  TIME [\d\:]{8}\s+PAGE[\d\s]+$';
```

Figure 2.6: Regular expressions used to recognize compiler listing types.

```
   000318         031600     03 MSGNUM                PIC X(6).                            03160001  BLW=00008+8E0,0000140 6C
   000319         031700     03 ERRORMSG              PIC X(80).                           03170001  BLW=00008+8E6,0000146 80C
   000320         031800                                                                   03180001
1PP 5655-G53 IBM Enterprise COBOL for z/OS  3.3.1               NC011     Date 09/10/2005  Time 14:05:54   Page    10
   LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8  Map and Cross Reference
   000321         031900/** MSG-1 ***                                                      03190001
   000322         032000     COPY MSG1.                                                    03200002
   000323C              ***************************************************************** 00000101
   000324C              *** NAME :MESSAGE 1                                                00000201
   000326C              ***************************************************************** 00000501
   000327C               01  MSG-1.                                                        00000601  BLW=00008+938         0CL32745
   000328C                   03 IMS-HEADER.                                                00000701  BLW=00008+938,0000000 0CL13
   000329C                    05 AA-VALUE USAGE COMP-8 PIC S9(3).                          00000801  BLW=00008+938,0000000 2C
   000330C                    05 BB-VALUE             VALUE +0 USAGE COMP-8                00000901  BLW=00008+93A,0000002 2C
   000331C                                            PIC S9(4).                           00001001
   000332C                    05 IMS-CODE             PIC X(9).                            00001101  BLW=00008+93C,0000004 8C
   000333C                    05 FILLER               PIC X(1).                            00001201  BLW=00008+944,000000C 1C
   000334C                    05 MESSAGE-TEXT         PIC X(12400).                        00003001  BLW=00008+965,000002D 32700C
   000335         032100                                                                   03210001
1PP 5655-G53 IBM Enterprise COBOL for z/OS  3.3.1               NC011     Date 09/10/2005  Time 14:05:54   Page    11
   LineID  PL SL  ----+-*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|--+----8  Map and Cross Reference
   000336         032200/** NEXT MESSAGE ***                                               03220001
```

Figure 2.7: Fragment of a compiler listing taken from the studied portfolio.

there are no unidentified compiler listings left. When such an approach does not lead to a solution within a reasonable amount of time one must seek for alternative approaches.

Code separation Once the compiler listing type is determined it is possible to carry out separation of the Cobol source code. The separation process involved matching the compiler listing lines against regular expressions capable of isolating the lines of Cobol code. For all compiler listing types the code of the copybooks was clearly distinguishable from the remaining preprocessed code. To separate lines of code for both the modules and the copybooks for each compiler listing type we required a set of two regular expressions. We found two different pairs of regular expressions that described layouts of the four different types of the compiler listings we dealt with. In Figure 2.8 we present a snippet of the *Separator*'s code containing the two pairs of the regular expressions.

```
# SET 1
$cobol_line = '^[\s\-1]\d{5}\s[^C]\s([\d\s]{5}.{0,67}).*';
$cb_line = '^[\s\-1]\d{5}\sC\s(.{0,72}).*[\r\n]';
# SET 2
$cobol_line2 = '^[\s0]{3}\d{6}\s{6}[\d\s]{3}(.{0,72}).*';
$cb_line2 = '^[\s0]{3}\d{6}C\s{5}[\d\s]{3}(.{0,72}).*';
```

Figure 2.8: Regular expressions used to separate Cobol code from the compiler listings.

Each regular expression in Figure 2.8 contains a sequence of characters embraced by the `()` characters. The content of this sequence defines the actual Cobol code of either a module or a copybook, depending on the expression. Before and after each block there are sequences of regular expression operators that define the prefix and suffix of the compiler listing line considered redundant. When processing, the *Separator* first determines the name of the compiler used to generate the given listing. On the basis of this information it chooses the proper regular expressions. Let us assume that the selected pair of regular expressions is contained in the `$cobol_line2` and `$cb_line2` *Perl* variables presented in Figure 2.8. By analyzing each line of the listing, from first to last, the *Separator* attempts to match each line with either of the expressions. If a line matches the expression denoted by `$cobol_line2`, the part of the line embraced by the brackets `()` is appended to the file where the preprocessed code of a Cobol modules is held. From the listing presented in Figure 2.7 the lines 1–3,6–7,19, and 22 would be appended to the Cobol's module. If the line matches the regular expression denoted by `$cb_line2` the line embraced by the brackets `()` is appended to the copybook it belongs to. The name of the copybook is determined on the basis of the `COPY` statement [103] that precedes the copybook content. In case of the listing presented in Figure 2.7 the *Separator* creates a file named 'MSG1' and appends to it properly trimmed lines 8–18. In case none of the regular expressions can be matched, the examined line of the listing is either used for metadata collection, as explained in the next paragraph, or is discarded.

Metadata collection Given a Cobol module m the *Separator* collects the following metadata from the corresponding compiler listing: the name of the module ($N(m)$), the time of the last compilation ($LC(m)$), and the name of the compiler used ($CN(m)$). The

$N(m)$ is identical with the filename of the compiler listing. Collection of the $LC(m)$ and $CN(m)$ metadata relies on the regular expressions presented in Figure 2.6. Each regular expression contains four blocks embraced with `()` brackets. The first block, in all four regular expressions, determines the value of $CN(m)$. The remaining blocks are used to determine $LC(m)$. These remaining blocks represent components of the date: day, month and year. Depending on the compiler listing type we interpreted these values differently. For instance, dates generated by the IBM COBOL for OS/390 & VM compiler were in the format of DD-MM-YYYY, while dates generated by IBM Enterprise COBOL for z/OS were in the format of MM-DD-YYYY. To prepare the extracted dates for further analyses we stored them in the DD-MM-YYYY format.

2.3.4 Metrics computation

In the process of collecting metrics, listed in Section 2.2.4, we considered as input the Cobol programs obtained by the *Separator*. The source code metrics were obtained for each Cobol source file. Information required on the collected metrics was retrieved from [76, 103, 140, 118, 33, 108]. We now explain details of the implementation.

Counting lines of code Computation of the $SLOC$, $CLOC$, $BLOC$, and LOC metrics boils down to counting occurrences of the particular line types in a Cobol source file. Whereas this operation is in general relatively simple we discuss it in more detail to show Cobol specific aspects we took into account. To recognize the different line types we used regular expressions which covered all possible cases encountered in Cobol source files. By way of an example we discuss the regular expression `^.{6}[^\*D].*` utilized for computing the $SLOC$ metric. Let us explain. The `^` character instructs *Perl* to align the regular expression with the beginning of the analyzed line in a source file. A valid Cobol line can be prefixed, the first six characters, with an arbitrary sequence of characters. This space is typically either left blank or filled with a line number. Since we want to exclude all the non-executable lines on the 7th position a test for the presence of `*` and `D` characters is done. For Cobol, the `*` character indicates a comment line, and the `D` character indicates a line of code that is deemed executable only if the `WITH DEBUGGING MODE` clause is specified in the `SOURCE-COMPUTER` paragraph. Otherwise this line is treated as a comment line. Since the total number of lines marked with D was relatively low, 327 in the entire portfolio, we simply disregarded this code as executable. From the 7th position and on, the regular expression accepts any arbitrary sequence of characters. All lines carrying comments, and hence contributing to the $CLOC$ metric, were isolated by the following regular expression: `^.{6}[\*D].*`. Other than lines with the `*` on the 7th position we also regarded as comments lines containing on that position `D`. Blank lines ($BLOC$ metric) in Cobol are those which contain a sequence of spaces, and also optionally a `\` character on the 7th position. These lines were isolated by the following regular expression: `^.{0,6}\?\s*`. Finally, the LOC metric was obtained after computing the $SLOC$, $CLOC$, and $BLOC$ metrics by adding up their values.

Obsolete language constructs To analyze the Cobol sources from the perspective of the presence of the obsolete language elements we used a list outlining such elements

from [140, p. 327]. On this list we found 11 categories of Cobol language elements dubbed as obsolete. These categories embraced 15 Cobol statements.

1. `ALL`
2. `AUTHOR`
3. `INSTALLATION`
4. `DATE-WRITTEN`
5. `DATE-COMPILED`
6. `SECURITY`
7. `MEMORY SIZE`
8. `RERUN`
9. `MULTIPLE FILE TAPE`
10. `LABEL RECORDS`
11. `VALUE OF`
12. `DATA RECORDS`
13. `ALTER`
14. `ENTER`
15. `STOP`

We assigned to the statements indexes ranging from 1 to 15, respectively. In the manual each construct was characterized with a keyword and the context in which its occurrence is considered obsolete. On the basis of these characteristics we designed regular expressions which enabled us to filter out from a Cobol file the relevant occurrences. For a given module m we counted the occurrences of the listed obsolete elements. For each element its total number of occurrences was reported as the $OBS_x(m)$. The x subscript represents the appropriate index associated with the obsolete element.

Cyclomatic complexity In order to measure the cyclomatic complexity of the code it is sufficient to count the so-called branching points in a program's code [108]. The cyclomatic number is then obtained by increasing the total number of branching points by one. The cyclomatic complexity is typically computed for a particular scope within a program, e.g. a procedure. In Cobol the concept of a procedure does not exist as such. We therefore considered the content of the entire `PROCEDURE DIVISION` to determine the cyclomatic number. To compute the number we followed a method used by IBM in the IBM Rational Assets Analyser [59]. The tool approximates the cyclomatic complexity for Cobol programs by counting occurrences of particular language constructs which correspond to the branching points. The total count is increased by one to obtain the number. The method relies purely on the lexical analysis of the code. Occurrences of the following constructs are counted.

1. `EXEC CICS HANDLE`
2. `EXEC CICS LINK`
3. `EXEC XCTL`
4. `ALSO`
5. `ALTER`
6. `AND`
7. `DEPENDING`
8. `END-OF-PAGE`
9. `ENTRY`

10. EOP
11. EXCEPTION
12. EXIT
13. GOBACK
14. IF
15. INVALID
16. OR
17. OVERFLOW
18. SIZE
19. STOP
20. TIMES
21. UNTIL
22. USE
23. VARYING
24. WHEN

Given a Cobol module m its cyclomatic number is recorded in $MC(m)$.

2.3.5 Historical data

Extraction of the metadata with regard to the Cobol files history, namely, date of creation (DC), and the following code changes (CHG) relies on the analysis of the *documentation statements* and the source code comments. In order to determine the value for the $DC(m)$ for a given module m we extracted comments labeled by the DATE-WRITTEN keyword. This step was accomplished by finding the keyword using a regular expression, and isolating the comment part. For those modules for which the DATE-WRITTEN keyword was not found the date $DC(m)$ was marked as unknown.

Extraction of the data characterizing history of changes (CHG) is more involved. Our inspection of the Cobol sources disclosed that comments located in the lines occurring before the DATA DIVISION statement frequently contain records of the relevant data. Of course, records of this kind have a potential to occur in any place in the code or not occur at all. Nevertheless, we used our observation as a heuristic which allowed us to limit the analysis to those comments which are likely to contain the relevant data.

Parsing comments is in general a cumbersome task. Earlier in the example in Figure 2.2 we presented a fragment of the code containing comments in which history of file changes was reported. In that example the data embedded inside the comments follows a clearly deducible layout. However, this example does not illustrate the variety of layouts used across the portfolio. In Figure 2.9 we present a compilation of various comment lines showing that there is a multitude of different layouts in which the change history data was written. All the comments originate from the source files from the studied portfolio.

The following are characteristics of the comments holding source files history records which we formulated after scrutiny of the content presented in Figure 2.9.

- Historical changes are typically characterized by means of three data items: date of change, version number, and name or identifier of a programmer. For instance, in line 1 we find a word VERSION followed by a - and a number, then JOHN,

```
1  00014 * * * * * * * * * * * * * * * * VERSION-57 JOHN JUNE    1999
2  00015 * * * * * * * * * * * * * * VERSION 02 NICK OCT 1978  * * * * * *
3  00020 * * * * * * * * VERSION 07 JACK FLINSTON NOV 1991    * * * * * *
4  00021 * 0100 NICOLE BLUES  01-1999                          *
5  00031 * 01   M. FRANK 280598 NEW
6        *  OCT. 96    04     TON        CHANGE                  *
7        *  SEP. 98    0501   LISA BOLKOT     CORRECTION     *
8  00020 ** **** 240  **    05    ** 08-03-96 ** PHILIP MAYER        **** **
9  00023 ** **** NUE9730    08    ** OCT 1996 ** D. BRICK            **** **
10 00024 ** **** NUE9731          ** 31-01-98 ** A.G.H. CONVERSION-       **
11       * 114  * 04 * FEB 95 * C. WHIM       * VERSION B NEW*
12       *NUE97625*05 * DEC 96 * G. LENDER * QQ0125                  *
13       * NUE9711 *  03 * MAY 96 * D. J. * QG0125                *
14       * CB2P01    *06 * MRT-04 * C.V.R. *                    *
15 003300**  **** ----  **    01    ** 26-11-81 ** R.J. COOK          ****  **
16 002817*   02      301     31-10-96  E. FLOYD           *
17 002821*         NU09610  10-07-95  A. HAM       PHASE 2  *
18 003700*   11    QS74-07  10-06-98  D. POST        CORR IV      *
19 003800*   14    QA1102   14-02-01  G. G.      DESCRIPTION    *
20 001600*    02     0025    06-11-92  B. COOLER.                              *
21 002500*    06    NU9610 19-06-96  M. J. JUNIOR         QG0114 -           *
22 004000*    VERSION 05      06/93            : MINIMUM      *
23 001720*     05    aug'02  D. Dingo             COMPILATION              *
24 00016 * 0102  0600 NICKOLSON   BLOB
25 009800* > 2    FEB 2001    J. MICHAL       CORRESPONDENCE *
26 004693* >11   24 -01-2005  G. FORD          FL     ,                   *
27 004400* > 13   JUL 2002    J. NICK        : NOVEMBER CHANGES  *
28 005800* 11      nov 2002    g. FORD
29 00014 * 01   JUN.1988 DICK TRACY    COMMUNICATION BUGFIX
30       * 01/ APR93 | TON WILD       | NEW                      *
31 003200*  >=02    MRT. 2001    TOM MICE      NUMBER CONV. *
32 00014 * * * * * * * * * * * * * VERSION-50 SEP 75 * * * * * * * * * *
33 00018 * * * * * VERSION 04 JANUARY 1981 BOB BICKER  * * * * * * * * *
34 002000* VERSION     : 03                                                  *
35 002100* DATE        : 07 MAR 2002.                                        *
36 002200* AUTHOR      : FRANK NICK.                                       *
```

Figure 2.9: Compilation of comments originating from different Cobol files containing change history data.

presumably the name of the programmer responsible for change action, and the related date JUNE 1999.

- The three data items are typically recorded in a single comment line, as in lines 1–33. We found a few exceptions. In some files the items are spread over a continuous sequence of lines (max 3). Lines 34–36 show an example of such a record.
- In some cases names or identifiers of the programmers, or version numbers do not occur at all. For instance, in line 32 there is no identifier or name of the programmer.
- There is no particular order in which the data items appear. For instance, in line 2 the version number is the first item whereas in line 6 it is the date that appears as the first element in the comment line.
- The styles in which the dates, names or version numbers are written vary. For instance, we found dates written as follows: JUNE 1999, 31-10-96, 07 MAR 2002. Also we found examples of invalid dates, e.g. 32-01-2003.

Using these observations and the examples of comments we formulated a list of regular expressions to support recognition of the change history records. Given a Cobol module m we considered all the comment lines found before the `DATA DIVISION` as input for the change history extraction. Let us recall, the $CHG(m)$ metadata is a set of

dates. During the analysis we populated the $CHG(m)$ set with the comments extracted dates presumably linked to the code changes. Even though we were also able to capture version numbers and author names we ignored them as we did not utilize them in the portfolio analyses. Of course, if these elements are found useful in fulfilling some particular investigation of a portfolio our tools are ready for their retrieval. The comment lines were analyzed in the order they appeared in the Cobol module. For each comment line the following operations were carried out.

1. The extractor maintained a buffer for storing at most three previously read lines. The currently analyzed line was appended to the buffer.
2. The content of the buffer was treated as a single comment line and matched against each regular expression we designed. This operation resulted in having no regular expression matched, only one matched, or more than one matched.
3. In case of no match the content of the buffer was discarded and no date was appended to the $CHG(m)$.
4. In case of one match the recognized date was appended to the $CHG(m)$.
5. In case more than one regular expression matched a choice of the likely correct match needed to be made. Given the very liberal style in writing comments it is relatively easy to confuse version number with a date, for instance, in line 24 the two numbers which appear in the beginning of the comment line can be interpreted either as dates or version numbers. To identify the likely correct match we introduced a heuristic function. The highest ranked match was used as a valid date and appended to the $CHG(m)$.

Of course, given the fact that comments are an optional component of the Cobol source code for some Cobol modules the extraction process yielded no dates at all. We will elaborate on the portfolio modules coverage later in the chapter.

2.3.6 Code generators

In determining whether the code of a Cobol program was hand-written or generated we also analyzed comments. We relied on the fact that code generators (CASE tools) typically leave in the code particular comment lines. We refer to these lines as CASE tools signatures. By having at our disposal a list containing portfolio specific CASE tools signatures, determining the origin of a Cobol source file boils down to comparing its comment lines with the signatures. Obviously, for an unfamiliar collection of source files the names of the CASE tools are not known upfront, and so are their signatures. Naturally, the code generated via the same CASE tools is expected to have similar, if not identical, comments. To exploit this fact we employed Latent Semantic Indexing (LSI) process to help us disclose similarities among the source file comments. LSI originates from the analysis of semantics of a natural language and has been successfully used to detect natural text similarities [106, 86, 22, 11, 31]. We therefore applied it to texts originating from the source

code comments. By having established similarity relationships among the source files it becomes possible to cluster the files into groups. Each of which is expected to hold files sharing some common characteristic, for example, origin from the same CASE tool. As it turned out application of LSI enabled us to recover the unknown CASE tools signatures. Some of which occurred with low frequency and of which the portfolio maintainers had no idea.

LSI in a nutshell LSI process operates on top of the concept of a document. Documents can be seen as containers with words, which are referred to as terms. The documents collection is modeled as a vector space. Each document is represented by a vector consisting of the numbers of occurrences of the terms. Let us denote with A the term-document-matrix. The columns of the matrix A are formed from the document vectors. A is a sparse matrix of size $m \times n$, where m is the total number of terms over all documents, and n is the number of documents. Each entry $a_{i,j}$ contains the frequency of a term t_i in a document d_j.

The LSI process involves a number of steps. First, the term-document-matrix is obtained and weighted by a weighting function to balance out the occurrences of very rarely and very frequently occurring terms. To carry out this step we used the *tf-idf* approach [29, 125] since it has been shown to perform well in combination with LSI [30, 174]. In *tf-idf* the local and the global importance of the terms is taken into account. The local importance of the terms is expressed with the term frequency (*tf*) metric, and the global importance, with regard to all the documents, is expressed with the inverse document frequency (*idf*) metric. Product of the two metrics provides the weighted frequency of the terms. Mathematical details on the *tf-idf* approach are discussed in [29, 125, 158]. Next, the Singular Value Decomposition method (SVD) is used. Its role is to break down the vector space model into less dimensions yet preserve as much information about the relative distances between the documents as possible. SVD decomposes matrix A into its singular values and its singular vectors. An approximation A_k of A is obtained after truncating it at the k^{th} largest singular value. A_k is referred to as the approximation of A with rank k. Finally, the value of k needs to be determined. The value of k determines the quality with which the documents are grouped together. In [158] the authors discuss several approaches to choosing k. As it turned out for us determining the value of k on the basis of formula 2.1, taken from [97], sufficed to facilitate our needs.

$$k = (m \cdot n)^{0.2}. \tag{2.1}$$

In formula 2.1 m and n are the numbers of rows and columns in the term-by-document matrix, respectively.

The similarity relationship between any two documents is commonly defined as the cosine between the corresponding vectors. If two vectors point in the same direction the corresponding documents are deemed to be similar. Cosine similarity between two documents d_i and d_j is computed on the basis of the A_k matrix according to formula 2.2.

$$cos(i,j) = \frac{e_i^T A_k^T A_k e_j}{||A_k e_i||_2 \, ||A_k e_j||_2}. \tag{2.2}$$

Formula 2.2 is an adaptation of the original formula for computing cosine between two vectors which enables finding similarity between documents on the basis of the matrix A_k. The parameters i and j determine the indexes of the documents for which the cosine similarity is to be computed. The e_i and e_j are the column vectors of the length equal to the number of rows of A with zeros at all positions except for i^{th} and j^{th}, where there are 1s. Such vectors when multiplied by the A_k matrix enable selection of the i^{th} and j^{th} column from A_k.

Preparing the documents To apply LSI to the source code from the studied IT-portfolio we had to prepare the documents. For our purposes the source code comments served as the basis for this task. Given a Cobol program its comment lines, if present, were identified and extracted. To identify the comments we used the same regular expression as the one used in counting comment lines of code, discussed earlier in this section. We did not need to use all the comments from the source files to have input for documents preparation. Our observations of the CASE tools behavior led to formulation of a heuristics that enabled us to limit the number of comment lines. Particularly, we considered those comment lines which were likely to contain the CASE tools characteristics. A manual inspection of a number of Cobol programs revealed that it is sufficient to take at most the first 20 lines of comment lines occurring before the `PROCEDURE DIVISION` statement. Also, to improve the effectiveness of LSI we filtered out from the extracted comments commonly occurring terms [158]. In our case we found plenty of dates and therefore we opted for removal of numbers. All the steps carried out other than providing us with the basis for documents construction also resulted in lowering the computational complexity for LSI operations by diminishing the amount of terms in the documents.

We had at our disposal $8,188$ Cobol programs ($\approx 99.9\%$ of all modules $8,198$) that were considered for documents formation. The 10 modules that were skipped did not contain any comments before the `PROCEDURE DIVISION` statement. As our manual analysis revealed none of the skipped modules appeared to originate from a CASE tool. After extracting and filtering the relevant comments with each Cobol program we associated a string made up of the content of the comments. From each string we created the document vectors and obtained the documents-terms-matrix. We weighted it using the *tf-idf* approach. The documents-terms-matrix we dealt with for the studied portfolio was of the size $12,987 \times 8,188$, where $12,987$ was the total number of unique terms in all obtained documents, and $8,188$ was the total number of documents.

Analysis process The process of finding the CASE tools signatures was carried out in a semi-automated manner. Whereas the LSI process is able to help us cluster the similar documents the recovery of the CASE tools signatures requires an additional effort. We opted for manual analysis of the clusters. The process of finding CASE tool signatures departed from a set containing all the documents and was done iteratively. We first summarize the steps of the process, and then elaborate on the details.

1. The documents from the set are clustered, and the obtained clusters sorted according to the number of elements they contain.

2. The top 10 largest clusters are selected for manual inspection. From each cluster the source files corresponding to the contained documents are obtained. These are then manually reviewed to establish whether the source code of any of the files originates from some CASE tool. If this is the case on the basis of the source code comments of that file the CASE tool signature is formulated through a regular expression. The process of cluster inspection is repeated until all the 10 clusters have been examined.

3. The regular expressions are used to determine names of the CASE tools across all the source files associated with the documents that remain in the set. The source modules for which the CASE tool was recognized are labeled as generated and the name of the tool is assigned to them. The corresponding documents are removed from the analyzed set of documents.

4. The above steps are repeated until examination of all of the top 10 clusters does not lead to recovery of any new CASE tool signature. When this happens we assume that all the source files that correspond to the remaining documents do not originate from any CASE tool, and therefore are hand-written.

To cluster the documents it is essential to have the document similarities available. We obtained the approximation of the documents-terms-matrix with the rank determined on the basis of formula 2.1. By means of formula 2.2 we computed the cosine similarity between each pair of documents, and formed the documents similarity matrix. The obtained matrix was used as the basis for clustering the documents. Hierarchical clustering methods are frequently used for clustering documents [23]. Application of a hierarchical clustering method produces a tree structure expressing the hierarchy of the clusters. In that tree the leaves correspond to the documents. The clusters are obtained by cutting off the tree at a specified level (threshold). One of the parameters in hierarchical clustering is the linkage criterion which is the way of determining the distance between sets of documents. The most common linkage criteria are the minimum (*single-linkage*), average, and maximum (*complete-linkage*) [158]. Apart from the linkage criteria it is also important to determine the tree cut-off threshold used to identify the actual clusters. In [97] Kuhn et al. applied several thresholds for clustering different data sets which originated from source code; these ranged between 0.4 and 0.75. Very often the threshold is chosen empirically. This is the route we chose to parametrize the clustering step. We experimented with varying the size of clusters and inspecting their content. Through a number of experiments we found that using as linkage criteria the average method with the cut-off threshold of 0.85 yields clusters which are of relatively small size and the grouped documents have very similar content. Such outcome turned out to have a desired effect since manual inspection did not require much effort and allowed for completion of the CASE tools signatures recovery in two iterations.

Mathematical manipulations involving large matrices are typically complex and time consuming. In case of LSI it is common to handle matrices with dimensions going into thousands of rows and columns. In our case the situation was not different. We opted for Matlab to conduct all the mathematical operations such as SVD calculation, the cosine

similarity matrix computation, or hierarchical clustering. Matlab turned out to be capable of providing strong support for sparse matrices manipulation and high performance [74].

CASE tools signatures recovery We applied our analysis process to the source code from the studied portfolio. We present a summary of the process in Table 2.2 by listing characteristics of each iteration. For each iteration we provide the total number of documents that entered the clustering phase (*Documents*), the total number of clusters obtained (*Clusters*), minimal and maximal sizes of the obtained clusters, and a vector listing sizes of the top 10 largest clusters.

Property	Iteration 1	Iteration 2
Documents	8,188	3,551
Clusters	2,933	1,440
Min. cluster	1	1
Max. cluster	241	15
Top 10 clusters	(241 87 42 36 31 29 22 21 20 19)	(15 13 10 9 8 7 7 7 7 7)

Table 2.2: Summary of the CASE tools signatures recovery process.

In the first iteration we clustered the entire set of documents. We examined the content of the top 10 largest clusters. In the first and third clusters we found documents associated with source files which were all generated by TELON. All of these files also contained an additional comment pointing us at a tool called 'Report Cobol conversion services'. This comment was also occurring in the source files mapped to the documents from the clusters: second, fourth, sixth, and tenth. Although it appeared to have originated from some tool we were unable to map it with any commercially available CASE tool known to us. As we learned from the portfolio experts this comment was generated by a home-grown conversion tool which was designed for the purpose of an internal project carried out within the organization. Given this fact we disregarded occurrences of these comments in our analysis. In the fifth cluster we found documents linked to COOL:Gen files and in the seventh Advantage Gen. In the eighth and ninth clusters we found documents associated with the TELON generated files which were free of the 'Report Cobol conversion services' comment. During the first iteration we formulated regular expressions capable of matching the CASE tool comment signatures of COOL:Gen, Advantage Gen and TELON.

In the second iteration we again examined the content of the top 10 largest clusters. In this iteration what became apparent was the high reduction in the sizes of the clusters and also a significantly lower number of documents subjected to clustering. In the largest cluster we found documents mapped to source files in which comments pointed again at the 'Report Cobol conversion services'. In the fourth cluster we found documents linked with comments generated by CANAM Report Composer. In the seventh cluster we found code generated by KEY:CONSTRUCT FOR AS/400. In the source files associated with the documents originating from the remaining clusters we did not find any signs suggesting that the code was generated and presumed the sources were hand-written. During the second iteration we formulated regular expressions for matching comments generated by CANAM Report Composer and KEY:CONSTRUCT FOR AS/400 tools.

Interestingly, the signature of the KEY:CONSTRUCT FOR AS/400 was not embedded inside comments but was part of the Cobol construct `SECURITY`. As explained earlier this construct is one of the *documentation statements* in Cobol and is meant to serve as a label for a source code comment. Let us recall that when preparing the documents for the LSI analysis we ignored the non-comment lines from the sources. The LSI must have grouped the sources together due to the consistently occurring comment lines containing the PROGRAM-NAME and TIME-WRITTEN terms throughout all the KEY:CONSTRUCT FOR AS/400 generated files.

```
Advantage Gen
^.{6}\*\s{21}(Advantage)\(tm\)\sGen\s[\d\.]+\s*

CANAM Report Composer
^.{6}\*\s{2}Source\scode\sgenerated\sby\sCANAM\sReport\sComposer\sV[\d\.]+

COOL:Gen
^.{6}\*\s{27}(COOL\:Gen)\s*

TELON
^.{6}\*TELON\-{60}

KEY:CONSTRUCT FOR AS/400
^.{6}\s+GENERATED\sBY\sKEY:CONSTRUCT\sFOR\sAS\/400
```

Figure 2.10: Regular expressions corresponding to the recovered CASE tools signatures.

Figure 2.10 presents the regular expressions which correspond to the CASE tools signatures recovered during the analysis. Given a module m the value of the metadata $GEN(m)$ was determined by matching each of the regular expressions against the comment lines occurring in the module. Once some line was matched the value of the $GEN(m)$ was set to the code generator name that corresponds to the used regular expression. If none of the regular expressions matched the value of $GEN(m)$ was set to $-$. Modules with the GEN value of $-$ are assumed to be hand-written.

2.4 Screening the data

We applied the presented analysis facility to the sources from the studied portfolio. As a result we obtained a codebase. Prior to using the extracted data for portfolio analyses it is indispensable to check it. In this section we discuss data normalization, source code coverage, and data plausibility checks we carried out with regard to the obtained codebase. First, we embark on normalization of the data. This process turned out to be inevitable for the DC, and CHG metadata given the multitude of formats in which names and dates were recorded in the source code comments. Next, we discuss the source code coverage. Due to the nature of the analyzed source code elements the availability of certain data was not guaranteed. An obvious example is the metadata derivable from comments. We express coverage as a percentage of the total source modules and provide the percentages for the individual metrics and metadata. Finally, we present the plausibility checks we

carried out on the extracted data.

2.4.1 Data normalization

Some of the metadata that we extracted from the source code was most likely never meant to be processed by means of automated tools. The dates we obtained from the comments did not follow a unique formatting. Also, we found numerous synonyms for even the simplest notions, e.g. the month names. Obviously, from the perspective of manual data analysis these synonyms and format inconsistencies are sometimes negligible. However, when dealing with huge amounts of data the use of statistical software tools is a necessity. And, in that case there must exist some agreed standards to which the analyzed data adhere to. To enable processing of the comments extracted data some metadata had to undergo normalization. In particular, this process was applied to the CHG and DC metadata.

How people write dates In textual representation of the dates not only formats differ widely, but also the semantics may differ. For instance, 02/03/04 means March 4, 2002 in Japan, February 3, 2004 in the USA, and otherwise it means March 2, 2004. So, in large IT-portfolios it is already daunting to interpret something as seemingly simple as a date field from comments. Of course, when there is more contextual knowledge there are ways to recover semantics of a date. For instance, 11-03 2005 as a history date and 09-30 2005 as another history date in the same history log, implies that 11-03 is not 11th of March. The following are a few examples of the comments embedded date fields that we encountered in the studied portfolio:

```
26-10-94        FEBR 1971          09/06/2000       OCTOBER/NOVEMBER 2000
2.10.94         AUTUMN-1976        20030527         DEC 28,1972
09-81           OCT                13 08 1990
09.80           32-01-2001         07 31, 1971
```

The multitude of date formats makes manipulations involving a group of dates, e.g. aggregation on per month basis, impossible. Before any comments extracted date was used as a value for either the DC or CHG metadata we normalized it to the form of DD-MM-YYYY, where DD represents a day number in a 2 digit form, MM holds the month number in a 2 digit form, and YYYY represents the year number in a 4 digit form. From the examples listed above it is clearly visible that in the process of date normalization we had to deal with issues such as synonyms resolution, missing values specification, and validation. Synonyms typically occurred for the month names. They were also part of certain types of date fields which rather than providing the actual dates suggested time frames. For example, the strings FEBR 1971, AUTUMN-1976, or OCTOBER/NOVEMBER 2000 represent such cases. We strove to normalize as many raw date fields as possible and for cases similar to FEBR 1971 and OCTOBER/NOVEMBER 2000 we defined rules that enabled us to obtain the actual dates by filling in the missing values. All dates containing a month and a year were assigned day value of 1. All occurrences containing a sequence of two consecutive months followed by a year were assigned the day value of 1 and a month value of the second month. Of course, each obtained date had to be validated. By doing

so we were able to discard the obviously incorrect date entries, for example, like the one listed above: 32-01-2001.

To handle the undesired occurrences of the natural language words we prepared a number of string-to-integer mappings. When designing those we took into account the various ways of expressing month names. For instance, for the month October we created mappings involving words: OCTOBER, its English contraction OCT, and also its Dutch equivalent OKT. The words used in the mappings were taken from the cases we encountered during manual inspection of a sample of the module sources. Apart from handling the month synonyms we also had to tackle the synonyms used to denote years. If provided, the year component of a date was written as either 2 or 4 digit integer. To be able to represent year as a 4 digit number we did append either a 19 or 20 prefix to any 2 digit year representation. To determine the correct prefix we followed the assumption that the years recorded in the portfolio source code comments must fall within a certain range. The lower bound of that range was determined by the year 1960 which corresponds to the early days of the Cobol programming language. The upper bound of the range was set to 2007 which is the first year following the latest compilation year recorded across all the modules in the studied portfolio.

To convert the raw date fields to dates adhering to the DD-MM-YYYY format we designed a number of regular expressions capable of parsing the extracted dates. With each regular expression we associated a semantic pattern which provided a means for determining values for the day, month and year components of a date. For instance, the regular expression

```
^(\d{2})[\.\-\/]?(\d{2,4})\.?$
```

matches raw date fields such as 09-81., 09.80., 09/88, or 0600. With this expression we associated a semantic pattern which assigns the day component the value of 1, the month value is derived from the content captured by the first set of parenthesis `()` in the regular expression and the year value is derived from the content captured in the second set of parenthesis.

Now let us explain the normalization process. Given a raw date field at hand we proceeded with it as follows. First, we determined which of the defined regular expressions match. If none of the regular expressions matched the date was discarded. Otherwise, interpretation of the identified date components took place. At this point all the non-numeric components of the date were resolved by means of the defined mappings. Next, the semantic patterns were used to assign values to the date components. Finally, the obtained date was validated. This basically meant checking if the date was a valid calendar date. Any invalid date was discarded.

2.4.2 Portfolio coverage

In the definition of the codebase we distinguished two types of source code facts: metrics and metadata. In case of metrics the source code analysis resulted in a complete coverage of all the portfolio modules. Namely, for each module in the portfolio each metric is

assigned a value. This fact follows directly from the definitions of the metrics and the mechanisms used for their computation.

In case of metadata the situation is different. Given a Cobol module the availability of values for the related metadata was dependent on the presence of the underlying data in the source code. For instance, the comments which were subject to our analyses are an optional element of the Cobol sources. So, in case they are absent the related metadata is also missing.

Metadata	Modules	(%)
CHG	4,393	53.59%
CN	8,198	100.00%
DC	2,580	31.47%
GEN	8,198	100.00%
LC	8,198	100.00%
N	8,198	100.00%

Table 2.3: Cobol modules coverage by the metadata.

In Table 2.3 we summarize the Cobol modules coverage by the metadata. In the first column we provide the metadata. In columns two and three we provide the numbers of modules covered along with percentages. Values for the N, LC, CN, and GEN metadata are available for all of the modules. For these metadata either the source code elements from which the values were derived were always present (N, LC, CN) or the rules used to determine the values allowed setting them for all the modules (GEN). The metadata availability is not complete in case of the DC and CHG metadata. Let us recall that values for the DC metadata were obtained from the optional Cobol language construct `DATE-WRITTEN`. During the code analysis by means of our tools we determined the presence of the `DATE-WRITTEN` constructs for 2,580 ($> 31\%$) modules. This figure represents the total number of modules for which the values of the DC metadata was defined.

In case of the CHG metadata the availability of values was dependent on the presence of a module change history recorded in the comments. For any given module in order to have CHG defined the following conditions had to be satisfied: comments are present before the `DATA DIVISION` statement, the comments contain the relevant data, and this data is extractable using our mechanisms. Of course, in the studied portfolio we had no guarantee that for each module all of these conditions were met. The total amount of modules with comments present before the `DATA DIVISION` amounted to 8,179 (99.76%). The number of the commented lines in the analyzed portion of the files averaged to 33 with 1 as minimum and 314 as maximum. From Table 2.3 we see that for over a half of the modules in the portfolio we managed to obtain some characteristics with respect to the history of module changes.

2.4.3 Plausibility checks

With the extracted data we want to recover managerial information. The correctness of the values we extracted from the code cannot be easily verified, but obvious invalidities are

recognizable. The plausibility of data is a concern that holds for all the data comprising the codebase. In our case especially prone for errors are values derived from the source code comments. We now present the plausibility checks we carried out for the obtained codebase.

Metrics For metrics we evaluated the likelihood of the obtained values. The range of values for each metric was made up of non-negative integers which was in line with the expectations given the definitions of the metrics. We checked the metrics by analyzing summaries of the distribution of the observations. To do this we calculated the so-called five-number summary for each metric which consists of the minimum, the maximum, the median, the first and the third quartiles.

Metric	LOC	$SLOC$	$CLOC$	$BLOC$	MC
Min	21	17	0	0	1
1st Q	679	475	164	0	22
Median	1,442	1,045	341	8	58
3rd Q	2,859	2,031	752	43	138
Max	52,174	37,948	14,226	1,627	4,628

Table 2.4: The five-number summaries for the metrics: LOC, $SLOC$, $CLOC$, $BLOC$, and MC.

Metric	OBS_x														
x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Min	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1st Q	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Median	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3rd Q	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
Max	0	1	2	1	1	1	0	0	0	24	2	4	10	2	21

Table 2.5: The five-number summaries for the OBS_x metrics.

In Tables 2.4 and 2.5 we present the calculated five-number summaries for each metric from the codebase. The smallest Cobol program in the portfolio appears to have as little as 21 LOCs. Cobol programs of such relatively low length occurring in an industrial portfolio draw attention, and therefore we reached for the source files characterized by this length. We found one Cobol program. It had 17 source lines of code, 1 comment line and 3 blank lines. We scrutinized its code and concluded that it is a valid Cobol program. On the other extreme we found a Cobol module with 52,174 LOCs. Of course, a program with this many lines of code is not improbable but it is certainly worth attention. We found one file in the portfolio with such length. It was a valid Cobol program in which source lines of code amounted to 37,948 ($\approx 72.7\%$ of its LOC). The remainder constituted comments (14,226). As it turned out the file appeared to be an outlier. The fifth largest program in the portfolio had already less than half the LOC of the largest. The number of its source lines of code constituted $\approx 60\%$ of the LOC. To gain more insight into the lines of code metrics we also checked their mutual relationships. Given the nature in which programs are written it is intuitive to expect that for any given program

m the following inequality $BLOC(m) < SLOC(m)$ is satisfied. We found that it indeed held true for all the modules in the portfolio. In case of another intuitive relationship, $CLOC(m) < SLOC(m)$ for any module m, we found 67 programs ($< 1\%$ of all the portfolio modules) where this inequality was violated. We inspected manually several of the spotted files and found them to be heavily commented. Of course, the violation does not mean the lines of code counts are invalid. All the observations revealed through the quick checks suggest that the lines of code counts we obtained were valid.

The values for the MC metric ranged between 1 and 4,628. The upper bound appeared alarming. Following the semantics of the MC metrics the value of 4,628 suggests existence of a Cobol program which implements as many as 4,628 of linearly independent paths through a program's code. This translates into a very complicated logic structure. To explain the presence of any high values of the MC metric we used the observation that counts of the program's branching points are related to the number of lines of code. This relation follows from the fact that when calculating the MC metric the occurrences of particular Cobol constructs are counted. These constructs occupy the physical lines of code. So the more they occur the more likely it is that the number of lines of code is higher. We took data for the LOC and MC metrics from the third quartile and computed Pearson's correlation coefficient. As it turned out the two vectors showed strong correlation of 0.7358703. We found this argument convincing to accept the obtained MC values as reasonable.

The values of the family of OBS_x metrics represent counts of particular Cobol statements. We checked basic properties that such counts should have. The metrics OBS_2 through OBS_6 represent counts of the optional Cobol keywords `AUTHOR`, `INSTALLATION`, `DATE-WRITTEN`, `DATE-COMPILED`, and `SECURITY`. These keywords occur in the `IDENTIFICATION DIVISION` of a program [103]. From practice we know that at most one occurrence of any of these keywords happens. From Table 2.4 we see that in all cases except for OBS_3 the range of counts is between 0 and 1. We examined closer the exception and it turned out that in the portfolio we found 1 Cobol module in which the `IDENTIFICATION DIVISION` contained 2 instances of the `INSTALLATION` keyword. This clearly explained the distinctive maximum value. The remaining constructs corresponding to the metrics denoted as OBS_1 and OBS_7 through OBS_{15} have a potential to occur many times inside the code, or not at all. For this reason the minimum value of 0 appeared valid. As far as the upper bound, we assumed that given the fact these keywords are known to be obsolete their occurrences should not be frequent. In fact, zeros for the first and third quartiles and the median suggest that not many modules contain the counted constructs. We therefore concluded that the relatively low maximum values appeared to be reasonable.

Dates To inspect the plausibility of the comments extracted dates we analyzed their relative positions with respect to the corresponding LC dates. Let us explain. In either the development or maintenance processes of a source module one typically first creates (or alters) the module's code, possibly reports this activity in the comments, and then compiles the module. It is fair to assume that all the dates reported in the comments should fall before the module's latest compilation date. For each module m we extracted the date of its last compilation, $CN(m)$. This date was recorded automatically by the

compiler in the compiler listing at the time of the module's compilation. We therefore assume its validity and treated it as a reference point in the module's development history. We used the CN dates to verify the plausibility of the extracted DC and CHG dates.

We analyzed the dates as follows. First, we determined the set of modules used for the analysis. We considered all the modules for which either the DC or the CHG metadata had assigned dates. In this way we obtained 5,177 Cobol programs. Next, for each selected module we picked the latest from the available dates. Finally, we computed the difference between the modules' last compilation date and last modification date. This way we obtained a vector of 5,177 integers representing the difference between the dates measured in the number of days.

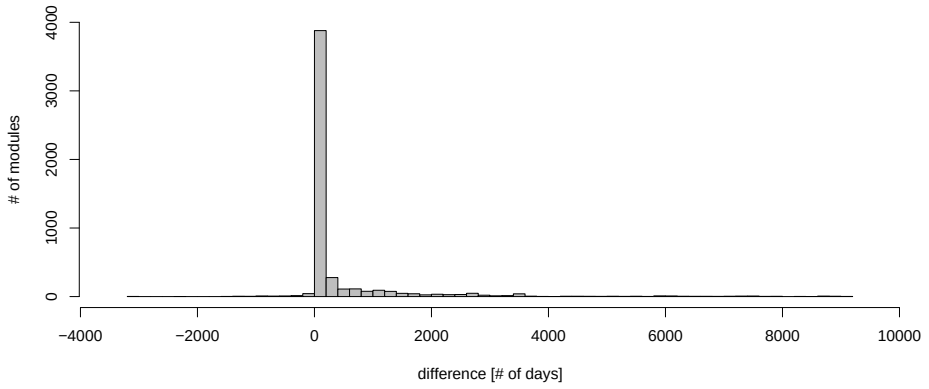

Figure 2.11: Distribution of the distance (measured in days) between the modules' last compilation and last modification dates.

Figure 2.11 presents the histogram of the 5,177 computed differences. The number of days is presented on the horizontal axis. One bar is distinctive in the histogram. It represents the bin which covers values ranging from 0 (inclusive) through 200 (exclusive). It corresponds to those modules which were compiled at most 200 days after their last modification. In the studied portfolio we found 3,879 of such modules. This represents nearly 75% of the modules for which the version history data was recovered from the source code.

All the bars to the right of the tallest one correspond to the modules for which the difference is at least 200 days. Although such distances between code changes and compilations may appear odd from the software process perspective they are possible. We found 1,202 modules with this property. We selected several modules corresponding to the differences represented by the right most bins in the histogram, and reviewed their source code. All of the modules contained the `DATE-WRITTEN` construct and no history records in the comments. All the DC values pointed at the late 70s. We did not spot any discrepancies between the extracted dates and the dates contained in the reviewed source code. Based on these observations we classified all the dates with a difference of at least

200 days as valid.

To the left of the tallest bar we find several significantly shorter bars. These represent possibly erroneous dates stored in the DC and CHG metadata. We deduced this from the observed negative differences between the compilation and modification dates. The total number of modules with such dates was 96, what represents less than 2% of the 5,177 modules. We discarded those DC and CHG dates from our further analyses.

Context relevance In the constructed codebase we stored data describing the software tools involved in the development and maintenance of the portfolio. Particularly, for every module m we retrieved the name of the compiler last used for its compilation ($CN(m)$), and the name of the CASE tool used for generation of its source code ($GEN(m)$). To evaluate the plausibility of the values extracted for the CN and GEN metadata we checked whether the detected compilers and code generators are relevant for the Cobol mainframe environment.

The values we obtained for the CN metadata provide names of four different IBM compilers. They are as follows: IBM OS/VS COBOL, COBOL for OS/390 & VM, IBM COBOL FOR MVS & VM, and Enterprise COBOL for z/OS. All of them are valid names of the Cobol compilers used in the mainframe environments.

We obtained six different values for the GEN metadata. Five of those were names of the CASE tools. The remaining one was a label used to mark the hand-written Cobol programs. We found the following code generators: TELON, COOL:Gen, Advantage Gen, CANAM Report Composer, and KEY:CONSTRUCT. TELON is one of the first CASE tools released in 1981 that supported Cobol code generation [16]. COOL:Gen and Advantage Gen are another CASE tools which among other programming languages generate Cobol code. Advantage Gen is in principle a synonym for COOL:Gen. The name got introduced after Computer Associates acquired the tool from Sterling Software [34]. CANAM Report Composer is a code generator specific to writing programs that generate reports [138]. It supports data retrieval and its presentation. KEY:CONSTRUCT is part of the KEY:Enterprise suite [141]. Since all the detected CASE tools are valid Cobol code generators we assumed the values of the GEN metadata as valid.

2.5 Information recovery

In IT-portfolio management aspects such as the market value of the IT-assets, operational costs, or their expectations for future, just to name a few, constitute essential information for IT-executives. In this section we focus on management information and present how we obtained it with the support of data stored in the codebase. First, we embark on the topic of measuring the size of IT-assets and introduce function points. Next, we show a number of benchmark formulas which are based on function points and useful to capture managerial characteristics of an IT-portfolio. Finally, we present how to estimate the average annual growth rate of the portfolio. We illustrate it by showing calculations for the studied portfolio.

2.5.1 Function Points

Software size is important for IT-executives since it underpins many key decisions such as effort, cost estimation or scheduling, to name a few. To measure information systems size we use function points (FPs) [6, 44, 28, 88]. A function point is a synthetic measure expressing the amount of functionality of an information system. Function points have proven to be a widely accepted metric both by practitioners and academic researchers [90, 84]. For executives it is important to know how reliable these metrics are. In [45, p. xvii, 28] we can read:

> As this book points out, almost all of the international software benchmark studies published over the past 10 years utilize function point analysis. The pragmatic reason for this is that among all known software metrics, only function points can actually measure economic productivity or measure defect volumes in software requirements, design, and user documentation as well as measuring coding defects. [..] Function points are an effective, and the best available, unit-of-work measure. They meet the acceptable criteria of being a normalized metric that can be used consistently and with an acceptable degree of accuracy.

Function point counting is obtained through manual analysis of the functional documentation of an IT project and the analysis is conducted by certified function point counters that adhere to widely recognized standards for function point counting [72]. The metric yields exchangeable results but the costs for obtaining it can be high. For this reason some have proposed alternative methods to consider. For instance, the FP-lite method presented in [5, 26]. In this study we employ yet another technique for deriving function points called backfiring.

Backfiring allows approximation of FPs on the basis of source code by taking the SLOC count and multiplying it by a static factor which is particular for a given programming language. A list of those factors is available, for instance, in [80]. It is claimed that the method yields results with the accuracy of approximately $+/-20\%$ [82, p. 79]. It is usually the only technique for deriving function point counts for information systems when functional documentation is not available, or insufficient to apply function point analysis, and therefore commonly used for sizing legacy systems. It is by far the cheapest method for obtaining function point estimates which does not rely on the software's functional documentation. The method is considered suitable for *rough-and-ready* calculations [24]. And, it is especially applicable when a large sample is concerned as the law of large numbers will level the discrepancies between types of applications on a portfolio-wide basis. In this study we deal with a large portfolio, and for our purposes getting the size indication rather than the accurate measure is sufficient. Besides, source code is our primary data.

Size approximation We estimate function point counts on per system basis. To be able to use backfiring it is essential to have the SLOC counts available for the source

code which implements the systems in question. The process of size approximation for a system S is accomplished according to the following formula.

$$f(S) = \frac{1}{107} \sum_{m \in S} SLOC'(m). \tag{2.3}$$

In formula 2.3 $f(S)$ denotes the backfired function point count for a given system S. We assume that S is a set of source files which form implementation of the system S. The constant 107 is particular for Cobol programming language and was obtained from [80]. The $SLOC'$ provides for an extension to the $SLOC$ metric. In addition, to the total number of source lines of code for module m it also embraces the source lines of code of the embedded copybooks. Since copybooks constitute in part the content of the Cobol sources we considered inclusion of their lines of code measures when deriving the systems' function points.

2.5.2 IT benchmarks

Function points alone do not suffice to derive managerial indicators relating to IT-portfolio and therefore additional support is needed through benchmark information. For the purpose of our analyses we utilized publicly available benchmarks since organization specific benchmarks were for internal usage only. Although with the public benchmarks the exact numbers differ the conclusions stay the same. Let us now discuss the indicators based on function points which we will use in our analyses.

Development time Useful in our explanations is a fundamental relationship between function point count and development time of an information system, estimated from Capers Jones' public benchmarks [78, p. 202].

$$d(f) = f^{0.39}. \tag{2.4}$$

In Formula 2.4, f represents the number of function points of an IT-system and d stands for development time (measured in months). The exponent in the formula is specific for MIS type of software which is part of all businesses and omnipresent in the financial services industry.

Assignment scope In addition to schedule power benchmark there are two other benchmarks related to assignment scope, also taken from [78, p. 202-203]. These are overall benchmarks, not specific for the MIS industry.

$$n_d(f) = \frac{f}{150}. \tag{2.5}$$

$$n_m(f) = \frac{f}{750}. \tag{2.6}$$

Formula 2.5 is used to estimate the size of the software development team. Formula 2.6 provides a way to estimate the number of staff needed for keeping an application up and running (operational) after delivery. In both cases the obtained numbers represent full time equivalent (FTE) staff. In both formulas f stands for the number of function points.

Cost of development To be able to estimate the total cost of development, abbreviated tcd, the following assumptions have been made. First, we assume the assignment scope of a system developer. The assignment is obtained from formula 2.5. We further assume a daily burdened rate of r for development and w for the number of working days per year. Finally, we need to take into account duration of the development. This is estimated from formula 2.4. The total development cost can then be calculated as follows.

$$tcd(f, r, w) = \frac{f}{150} \cdot r \cdot w \cdot \frac{f^{0.39}}{12}. \tag{2.7}$$

In formula 2.7 the first factor in the product is the expansion of the n_d formula. Then, we have the r and w parameters. And, the last factor is the right side of the d formula divided by 12 in order to convert months into years.

Cost of operation Another interesting indicator is the yearly cost of keeping a system operational during its life-time. We estimate this with the use of staff assignment scope formula 2.6. We express the yearly cost of operation, denoted by yco, as follows.

$$yco(f, r, w) = \frac{f}{750} \cdot r \cdot w. \tag{2.8}$$

As we see in formula 2.8 apart from f there are also two additional parameters: r and w. They stand for a daily burdened rate for maintenance and the number of working days per year, respectively.

Risk of failure Benchmark data indicates a very strong correlation between the size of software and the chance of an IT-project failure [167]. Based on the public benchmark data from [82, p. 192] the following formulas for chance of IT-project failure were estimated in [167].

$$cf_i(f) = 0.4805538 \cdot (1 - exp(-0.007488905 \cdot f^{0.587375})). \tag{2.9}$$

$$cf_o(f) = 0.3300779 \cdot (1 - exp(-0.003296665 \cdot f^{0.6784296})). \tag{2.10}$$

In formula 2.9, cf_i stands for the chance of failure in case of in-house developed projects and f for the number of function points. In formula 2.10, cf_o stands for the chance of failure for projects that are outsourced. The formulas are suited to be used for

projects which size is less than 100,000 function points. The formulas 2.9 and 2.10 have asymptotic behavior which prevents them from reaching one as the size of IT-project goes to infinity. For our analysis the formulas suffice as the size of each IT-asset in our portfolio falls within the covered range.

Portfolio indicators All the formulas we presented so far are applicable to individual systems. When carrying out a managerial assessment of an IT-portfolio we are interested in deriving characteristics particular not just to one system but a group of systems. Being able to assign the managerial indicators to groups of systems gives the possibility to assess certain properties which apply to, for instance, systems belonging to a particular business unit. It also provides means to compare business units, or carry out other operations. Let us now present how we use the managerial indicator formulas in the context of an IT-portfolio.

In order to aggregate the rebuild cost for an arbitrary group of systems we first estimate the total development cost for each system from the group individually, and then add up the obtained results. Let us recall that the tcd formula requires two extra parameters: r and w which we need to specify. For simplicity we assume they are constant for all systems. Let A denotes a set of information systems. The rebuild cost formula 2.11 for a portfolio A is expressed as follows.

$$P_{rc}(A, r, w) = \sum_{S \in A} tcd(f(S), r, w). \tag{2.11}$$

In order to estimate the risk of failure of rebuilding a group of systems we use formula 2.12 proposed in [167, p. 49].

$$P_{rf_y}(A) = \frac{1}{|A|} \cdot \sum_{S \in A} cf_y(f(S)). \tag{2.12}$$

The yearly operational cost of keeping a collection of systems up and running are estimated on the basis of formula 2.13. Here, again we need two extra parameters: r and w. For simplicity we assume they are constant for all the systems. The yearly operational cost formula for a portfolio is as follows.

$$P_{yco}(A, r, w) = \sum_{S \in A} yco(f(S), r, w). \tag{2.13}$$

In formulas 2.11, 2.12 and 2.13 the function f approximates the size of the given information system S, according to formula 2.3. The function cf_y represents either a formula for estimating the chance of failure in case of in-house developed projects (formula 2.9) or the one used for outsourced developments (formula 2.10). The use of either of the formulas is indicated through the y subscript.

Estimation of the duration of rebuilding a group of systems is done as follows. Let us assume that we start rebuilding all the systems at one moment in time. The rebuild process

of a group of systems is considered finished the moment all of the systems are complete. We also assume that each rebuild task is successfully accomplished. We estimate the duration of rebuilding a group of systems according to formula 2.14.

$$P_d(A) = \max_{S \in A}\{d(f(S))\}. \tag{2.14}$$

To estimate the size of the development team needed to accomplish the rebuild of a group of systems we follow the earlier assumptions and calculate it according to formula 2.15.

$$P_{n_d}(A) = \sum_{S \in A} n_d(f(S)). \tag{2.15}$$

The formula for estimating the number of staff needed to maintain a given group of systems is done as follows.

$$P_{n_m}(A) = \sum_{S \in A} n_m(f(S)). \tag{2.16}$$

Again, in formulas 2.14, 2.15 and 2.16 A denotes the set of systems for which the estimate is done. The identifiers d, n_d, and n_m denote formulas 2.4, 2.5 and 2.6, respectively.

2.5.3 Portfolio dynamics

Software portfolios evolve. One reason for this evolution is volatility of the business that poses new demands on IT. As a result new systems are added, existing ones modified, replaced or removed. This dynamics commonly brings code alterations which typically result in changes to the size of IT-portfolios. As the real-world cases show IT-portfolios grow in size [112, 136]. Given the strong correlation between the software size and the managerial indicators the information on how fast the portfolio's source code base grows enables making projections on the potential consequences for IT-portfolio management. For instance, the expected yearly costs of keeping it operational. To enable such projections it is essential to quantify the phenomenon of portfolio growth. When no information on the portfolio's size growth is available we propose to utilize data derived from its source code.

Amassing source code We used the codebase to obtain a view on the historical dynamics of the portfolio expansion. For this purpose we considered all the modules with the defined DC date. Let us recall, the dates were available for a subset of 2,580 (31.5% of 8,198) modules, and covered the period from 1967 until 2005. For any given module m we interpreted its $DC(m)$ date as the point in time when it was incepted into the portfolio. We grouped the DC dates according to their year component and for each group we counted the number of items. This way we obtained a sequence of integers indexed with the year numbers. Based on this sequence we created a sequence of cumulative sums.

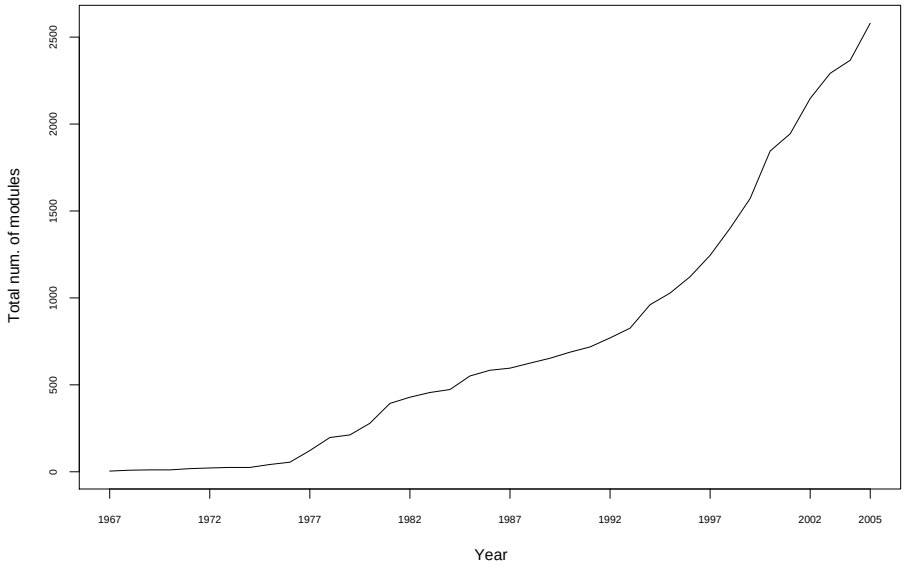

Figure 2.12: Reconstruction of the process of amassing source code in the portfolio on the basis of modules with the defined DC dates.

Figure 2.12 presents the plot obtained on the basis of the constructed vector. The horizontal axis represents the time frame for which the DC dates were available. The vertical axis represents the cumulative number of modules in any given year. It is not surprising that along with the increasing year values the total number of modules also increases. One possible interpretation of the obtained plot is that it presents the process of amassing source code in the portfolio over time. Let us note that other than code addition the plot also reflects possible removals of the modules. Those modules which were removed from the portfolio are gone and so we could not extract any DC dates for them. Apart from that we have no other information on the removals. Nevertheless, the data we used for the construction of the view serves as basis for the study of the portfolio growth rate.

Portfolio growth To obtain an indication of the portfolio growth rate we used the codebase data. Particularly, we calculated a compound annual growth rate of the portfolio over a fixed time frame. Experts agree that recent code changes are the most relevant for projecting future changes [47]. Table 2.6 provides the characteristics of the changes in the number of Cobol modules in the last 10 years. The counts were derived on the basis of the available DC dates.

Let us explain Table 2.6. In the first column we list consecutive numbers which correspond to the years for which the measurements were taken. In the second column we

i	Delta	Total modules	Growth ratio (r_i)
1	0	1,122	1.0000000
2	123	1,245	1.1096257
3	156	1,401	1.1253012
4	171	1,572	1.1220557
5	273	1,845	1.1736641
6	99	1,944	1.0536585
7	203	2,147	1.1044239
8	145	2,292	1.0675361
9	75	2,367	1.0327225
10	213	2,580	1.0899873

Table 2.6: Yearly changes in the amount of source code calculated on the basis of modules with the available DC dates.

provide the totals for the number of modules added to the portfolio between year i and $i-1$. In column three we list the cumulative numbers of modules for any given year. The last column provides the ratios (r_i) between the number of modules in year i and $i-1$. For year number 1 the delta is 0, indicating no module addition. Also, we set the ratio r_1 to 1 to indicate no change in the number of modules. The sequence of r_i ratios serves as input for calculating the compound annual growth rate. The rate calculation is based on the geometric mean. The compound annual growth rate is expressed by formula 2.17.

$$cagr = \left(\prod_{i=1}^{n} r_i \right)^{\frac{1}{n}} - 1. \tag{2.17}$$

In formula 2.17 n determines the number of data points, the ratios (r_i), used for the calculation of the rate. In our case we chose n to be 10. So that we have an idea of the average growth rate of the last ten years. This is presumably an accurate indicator for the coming years. For the data in Table 2.6 the $cagr$ amounts to approximately 8.7%. This percentage gives us the compound annual growth rate which expresses the number of modules added to the portfolio on yearly basis. From the perspective of our analyses it is desired to be able to estimate the number of lines of code by which the portfolio expands. Lines of code allow us to derive function points which serve as input to all the management indicator formulas we presented earlier. To convert the module numbers to lines of code we used the median for the $SLOC$ metric.

To get an idea of how realistic the obtained $cagr$ figure is we compared it with the available public benchmarks referring to other portfolios. It is claimed that each year Fortune 100 companies add 10% of code through enhancements and updates [112]. In [136] Sneed reports on an evolution of a large banking application system for securities processing. The author provides three measurements of the system's size taken every second year and covering a five-year period in which the system was developed. We used the data, filled in the missing values, and applied formula 2.17 which yielded the $cagr$ of approximately 15.1%. As Sneed reports in the last two years the system grew by approximately

10% per annum. We considered the data from Table 2.6 and computed the $cagr$ for the last five years ($i = 6 \ldots 10$). For the studied portfolio we obtained the $cagr$ of approximately 6.9%. Whereas all the figures reported here differ they are in the same order of magnitude.

2.6 Managerial insights

In this section we present how to exploit the codebase potential to address IT-management needs. We use the studied portfolio to illustrate the process of obtaining managerial insights. We employ in this task the formulas presented so far, knowledge on the IT-market, best practices in software engineering, constitutional knowledge about the corporate IT environment, and feedback from experts. We begin with the recovery of the portfolio structure and present its rudimentary characteristics. Next, we propose a way to assess the IT-portfolio in terms of its market value, the cost of keeping it operational, the number of staff needed, and other managerial indicators. Then, we embark on the recovery of risks and costs which are inherently entrenched in the portfolio and driven by the IT-market. We show what vendor lock-in issues we encountered. We discuss how obtaining portfolio-wide insight into the specifics of the development environment provides information useful in planning changes for lowering risks and cutting costs. Moreover, we study certain aspects of maintainability. For instance, by studying data on the last compilation of the portfolio sources we revealed facts which suggest difficulties with future maintenance. Finally, we investigate the quality of the portfolio source code. We show how to obtain an insight into the frequently altered areas in the portfolio, which we refer to as hot-spots, and use it to steer code quality improvement initiatives.

2.6.1 Size structure

Given that software size underpins many managerial indicators we began our analysis of the IT-portfolio with recovery of its size structure. In our view the size structure of an IT-portfolio embodies a number of properties. Particularly, its partitioning into information systems, their size, and distribution of their size. We now present how we recovered these properties from the codebase.

Portfolio organization From the source code perspective identification of information systems boils down to grouping of the source modules which implement the individual systems into sets. IT-systems typically represent logical units that maintain a certain functionality within a portfolio. This fact is commonly reflected in the way the source code is organized. In our case study source code organization into systems was linked with the file-naming convention. Namely, the first two characters in the filename of a Cobol module served as an identifier of the system it belongs to.

We utilized the metadata $N(m)$, name of a Cobol module m, from the codebase to carry out the partitioning. For each module m we considered the first two characters of its name to obtain the identifier of the system it belongs to. This way we partitioned the

entire set of modules into 47 disjoint sets. Each set of files was tagged with a letter S indexed by a number ranging from 0 to 46, as follows

$$S_0, \ldots, S_{46}$$

to represent each information system in the portfolio. The obtained partitioning was in line with the organization's view on the information systems. We were able to locate the two-character codes of the systems in the IT-inventory list that the portfolio experts provided to us. For each code we found information on the function it performs and its production environment. All systems we identified were listed as mainframe applications.

Size distribution To get an idea of the systems' size we applied formula 2.3. We used as input the sets S_0 through S_{46}, and as a result obtained a vector of 47 integers. The approximated function point counts ranged from 14 up to 23,843. Figure 2.13 presents more detailed characteristics.

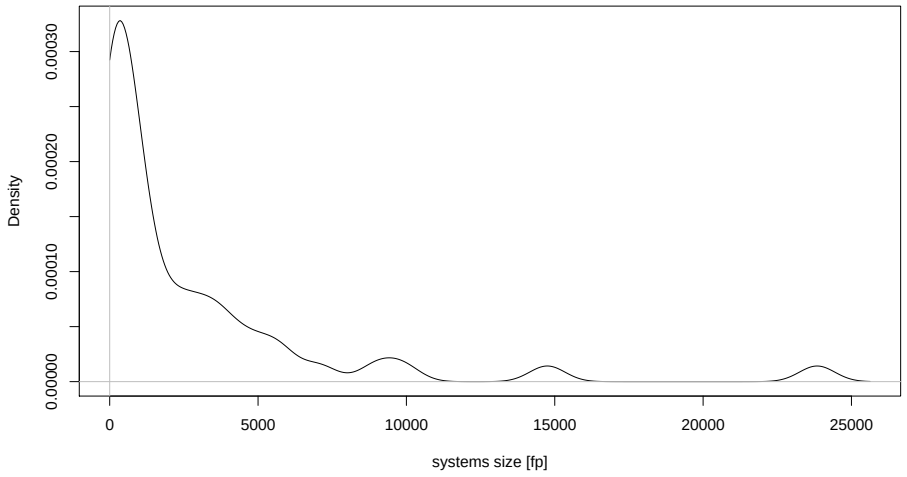

Figure 2.13: Distribution of the estimated function point counts.

The plot in Figure 2.13 shows a density function estimated on the basis of the obtained function point counts. The majority of the portfolio systems population fall in the lower end of the scale with the median of 905. Several outliers occur to the right of the scale. In [175] the authors also report on the software size distribution. There the sample consisted of 365 IT-projects. The projects originated from various industries and Cobol was among the top used programming languages. Interestingly, we found that the shape of the density function estimated for the systems originating from the portfolio studied in this chapter strongly resembles the one revealed in the other research.

We also present for comparison size distribution in another large software portfolio. We received its characteristics through the courtesy of Capers Jones. The portfolio serves a large USA conglomerate. It consists of 3000 in-house developed applications with over 340 million of lines of code. Programming languages used to write applications include: Cobol, PL/I, SQL, QBE, ASP, Visual Basic, Mumps, Chill, ESPL/1, Coral, Bliss, Assembly, Ada, Jovial, Forth, Embedded Java, Java, J2ME, Objective C, C, C++, C#, Perl, Ruby. In addition to that there are also COTS, open-source and end-user applications present.

Systems' size ranges	Number of systems
$> 10,000$ FP	20
$1,000 - 10,000$ FP	425
$0 - 1,000$ FP	2,555
Total	3,000

Table 2.7: Size distribution (function points) in a portfolio comprising more than 340 million of lines of code.

In Table 2.7 we present software size distribution in this portfolio. Size distribution is again similar to the one observed in the portfolio we used as case study. Interestingly, the number of very large systems ($> 10,000$) is relatively small (0.67%). In the portfolio we study we found only 2 systems of such size(4.2%).

Class property	Size class (FPs)				
	[0..100]	(100..1,000]	(1,000..10,000]	(10,000..100,000]	All classes
Systems	9	15	21	2	47
Min	14	108	1,035	14,749	14
1st Q	19	206	2,110	17,023	164
Median	22	427	3,158	19,296	905
3rd Q	26	708	4,905	21,570	3,361
Max	93	905	9,864	23,843	23,843
Total FPs	339	6,674	79,883	38,592	125,488

Table 2.8: Characteristics of the size classes of the portfolio systems.

In Table 2.8 we provide the distribution of the systems' size along with the basic statistics. Slightly more than half of the systems counts less than a thousand function points. Such sizes are common for systems regardless of the industry they originate from, as reported in Jones' database [79]. A significant number of the systems is large, more than a thousand function points. Among them we find two exceptionally large systems, above 10,000 function points. The total number of function points for all the systems together in the portfolio amounts to 125,488. Given the 20% accuracy of the backfiring technique the portfolio is within the range of 100,000 and 150,000 function points. To compare, a large international bank approximately owns 450,000 function points of software, and a large life insurance company possesses 550,000 function points [79, p. 51]. These reference figures suggest that we deal with a portfolio comprising a significant amount of software.

2.6.2 Essential information

Although the recovered information on the portfolio's size structure does not tell what implications there are for the management it does set foundation for carrying out such investigations. We assessed the portfolio from the point of view of an executive. Particularly, we discuss how we captured the portfolio's financial aspects relating to operations. We also present the way in which we assessed its market value. Additionally, we show what possibilities the codebase yields for carrying out what-if scenarios. To illustrate our points we present a few analyses performed for the management of the organization.

Systems importance The organization's internal documentation provided a three-level scale to rank systems in terms of their importance for the business operations. The level of a system's importance determines the expected resolution time for reported problems, level of service monitoring, and support outside the office hours. In Table 2.9 we present the systems ranking scale.

Importance level	Description
Critical	System's failure can result in either a crisis for the primary business operations and may incur substantial financial losses (including damage of the organization's reputation), or immediately affect internal processes of the organization.
Sensitive	System's failure results in a limited disruption of the internal processes, and has a limited risk on the financial losses of the organization.
Non-sensitive	System's failure can result in no to little damage for the organization.

Table 2.9: Levels of importance of the portfolio systems in business operations.

Importance level	Size class (FPs)				
	[0..100]	(100..1,000]	(1,000..10,000]	(10,000..100,000]	All classes
Critical	2	5	11	2	20
Sensitive	2	5	4	0	11
Non-sensitive	3	4	5	0	12
Not-rated	2	1	1	0	4
Total	9	15	21	2	47

Table 2.10: Distribution of the portfolio systems over size and their importance levels.

Table 2.10 provides the distribution of the systems over size and their importance levels. The importance levels were available for 43 systems. The remaining 4 systems did not have any importance level assigned and therefore we listed them as *Not-rated*. A significant number of systems is ranked as critical (20 out of 47). In this group we find the two largest systems of the portfolio. Interestingly, most systems which are ranked as critical are also large, more than 1000 function points. The systems ranked as sensitive and non-sensitive constitute approximately 23.4% and 25.5%, respectively.

Ongoing needs We carried out an assessment of the ongoing financial and staff requirements for the portfolio. Our calculations were performed on the basis of sub-portfolios.

Such an approach enabled us to analyze components of the portfolio at a higher granularity level than the one available at the level of individual systems. By doing so we were able to attach concrete figures such as monetary amounts, FTEs, function points, etc.; to concrete business concepts. And, make interpretation of the results more meaningful for the organization's management.

Importance level	Critical	Sensitive	Non-sensitive	Not-rated	All systems
Systems	20	11	12	4	47
Total FPs	89,379	13,620	21,107	1,382	125,488
Total FPs (%)	71.23%	10.85%	16.82%	1.10%	100.00%
P_{n_m}	119.2	18.2	28.1	1.8	167.3
P_{yco}	23.8	3.6	5.6	0.4	33.4

Table 2.11: The portfolio's benchmarked spendings and staff assignment.

In Table 2.11 we present our assessment of the ongoing financial and staff requirements for the portfolio. The P_{yco} values are given in millions of USD. Let us recall that in the P_{yco} formula we need to provide the daily burden rate (r) and the number of work days in a year (w). We assumed r to be 1,000 USD and w to be 200 days. Whereas the actual dollar value is likely to be imprecise due to volatile pricing conditions on the IT market, the point is to provide the order of the amount of money which is demanded from the organization to keep the portfolio up and running.

What is clear from Table 2.11 is that systems dubbed as critical constitute the largest group of systems in terms of function points. What is interesting is that the group of these 20 systems (20 out of 47, $\approx 43\%$) all together represent the majority of all the function points (71.23%) in the portfolio. This percentage also gives the approximate share of the staff and the yearly operational cost that is required to keep the systems up and running. The remainder is distributed over the sensitive, non-sensitive, and not-rated systems. The actual spendings and staff assignment were confidential and we do not present them. Still, the results we obtained on the basis of the codebase data and the public benchmarks allow the management to compare the IT condition with the industrial averages. On the one hand, significantly higher cost and staff assignment than the estimated values should raise concerns. On the other hand, significantly lower actual figures could indicate, for instance, good management practice, missing financial data, etc. Nevertheless, the assessment of this kind provides executives with an insight into some of the key parameters of the IT-portfolio.

Market value CIOs and CFOs will agree that IT constitutes an invaluable component of the organization of which loss would mean capital destruction for the company. But what amount of capital is potentially at risk? To answer this question we propose to carry out a market valuation of the IT-portfolio, and to accomplish this we use the benchmark formulas fed with the codebase data. We approximated the market value of the portfolio by estimating the portfolio's rebuild costs. Additionally, we derived the expected failure risk of such a venture, its duration and staff needed. Table 2.12 provides the results of our assessment.

Importance level	Critical	Sensitive	Non-sensitive	Not-rated	All systems
Systems	20	11	12	4	47
Total FPs	89,379	13,620	21,107	1,382	125,488
P_d	51.0	29.4	36.1	16.0	51.0
P_{n_d}	595.9	90.8	140.7	9.2	836.6
P_{rb}	356.2	34.9	64.2	2.3	457.6
P_{rf} (in-house)	24.11%	14.09%	15.37%	7.07%	18.08%
P_{rf} (outsourced)	15.69%	8.41%	9.48%	3.86%	11.39%

Table 2.12: Market valuation of the portfolio.

In Table 2.12 we present selected managerial indicators computed for the same sub-portfolios as those listed in Table 2.11. In the first two rows we provide the totals for the number of systems along with the estimated total function point counts, respectively. In the next three rows we provide estimates for the duration of rebuilding each group of systems (P_d given in months), the staff needed to accomplish this (P_{n_d}), and the average cost of rebuild (P_{rb} given in millions of USD). To carry out the last estimate we needed to assume the daily burden rate (r) for development and the number of working days in a year (w). We set r to 1,000 USD and w to 200 days. Finally, in the last two rows we provide rates for the groups rebuild exposure to failure (P_{rf}). This is done for two development scenarios: in-house and outsourced.

The estimates of the systems' rebuild costs show significant amounts. From these numbers we infer the market value of the portfolio. Furthermore, on the basis of the calculations it is clear that the time it would take to complete rebuilding each sub-portfolio is considerable. The most time consuming would be to rebuild the critical systems, 51 months (4 years and 3 months). The remaining sub-portfolios require minimum 16 months for completion. The sub-portfolios' exposure to the risk of replacement failure is in the range of 3.86% and 24.11%. The actual risk differs per sub-portfolio and depends on the rebuild strategy chosen (in-house vs. outsourced). The figures obtained through estimations suggest that we deal with a significant amount of capital that the studied portfolio represents. The software is a large and important asset of this organization. Moreover, by considering the expected time of systems' rebuild and the associated risk of failure it is clear we deal with a hardly replaceable IT-portfolio.

What-if Planning is one of the essential activities in sound IT management. One well-known method for predicting the potential future outcomes of decisions comes through scenarios analyses. This approach requires relevant data. We analyzed the potential impact that the growth in the portfolio's size may have for its management. For instance, how are the yearly operational costs affected in case the portfolio size increases by 10%. To accomplish our analyses we used the data available from the codebase.

Table 2.13 presents the data we used to estimate the compound annual growth rates ($cagr$) for three sub-portfolios: critical, sensitive, and non-sensitive. We skipped the smallest group of systems for which the importance levels were not rated. The values shown in Table 2.13 were derived on the basis of the known DC dates for the Cobol modules from the three sub-portfolios; similarly as in the example discussed earlier. Here

	Critical		Sensitive		Non-sensitive	
i	m_i	r_i	m_i	r_i	m_i	r_i
1	453	1.00000	36	1.00000	535	1.00000
2	529	1.16777	41	1.13889	562	1.05047
3	624	1.17958	49	1.19512	606	1.07829
4	737	1.18109	51	1.04082	656	1.08251
5	942	1.27815	93	1.82353	681	1.03811
6	1010	1.07219	106	1.13978	696	1.02203
7	1174	1.16238	119	1.12264	718	1.03161
8	1298	1.10562	121	1.01681	733	1.02089
9	1343	1.03467	125	1.03306	750	1.02319
10	1437	1.06999	136	1.08800	857	1.14267
$cagr$		12.24%		14.22%		4.82%
All mods	5634		950		1527	
DC mods	25.51%		14.32%		56.12%	

Table 2.13: Estimates of the compound annual growth rates for the source code implementing the systems from the three sub-portfolios: critical, sensitive, and non-sensitive.

we also considered the last 10 years of history. Each sub-portfolio was characterized with a vector of 10 pairs: the total number of modules (m_i) at time i and the ratio (r_i) between the numbers m_i and m_{i-1}. The ratios at time $i = 1$ were assumed to be 1. Underneath the columns r_i, in row $cagr$, we provide the estimates of the compound annual growth rates obtained on the basis of formula 2.17. In the last two rows we provide the total number of Cobol modules that fall into each sub-portfolio, and the percentages of the modules with the defined DC metadata.

With the estimated average growth rates for the three sub-portfolios at our disposal we now consider future growth of the sub-portfolios over the period of 10 years. In this period for each sub-portfolio we approximated the expected number of function points on a yearly basis. We assumed that the future growth of each of the sub-portfolios would consistently follow the corresponding $cagr$ estimates. Having the function point estimates available we then calculated the expected P_{yco}, P_{n_m}, and P_{rb}. For this we also needed to make a few additional assumptions. Let us recall that for the P_{yco} and P_{rb} formulas we need to specify two additional parameters: r, the daily compensation rate and w, the number of working days in a year. For the sake of simplicity we assumed that r remains at the level of 1,000 USD per day and w at 200 days per annum in the entire analyzed period of time. Of course, for a calculation concerning a long period we should take inflation into account. For the sake of explanation we left that out.

Figure 2.14 presents four plots each of which shows the calculated estimates for the expected number of function points, P_{yco}, P_{n_m}, and P_{rb} for each sub-portfolio. In each plot the horizontal axis shows the analyzed time frame. Ticks on the axis are labeled with integers ranging from 1 until 10. The vertical axes provide the values of the calculated indicators for years 1 through 10. The indicators referring to the sub-portfolio of critical systems are depicted with the solid line. The dashed lines refer to the sensitive systems, and the dotted lines to the non-sensitive. Clearly the lines of the same type in all of the plots follow similar patterns. This is not surprising given the definitions of the P_{yco}, P_{n_m}, and P_{rb} formulas which all dependent on the function points and are all increasing.

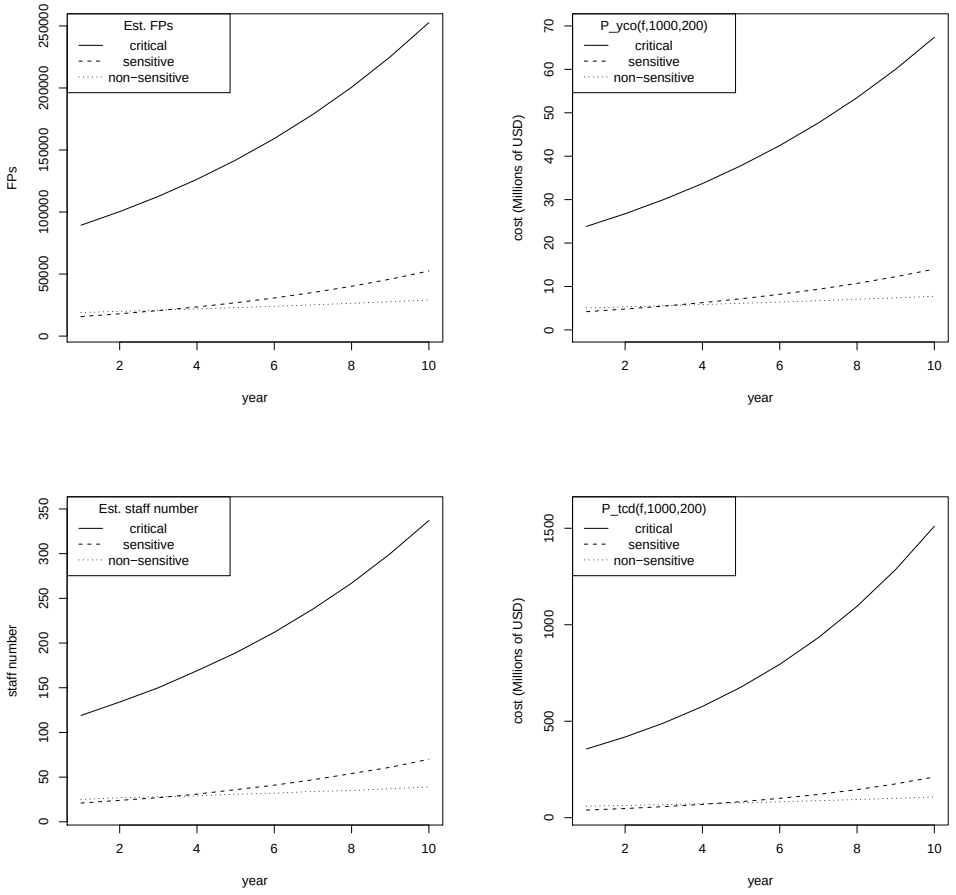

Figure 2.14: Growth scenarios of three different sub-portfolios.

Subject to the most notable changes are the solid lines which represent the sub-portfolio with critical systems. To explain this let us recall that these systems implement the vast majority of the function points ($> 72\%$). The important information conveyed through the plots is in the ratios between the future estimates and the present estimates. Let us analyze these ratios for the group of critical systems. In 10 years time the total number of functions points is expected to increase from approximately 90,000 to 250,000 function points. This means an increase of about 270%. What follows from it is that the expected yearly operational costs and staff number will increase accordingly. From the market value perspective we infer a change by 424%. Of course, our analyses are done under very rigid and simplistic assumptions. Nevertheless, they clearly illustrate the IT-portfolio management support which is attainable on the basis of source code derived data, and

benchmark formulas.

2.6.3 Vendor-locks

An important element of systems development and maintenance are third-party technologies. These technologies once selected for the portfolio constitute its nondetachable part; making a switch almost prohibitive. In the words of Michael Porter: there are low entry barriers, but high exit barriers [123, p. 22]. There exist numerous locks for the portfolio created by the technology vendors. Once the technologies are chosen the vendors are in a position to dictate conditions of usage. And, this situation asks for a managerial oversight so that various IT related risks and costs remain controllable.

There are a number of aspects to take into account when dealing with third-party technologies. Let us consider some of them. First, usage of technologies requires access to trained programmers. Depending on the technology difficulties may arise if the availability of the programmers on the market falls short. Next, a vendor may cease support for its technology. Such an event is likely to trigger the need for migration, and that alone carries plenty of other risks for the portfolio. Finally, the technologies are not for free. As it turns out the associated license fees alone constitute a substantial cost component, especially when measured at the portfolio level. There are clearly reasons to take actions to mitigate the existing risks and control costs. To be able to make any decision it is vital to have adequate insight into the portfolio.

We now show how to use the codebase data to obtain portfolio-wide insight into third-party technologies. In particular, we focus on the code generators and the compilers. In each case we show the characteristics obtained, present findings, and discuss them in the context of portfolio management.

Code generators breakdown To capture the extent to which the source code in the studied IT-portfolio is dependent on code generators we analyzed the distribution of the modules over the code generators we detected during our earlier analysis. For this purpose we used the GEN metadata. Let us recall, $GEN(m)$ provides us with the name of the code generator used to write the source code of a module m. All modules for which no code generator was detected are assumed to be hand-written.

Code generator	Modules	(%)
TELON	1,922	23.4447%
COOL:Gen	1,371	16.7236%
Advantage Gen	1,337	16.3089%
CANAM Report Composer	58	0.7075%
KEY:CONSTRUCT FOR AS/400	18	0.2196%
Hand-written	3,492	42.5958%

Table 2.14: Distribution of the portfolio Cobol modules over the detected code generators.

Table 2.14 presents the distribution of the portfolio Cobol modules over the detected code generators. In the first column we list the names of the code generators encountered in the portfolio. In the second column we provide totals for the number of modules

generated with each of the CASE tools, and in the third their percentage with respect to the total number of modules in the portfolio. We find three code generators that account for over a half of the modules: TELON, COOL:Gen and Advantage Gen. Let us recall that Advantage Gen is a synonym for COOL:Gen. Next to the major tools we also find two code generators which cover relatively small portions of the portfolio: CANAM Report Composer and KEY:CONSTRUCT FOR AS/400. We find 76 modules in total which constitute less than 1% of the portfolio modules. The remaining portion of the modules, labeled as *Hand-written*, contains hand-written source code.

We consulted our findings with the portfolio experts. All code generators except for KEY:CONSTRUCT were recognized immediately. TELON, COOL:Gen, and Advantage Gen were highlighted as the core development tools maintained for the portfolio. As it turned out the KEY:CONSTRUCT code generator which is responsible for the very few modules was not known. This finding is interesting as it suggests that there are discrepancies between the code generators lists obtained from source code analysis and the one that is known to the portfolio experts. Moreover, the fact that the tool was not known may also suggest that it is not available for the programmers. Such a situation obviously raises potential problems with maintainability.

Migration impact One important information that we learned from Table 2.14 is that maintenance of over half of the portfolio modules depends on third-party technologies. Furthermore, from the expert team we learned that usage of code generators does not guarantee a positive effect on the development productivity. For instance, the organization's internal benchmarks suggested that TELON based developments suffer from lower productivity factors than those made in native Cobol. This observation certainly raises a question on the rightfulness of using this tool for this portfolio. Nevertheless, from the risk and cost perspective devising exit strategies for some of the technologies looms as a tempting option. Dissociation of code generators from a Cobol portfolio has benefits which include, for instance, lowering of the maintenance costs through removal of license fees, elimination of reliance on skills which are in a diminishing resource pool, or introduction of an easily maintainable native Cobol code [14, 137, 139]. From the practical point of view dissociation of code generators would mean execution of a migration project (e.g. TELON to native Cobol). Let us note that migrations are in general deemed to be risky endeavors [151]. And, when a large and business critical IT-portfolio worth millions of dollars is at stake it is crucial to comprehensively evaluate the associated risks.

We now show that on the basis of the codebase data it is possible to support migration decisions. To gain the idea about the possible consequences a migration from the code generators to native Cobol could have for the studied portfolio we carried out two analyses. First, we studied the distribution of the code generators over the systems' importance levels. This allowed us to recognize parts of the portfolio which are threatened with the migration related risks. Second, we analyzed the impact on staff and licenses.

In Table 2.15 we provide an extended view into the distribution of the portfolio modules over the code generators in which an additional dimension is taken into account; the importance levels of the systems. Let us recall, the importance levels were provided on a per system basis. We derived the importance levels for the modules by first assigning each module to the system it implements, and then looking up the importance level of that

	Critical		Sensitive		Non-sensitive		Not-rated	
Code generator	#	%	#	%	#	%	#	%
ADVANTAGE	610	7.44%	12	0.15%	715	8.72%	0	0.00%
CANAM	54	0.66%	2	0.02%	2	0.02%	0	0.00%
COOL:GEN	1,292	15.76%	62	0.76%	17	0.21%	0	0.00%
KEY:CONSTRUCT	18	0.22%	0	0.00%	0	0.00%	0	0.00%
TELON	1,458	17.78%	381	4.65%	20	0.24%	63	0.77%
HAND-WRITTEN	2,202	26.86%	493	6.01%	773	9.43%	24	0.29%
Totals	5,634	68.72%	950	11.59%	1,527	18.63%	87	1.06%

Table 2.15: Distribution of the portfolio Cobol modules over the code generators and the systems' importance levels.

system. We then partitioned the modules into four subsets which we labeled: *Critical*, *Sensitive*, *Non-sensitive*, and *Not-rated*. Modules which implement the critical, sensitive, and non-sensitive systems were placed in the corresponding subsets. All the modules which belong to the systems of unspecified importance level were assigned to the *Not-rated* subset. The first column in Table 2.15 lists the code generators. In the subsequent columns we provide the modules' distribution over the code generators for each subset. Each subset is characterized with two vectors. The first vector gives the module totals and the second their percentage with respect to the total number of modules in the portfolio.

The *Critical* subset is the largest with as many as 68.72% of all the portfolio modules. This percentage includes 26.86% of the hand-written modules and 41.86% of the generated modules. Let us note that in the entire portfolio the generated sources constitute 57.4% of all the modules (derived from Table 2.14). This means that the vast majority of the CASE tools dependent Cobol sources form the implementation of the systems which are deemed critical. This fact is striking since it implies that any source code migration project concerning systems from the critical sub-portfolio is expected to have a relatively high impact on the business. Let us also note that the critical sub-portfolio is also the most diversified in terms of the number of different code generators. We find there all 18 KEY:CONSTRUCT modules and 54 (out of 58) CANAM sources. Having to tackle with a large number of different technologies in a migration project would certainly escalate its complexity.

Code generator	modules	Est. FPs (f)	$n_m(f)$
HAND-WRITTEN	3,492	26,776	35.70
TELON	1,922	41,031	54.71
COOL:Gen	1,371	21,971	29.29
ADVANTAGE	1,337	24,143	32.19
CANAM	58	809	1.08
KEY:CONSTRUCT	18	221	0.29

Table 2.16: Distribution of the portfolio modules over staff needed for maintenance.

Table 2.16 shows the distribution of the portfolio modules over the staff needed for maintenance. In the first column we list the names of the code generators and in the second we provide the total number of modules generated by each code generator. For each code

generator we approximated the number of function points that the modules implement. For this purpose we used the backfiring technique. The results are listed in column three. In the fourth column we provide the approximate the size of the team needed to maintain the modules of each code generator. We obtained these figures on the basis of the function points by using formula 2.6. It is immediately clear that a significant number of staff is needed to maintain the generated Cobol sources. By adding up all the n_m values except for the one corresponding to the hand-written code we arrive at a maintenance team of 117.56 FTE. This number constitutes nearly 77% of the entire estimated personnel. The figures listed in column four can be also interpreted as the number of licenses required for the portfolio. In case of the studied portfolio the experts use a proprietary model for determining the numbers of licenses that need to be purchased annually. We cannot share details concerning this model due to confidentiality. The experts claim that the costs relating to TELON and COOL:Gen constitute a substantial fixed component of the yearly operational spendings on IT. To get an idea of a license price we used public data and retrieved a figure of 84,045 USD for a TELON license [1]. Of course, to estimate a bill due for licenses at the portfolio level one must take into account the actual licensing model used in an organization. Nevertheless, the license related costs alone are high and therefore deserve adequate attention from the IT executives. The insights we obtained through our analyses are supportive in various decisions, for instance, assessment of gains from a migration project.

Last compilation The ability to successfully compile source code is vital in assuring maintainability of an IT-portfolio. Dealing with the compilation of a large and decades old portfolio can be challenging. For instance, in Cobol it is not difficult to affect code semantics through the use of different compilers. In [163, p. 35] the author presents findings concerning different semantics for a `PERFORM` statement from analysis of 8 different Cobol compilers. Even an upgrade of a compiler to a newer version does not guarantee semantic consistency. Not to mention that simple alterations in the compiler's flags may affect the semantics [102]. Despite all those inconveniences compiler migrations are a part of the software's life cycle. Reasons for migration include, for instance, withdrawal of support by a vendor, the requirement from the business to have all business applications run with the best possible performance, or elimination of the cost of unsupported compilers [58, 83]. Whatever decision is being made with respect to the compilation environment it is crucial to be able to understand the impact the change has for the portfolio. We now show how to use the code extracted data to obtain a portfolio wide insight useful in diagnosing the compilation environment.

We analyzed the compilation environment of the studied portfolio using the data characterizing its last compilation. Let us recall, in our codebase we find two metadata relating to compilation: CN (compiler name) and LC (date of last compilation). We first studied the compilers and their usage. Table 2.17 provides the names of the compilers along with the distribution of their usage over the portfolio modules.

In the first column of Table 2.17 we list the names of the compilers derived from the CN metadata. In the second column we provide the total number of Cobol modules compiled, and in the third the percentage of the modules with respect to the total number of modules. All the listed compilers are IBM products. Of these compilers all except for

Compiler name	Modules	%
IBM COBOL FOR MVS & VM	15	0.18%
IBM COBOL for OS/390 & VM	4,003	48.83%
IBM Enterprise COBOL	1,002	12.22%
IBM OS/VS COBOL	3,178	38.77%

Table 2.17: Last compilation of the portfolio modules: compilers coverage.

IBM Enterprise COBOL had already been withdrawn from support. What strikes, is the amount of modules compiled with IBM OS/VS COBOL, 38.77%. Let us note that IBM OS/VS COBOL is one of the oldest IBM compilers. It supported the very old Cobol 68 standard. IBM ceased supporting the compiler already in 1994, that is 16 years ago at the time of writing this chapter [58]. Source code compiled with IBM OS/VS COBOL is a good example of the area in the portfolio that requires a technology related change. It is important to note that a compiler upgrade means possible source code adaptations. And, these are known to be associated with various risks. Given the significant number of modules compiled with IBM OS/VS COBOL the impact of a compiler migration project on the portfolio is likely to be high.

To gain a better understanding of the usage of the IBM OS/VS COBOL compiler we analyzed the times of the last compilations. For each module in the portfolio we calculated the number of days that has elapsed since the latest module compilation in the portfolio. We used as our reference point the latest LC date.

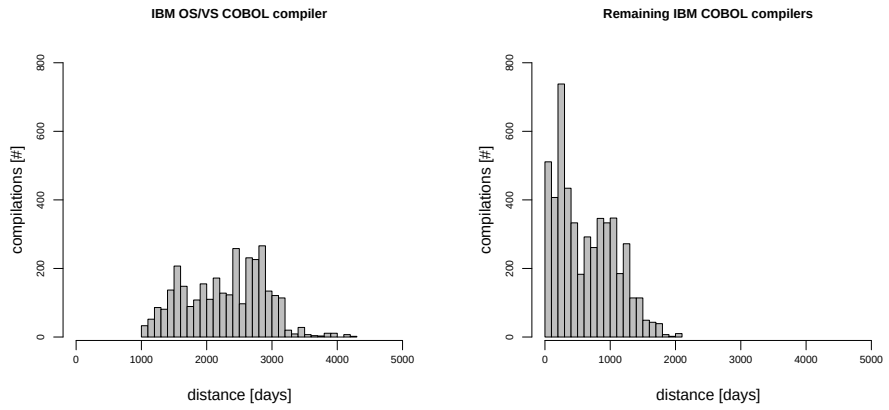

Figure 2.15: Compilations in the portfolio: the IBM OS/VS COBOL compiler and the remaining compilers.

In Figure 2.15 we present two histograms. The left one shows the distribution of the time intervals which apply to the IBM OS/VS COBOL compiled sources. The right one shows the distribution for the remaining modules. There is a notable difference between the two histograms. All the last compilations with IBM OS/VS COBOL took place

between 1000 and 4300 days since the latest compilation in the portfolio. For the compilations done with the remaining group of compilers the time frame spans between 0 and 2100 days. The larger distance in compilations with the IBM OS/VS COBOL suggests the lack of changes in a significant period of time. According to our checks the IBM OS/VS COBOL compiled modules are part of the implementation of as many as 38 (out of 47) different information systems. On the basis of the mainframe usage reports we determined that some of these systems belong to the top most used applications in the portfolio. So this observation clearly suggests that the related modules are important. It is also clear that the modules appear to form an area in the portfolio which is out of ordinary, and therefore worth investigating. With the kind of analysis we carried out it was possible to disclose the anomalies.

	Critical		Sensitive		Non-sensitive		Not-rated	
Compiler name	#	%	#	%	#	%	#	%
IBM COBOL FOR MVS & VM	15	0.18%	0	0.00%	0	0.00%	0	0.00%
IBM COBOL for OS/390 & VM	2962	36.13%	302	3.68%	730	8.90%	9	0.11%
IBM Enterprise COBOL	497	6.06%	147	1.79%	358	4.37%	0	0.00%
IBM OS/VS COBOL	2160	26.35%	501	6.11%	439	5.35%	78	0.95%

Table 2.18: Distribution of the compilers usage over the systems' importance levels.

Similarly as for the code generators we obtained the distribution of the compilers usage over the systems' importance levels. The results are shown in Table 2.18. We see that a large percentage of modules forming the implementation of the critical systems are compiled with old compilers. Compilations with the IBM OS/VS COBOL account for over a quarter of all modules. Whatever decisions the IT executives are to make with respect to the compilers upgrade it is clear, on the basis of the insights we obtained, that migrations will concern a significant portion of the portfolio and will hit many business critical systems.

2.6.4 Nuts and bolts

Whereas source code appears to be a distant entity from the business management perspective it is prohibitive not to allocate it the attention it deserves. Problems resulting from maintenance of poor quality code have a negative impact on productivity and often lead to unwanted costs and time overruns. Insight into code quality is vital in order to control potential maintenance risks. Naturally, obtaining it is infeasible without actually screening the source code, and therefore source code analysis methods are the basis for implementation. Apart from the tooling, one also requires code quality standards that source code is expected to adhere to. To obtain these standards organizations may resort to software engineering experience. In the analyses presented in this chapter we characterize source code quality by looking at the cyclomatic complexity and the presence of obsolete programming language constructs. Whereas these views certainly do not exhaust all possible quality checks one can conduct on Cobol sources they suffice to illustrate the quintessence of our approach to obtaining code quality control information on a portfolio-wide scale.

Under the hood Modules with large cyclomatic numbers are expected to be difficult to maintain and error-prone. Similarly, modules which contain obsolete programming constructs hamper longevity of the code due to possible code incompatibilities with newer compilers. Some programming constructs also complicate understanding of the program's semantics and therefore impede effective alterations of the code. All these aspects are particularly relevant for the hand-written modules, and for this reason we limited our analyses to such modules only. In the studied portfolio we find 3,492 (42.6%) programs which meet this criteria. For the MC metric we analyzed the distribution of its values over the selected programs. With respect to the obsolete constructs we primarily studied the proportion of those modules in the portfolio. Whereas occurrences of the obsolete constructs are in general undesired, not all of them cause obvious damage. For instance, from a program's semantics point of view the *documentation statements* are meaningless. Also, they do not impede code compilation. In fact, their presence turned out to be useful for our study since we could extract modules' creation dates. However, in order not to obscure the portfolio quality insight with unnecessary details we excluded these statements from our analyses. We treated one construct, the `ALTER` statement, separately for its reputation for complicating program comprehension.

Metric	Modules	%	Min	1st Qu.	Median	3rd Qu.	Max
MC	3,492	100.00%	1	16	33	71	923
OBS_{alter}	16	0.46%	1	1	1	1	10
OBS_{other}	630	18.04%	1	1	2	3	24

Table 2.19: Code characteristics of the hand-written modules.

Table 2.19 presents the code characteristics of the hand-written modules. In the first column we list the quality metrics: MC, OBS_{alter} which is the `ALTER` statements counter, and OBS_{other} which is the cumulative counter of all the obsolete constructs excluding both the *documentation statements* and the `ALTER` statements. In the second column we provide the total number of modules for which the metrics give non-zero values. For MC the number is equivalent to the total number of hand-written programs. In case of the obsolete constructs it is the total number of modules in which the constructs occur. In the third column we give the percentage of the total number of the hand-written programs. And, in the remaining columns we provide the five-number summaries of the metrics.

The number of modules with the `ALTER` statement is relatively low, constituting less than 0.5%. The statement occurs sporadically with only 1 module having as many as 10 `ALTER`s. All the modules were introduced in the 70s as we read from the corresponding DC dates. We learned from the portfolio experts that a project for gradual removal of the `ALTER` statements was initiated within the organization at a certain point in time. This might explain the low number of `ALTER`s across the portfolio. In the portfolio we also find 630 modules with all kinds of other obsolete constructs. They constitute a fifth of the hand-written programs. For the major part their occurrence is sporadic in the modules' code, with an increased density in the highest quartile.

A typical way to use McCabe's cyclomatic number is to compare observations with a predetermined threshold. Programs for which the cyclomatic number exceeds the chosen

threshold are deemed to be risky for maintenance. McCabe proposed a theoretical threshold of 10 for the programs [108]. In [49] we find that Hewlett Packard requires modules with a cyclomatic complexity higher than 16 to be re-designed. In [75] we find that modules for which McCabe exceeds 50 are very risky to maintain. In the studied portfolio we have 1,237 modules ($\approx 35\%$ of the hand-written and $\approx 15\%$ of all the modules) with a cyclomatic number of over 50. This clearly suggests complicated program structures in a relatively large portion of the portfolio. Although, it does not follow automatically from this that the programs suffer from maintenance problems it is clear that a question for possible re-design should be raised. Nevertheless, through the analysis it becomes clear for what portion of the portfolio actions with respect to maintenance risk mitigation should be considered.

Strategic control Source code quality control can be viewed as yet another process within the organization. Opting for its implementation will imply additional investments. Ideally, source code quality control is incorporated as a portfolio-wide process. Such a process would involve identification of the quality flaws and their improvement. From the practical point of view one must take cost-efficiency into account. One way to achieve this is to allocate quality improvement efforts to portions of the portfolio which need it the most. We propose to consider volatility in the portfolio code as a measure to identify parts in need of improvements. We measure the volatility of a certain part of the portfolio (module, system, sub-portfolio, etc.) by counting the number of times its code was subject to some activities (modifications, compilations, etc.) during a fixed period. From the IT-management perspective insight into volatility is important since it provides information on where the effort is spent. Given the source code quality perspective, when volatile layers happen to be implemented with low quality code the chance of running into maintenance difficulties escalates. One way to get insight into the volatility in the portfolio is to study the historical data which tells us something about the changes. In our case study we have at our disposal data characterizing code changes which apply to individual Cobol modules. We assume modules to form the lowest code layer in the IT-portfolio. From there, using various aggregation schemes, we are able to obtain volatility measures of the IT-systems and the sub-portfolios.

For volatility analysis we used the DC, CHG, and LC metadata as surrogates for the modules version history data. Let us recall, given a module m the $DC(m)$ provides the creation date, $CHG(m)$ the list of dates reported in the module's source code comments as records of code related activities, and $LC(m)$ the latest compilation date. All these dates combined together form a list. Each item on the list corresponds to some historical code related activity taken with respect to the module m. We utilized this list purely for the purpose of obtaining counts of the total number of coding activities relating to the modules. Let us also recall that while the LC date is available for all the modules, the DC and CHG are not. In particular, they are absent for the majority of auto-generated modules. In total we found 5,177 Cobol modules ($> 63\%$ of all) with either DC or CHG, or both, defined.

To measure volatility we needed to fix a time period. We chose to consider data reaching backwards over a period of 10 years from the latest compilation reported in the codebase (the latest LC date). We found 8,163 modules for which at least one of the dates

(DC, CHG, and LC) falls into the chosen time frame. For each module we divided its total number of dates relating to the code activities by 10 to obtain the yearly average. The five-number summary for the obtained vector of averages is as follows: the value for both the minimum and the first quartile is 0.1, the median is 0.3, the third quartile is 0.4, and the maximum is 7.4. These numbers already tell us that for the vast majority of modules there are hardly any activities reported. The values falling into the top 25% range show more variability with values ranging between 0.4 and 7.4.

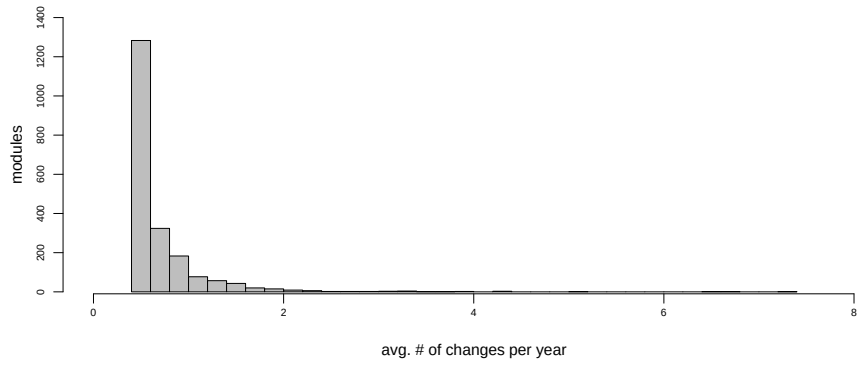

Figure 2.16: Distributions of the average number of code related activities over the top 25% of most volatile Cobol modules.

Figure 2.16 shows the distribution of the average number of code related activities for the top 25% of the most volatile Cobol modules. We find 251 Cobol modules, $\approx 3\%$ of all modules in the portfolio, with the average number of code related activities per year exceeding 1. Among those modules 203 are hand-written modules and the remaining TELON generated. The height of the bars in the histogram steeply decays from left to right. The histogram clearly shows that in the portfolio there are relatively few source modules which were frequently subject to code related activities. Such source files are typically part of implementations in which alterations are driven by pure business demand or plain necessity resulting from encountered defects. In either case they are worth deeper investigation. For the sake of future reference we will call these modules *hot spots*.

Hot spots We now elevate our analyses from the level of modules to the levels of information systems and sub-portfolios. We considered the *hot spots* and assigned them to the information systems they implement. This operation resulted in distribution of the 251 hot spots among 27 (out of 47) systems. In this group 14 systems were ranked as critical. We checked properties of the distribution of the hot-spots among the systems. The five-number summary is as follows: the minimum is 1, the first quartile is 2, the median is 5, the third quartile is 13, and the maximum is 42. It is clear that there are relatively few systems with many hot spots, and the vast majority of the hot spots occurs in the top

25% of the systems. In Table 2.20 we provide detailed characteristics of the top 25% of systems.

ID	Hot spots	Generated	Hand-written	Top volatile	Importance Level
S_1	42	8	34	6.8	Critical
S_2	37	12	25	5.1	Critical
S_3	27	0	27	7.4	Sensitive
S_4	19	8	11	3.8	Critical
S_5	16	12	4	2.4	Critical
S_6	16	1	15	2.0	Critical
S_7	13	0	13	2.1	Non-sensitive

Table 2.20: Characteristics of the top 25% systems ranked in terms of the number of the hot spots they contain.

The first column in Table 2.20 provides identifiers of the systems. In the second column we show the total number of modules from the hot spots set. In columns three and four we specify how many of the hot spots are auto-generated and hand-written, respectively. In the penultimate column we give the maximum of the average yearly numbers of the code related actions reported for the hot spots in each system. In the last column we provide the importance levels assigned to the systems by the portfolio experts. Most of the systems are considered critical. Also, the majority of the hot spots are modules which are hand-written. We inspected the quality criteria for the hand-written hot spots and compared them with the remaining source modules.

Group	Metric	Modules	%	Min	1st Qu.	Median	3rd Qu.	Max
Hot-spots	MC	203	5.81%	3	95	158	245.5	923
	OBS_{alter}	0	0.00%	0	0	0	0	0
	OBS_{other}	5	0.14%	0	0	0	0	5
Other	MC	3,289	94.19%	1	15	30	63	876
	OBS_{alter}	16	0.46%	1	1	1	1	10
	OBS_{other}	625	17.90%	1	1	2	3	24

Table 2.21: Characteristics of the hand-written modules from two groups: *hot spots* and other modules.

Table 2.21 provides characteristics of the hand-written modules from the two groups: *hot spots* and other modules. The content for Table 2.21 was generated in a similar manner as the one for Table 2.19. When comparing the two groups of Cobol modules it is clear that the distribution properties for the metrics differ. It is especially visible for the MC metric where the corresponding values of the five-number summary are much higher for hot spots. The vast majority of the hot spots have McCabe values exceeding 50, suggesting program structures which are very risky to maintain. The other observation is that the majority of the hot spots is mainly free from obsolete language constructs. There are no `ALTER` statements, and the remaining constructs occur sporadically. The presented insight into the portfolio hot spots provides IT managers with information which can support, for instance, decisions concerning allocation of the source code quality improvement efforts.

2.7 Discussion

Our approach to recovery of management information has been demonstrated on the basis of a large industrial Cobol based software portfolio. Cobol has been the major programming language for IT applications. Naturally, it is not the only programming language used in the industrial setting. More recent languages, like Java, C#, Objective C, Ruby, etc., are often encountered. There are also other legacy languages such as Ada83, mumps, Jovial, CHILL, C, Fortran, etc. One aspect of our approach that might interest portfolio managers is whether our method would be applicable to software portfolios based on these other languages?

Java, C#, Objective C, Ruby are more modern languages and for instance they often do not contain an evolution history as in the more traditional languages like Cobol or Fortran. Most often in the case of modern languages there is also version control support like CVS/SVN where a version history is denoted. Once you recovered that either from comments in the source code (the evolution history) or via the version management system (or both) you can use exactly the same analysis techniques as are used for Cobol. Also other aspects from our Cobol study can be applied to other languages. Of course, Jovial and Chill are for implementing real-time systems, think of AWACS in Jovial and PBXes in Chill. For such application areas often more information is present like up-to-date documentation, defect administrations, incident administrations, configuration administrations, release administrations and (of course) version administrations. Similar to the recovery of data from source code you can then additionally recover data from the documentation and the accompanying administrations. Sometimes such administrations are a bundle of word documents (think of release notes) and sometimes it is highly structured information combined with textual information in professional version management systems. Either way, with our approach it is possible to extract data and analyze it using exploratory data analysis.

We give two examples to illustrate a broader use of the approach in the IT setting. In one case the technical state of a government software portfolio was not objectively known. Some argued it was in a bad state and an overhaul of the entire (20 million line) portfolio was unavoidable. Others, mainly the programmers, disagreed. The executives felt that updates took too long, and maintenance was too expensive. In such a situation factual information recovered from the code and administrations of the underlying IT portfolio can shed light on the true state of the software. In this case there was an elaborate incident administration that was very well maintained and kept up to date. From incoming calls of citizens to issues from inside the organization. It turned out that outside callers were always registered by their social security number (SSN). Since such numbers are not random but follow a certain algorithm it was possible to extract all these numbers form this administration. This data was analyzed using EDA. First a frequency plot was made, which showed the number of times each SSN occurred in the incident administration that spanned a period of 12 years. Not a single SSN turned out to occur more than 15 times. Note that high frequencies of this number are an indication of a recurring problem or of problems in the software. This was not the case. Then a time sensitive plot was made: the amount of SSNs over time aggregated by month. The highest level never topped 5, which means that on average never more than 5 people called with an incident over all ten

years. This was a second strong indication that there were ever operational problems with the software, and that only seldom citizens complained, and that there were no cases that kept on returning since problems persisted.

In another case in the private sector for a merge/acquisition an IT due diligence was necessary [12]. Also here the question was what the state of the software comprised. Now often IT due diligence must be carried out quickly, and there is no time for a detailed analysis. So the same approach was used as above. There was a fairly good incident administration. In this case all the customers were identified by their customer identifier. Also that identifier was recoverable via an algorithmic test. Here it turned out that the frequencies were high, median higher than the top in the former case with outliers to hundreds of complaints per month. Also the monthly complaint rate showed high values often in bursts. These turned out to match with the release dates in the release administrations. Also plots of reoccurring customer identifiers over time showed complaints that were not solved for many months. This gave rise to significantly lowering the value for the software in the books. The owner of the software acknowledged after this discovery that the software was not as maintainable as outlined in their offering.

The point we like to make here is that by extracting data relevant for certain questions and using often visual statistical tools known from EDA it is possible to obtain answers on relevant questions for management. Whether it be projections of future maintenance costs in an outsourcing context or whether the price in the books for the software in an acquisition deal is realistic, the method is the same: extract data using simple tools, analyze the data using EDA, incorporate the domain knowledge and, if possible, request feedback from domain experts. In this way fact-based answers can be obtained.

2.8 Conclusions

IT management urgently needs relevant information which enables risk mitigation or cost control. However, as it turns out the required information is frequently either missing or its gathering boils down to daunting tasks which do not always deliver results. In this chapter we showed how to exploit the concealed source code data to yield the information needed in IT-portfolio management. In particular, to obtain the data we analyzed the source code statements, source comments, and also compiler listings. We demonstrated how to depart from the raw sources, process them, organize, and eventually utilize so that the bit-level data gets leveraged to the portfolio level and becomes useful for board-level executives.

In this work we analyzed a Cobol IT-portfolio of a large organization operating in the financial sector. We dealt with a mixture of Cobol code written manually and generated with CASE tools, such as TELON, COOL:Gen, CANAM, and others. The portfolio is decades-old and large in many dimensions; for example, in terms of lines of code, number of systems, or number of modules. It contains more than 18.2 million physical lines of code, partitioned over 47 information systems. Some Cobol programs date back to the 1960s.

To enable data extraction we developed an inexpensive analysis facility which we applied to the portfolio under study. With this and our other study we showed that our approach is applicable on an industrial scale [100]. Bearing in mind the principles behind

the design of our approach there are no limitations as to its scalability and applicability in practice. We discussed a number of the more common managerial quandaries. On the basis of a number of examples we showed how to utilize the code extracted facts to deliver support in resolving these quandaries.

Our approach enabled us to provide various managerial insights into the IT-portfolio. Apart from recovering information on the essential properties of the portfolio, for instance, the size of the information systems, we were also able to estimate the growth rate of the portfolio using source code derived data. We showed that the amount of source code in the studied portfolio expands annually by as much as 8.7%. We also showed how to estimate the portfolio market value, how to assess the cost of operations, and the staff assignment scope. Using the recovered information we performed a number of what-if scenarios for the portfolio to project future managerial indicators. We exposed various technology related challenges for the management. For instance, as it turned out maintenance of almost 60% of the portfolio source modules depends on expensive CASE tools. Almost 40% of the modules rely on the no longer supported IBM OS/VS COBOL compiler. Approximately 15% of the modules suffer from excessive code complexity. We also analyzed various migration scenarios for the code generators and compilers. We found that such migrations will have a relatively high impact on the top critical business applications. Furthermore, the complexity of the migrations turned out to be non-trivial. For instance, we found that the Cobol implementation of the top critical systems involves as many as 4 different code generators. All the presented insights were discussed in the context of the organization which operates on the studied IT-portfolio.

By reaching for source code it becomes possible to obtain information not available to IT-executives otherwise. So, aligning code analysis with IT management delivers new dimensions. It is possible to easily access information which has a potential to support decision making at the strategic level.

In this work we restricted ourselves to obtaining insights into the IT-portfolio that address some more common managerial quandaries to illustrate the essence of our approach. All assumptions were made explicit, so that executives can adjust the assumptions to their own specific situation. For example, in the case of benchmarks organizations can use their own custom figures, provided they are available. The quintessence of using code to fuel decision making at the board level does not change by that. Finally, we believe that our work will motivate the use of source code analysis to support decision makers with IT-portfolio management.

CHAPTER 3

Reducing operational costs through MIPS management

3.1 Introduction

Information technology plays an important role in many organizations. In our contemporary world, business environments have become global and significantly escalated challenges for delivery of adequate computing services which meet stringent performance goals and operate at low cost. A quote by Benjamin Franklin - “A penny saved is a penny earned” - is very up-to-date in todays business reality, for more capital available for the business means more investment opportunities to be pursued. IT related operational costs are high and no wonder that businesses ought to strive for their reduction. For Dutch banks it was estimated that total operational IT costs oscillate at around 20%–22% of total operational costs [12]. Businesses have taken a strong stance on operational costs reduction and sought solutions to slash IT costs. For instance, Citigroup estimated that removal of redundant systems from their IT portfolio yields a savings potential of over 1 billion USD [52]. So far, organizations have tried a number of approaches to cut IT costs which included staffing reduction, outsourcing, consolidating data centers or replacing old software and hardware with its newer counterparties. All these approaches, however, carry an element of risk and might end-up costing a lot especially when complex software environments are involved. There exists an approach which enables IT costs reduction at low-risk for both business and IT. It involves management of CPU resource consumption on the hardware platforms.

MIPS, a traditional acronym for millions of instructions per second, have evolved to become a measurement of processing power and the CPU consumption. MIPS are typically associated with running critical enterprise applications on a class of computers known as mainframes. The term originates from the compartments where these computers used to be housed: room-sized metal boxes or frames [33]. From a business perspective mainframes have proven to be secure, fast and reliable processing platforms. However, running computer programs on mainframes incurs costs. MIPS related costs are inevitable

and high for those organizations which have grown on top of mainframes. As Nancy White, a former CIO of Certegy corporation, once said: "MIPS and salaries are my highest unit cost per month" [18]. One obvious question that arises is: why not use platforms where running programs does not incur costs? Running programs costs money on every computing platform. Mainframe users have traditionally chosen for usage fees because of the shared nature of the platform (many users use one big computer). However even when you run your programs in a large cluster of virtual machines you get charged on the capacity you reserve and/or use. Moving to another platform (from mainframes) might bring cost savings, but such step is not risk free.

Moreover, in reality business managers must deal with legacy applications. Software development in the domain of management information systems (MIS) has been dominated over the past decades by Cobol. Studies show that Cobol is used to process 75% of all production transactions on mainframes. In the financial industry it is used to process over 95% of data [8]. A recent MicroFocus's survey claimed that the average American interacts with a Cobol program 13 times a day and this includes ATM transactions, ticket purchases and telephone calls [53]. Furthermore, the Cobol programming language constantly evolves and adapts itself to the latest technology trends [76, 126]. This is especially visible in case of web enabled mainframe applications, where modern web front-ends cooperate with legacy back-end systems. For organizations, which have been dependent on mainframes for years, any strategy which supplants this reliable hardware platform bears risks for business operations [165].

The amount of MIPS used by the average IT organization is on the rise. IT industry analysts estimate that most large organizations utilizing mainframes should expect their systems' CPU resource consumption to increase by 15–20 per cent annually. A Macro 4 project manager, Chris Limberger, explains financial consequences of this increase as follows [3]:

> Each additional MIPS typically costs around GBP 2,500 in hardware and software charges. So if a company running a 10,000 MIPS system increases capacity by as little as ten per cent per annum, the incremental cost will be in the region of GBP 2.5 million. That's pretty typical but if your business is growing and if you're upping the level of activity on your mainframe, you can expect much more.

Despite the fact that the incurred usage costs are substantial monitoring of CPU usage is not implemented in a granular way. So sometimes customers have no idea where the CPU resources are being consumed. In fact, majority of managers (58%) admit that they do not continually monitor consumption of CPU resources [17]. The reason there are so many hardware and software tools available to monitor and modify CPU cycle usage (among other resource users) is due to the demand. Given the substantial cost implications for a business the need for introduction of measures which lead to reduction of MIPS utilization is indispensable.

3.1.1 Targeting source code

Where does the accrual of the amount of MIPS used happen? MIPS usage is directly driven by CPU usage, and CPU usage depends on the applications' code. Inefficient code of the applications is considered to be one major driver for MIPS usage [17]. Therefore by improving performance of the code it is possible to lower CPU resource consumption. For instance, software ran on mainframes typically constitutes management information systems. For this class of software interaction with a database is ubiquitous. At the source code level interaction with relational database engines is typically implemented with SQL, a Structured Query Language, which is a specialist language for data manipulation [71]. As a general rule of thumb most ($\approx 80\%$) performance hampering problems are traceable to inefficient SQL code [18].

Efforts aimed at optimizing software assets' source code are a viable option for limiting CPU resource usage. In fact, code improvement projects have yielded operational cost savings. According to [19] a financial services company identified two lines of code that, once changed, saved 160 MIPS. Given the market price of MIPS the two lines of code contributed to substantial cost reduction. Therefore, from the perspective of cutting operational costs having the capacity to capture inefficiencies occurring in the software portfolio's code is vital. For the MIS class of software code optimizations in the area of database interaction loom as particularly worth extending the efforts.

Since the reality of large organizations is such that mainframe usage monitoring is not implemented in a granular way sometimes customers have no idea where the CPU resources are being consumed. Mainframe customers have plenty of ways to monitor performance of SQL at the application level. IBM manuals describe many potential ways to assess DB2 performance including DB2 Optimizer, SQL EXPLAIN statement, OS RMF and SMF data, OMEGAMON, RUNSTATS, Materialized Query Tables, Automatic Query Rewrite, or Filter Factors [69]. And, these are just IBM facilities. There are also dozens of third party products that measure DB2 performance. These tools certainly help in code optimization initiatives aimed at lowering CPU resource consumption. However, when considering control of CPU resource consumption on a large organizational scale their use is no longer obvious as business poses its own constraints that must be addressed.

3.1.2 Business reality

Our case study involves a large mainframe production environment comprising 246 Cobol systems which propel an organization operating in the financial sector. The underlying source code contains 23,004 Cobol programs with over 19.7 millions of physical lines of code. Approximately 25% of the programs interacts with DB2. The portfolio spans decades with the oldest program dating back to 1967. Some programs we have seen originated in the 70s and nowadays they are still being altered and used in production. The development environment for this portfolio resembles a technology melting pot as it is highly diversified. The code is not only hand-written; at least 5 different code generators, and at least 4 different compilers are used in development and maintenance. These characteristics clearly exhibit that executives deal with a complex IT entity. And, assuring operational risk management in such a context is a non-trivial task.

The portfolio serves millions of clients worldwide in various domains including retail, investment and corporate services. The portfolio is business critical; if a part of the 246 systems goes off-line the business cannot carry on. The applications must meet stringent up-time and performance requirements. Any action that might endanger operational continuity is unacceptable. Alterations to the production environment are naturally possible but are avoided unless they are strictly necessary. Other than operational risk the managers must also bear in mind a large effort involved, for instance, in testing, validating and releasing. And, while cost cutting is of high importance the requirement for properly functioning software takes precedence.

Constraints Extending efforts to enable portfolio-wide control of CPU resource usage must be fit into the reality in which this business operates. In this context we faced a number of constraints of which two were essential for our choices. One being of a contractual nature. The other concerning operational risk.

The IT-portfolio is maintained by a third party. As a result the organization did not have direct access to the mainframe itself. Gaining access to the machines turned out to be far from trivial under the existing outsourcing agreement. Only a small group of dedicated people from the contractor side had access. Such setup was put in place to allow the contractor almost unrestricted control over the mainframe and enable fulfilling strict conditions stipulated by the service level agreements. In these circumstances we were able to obtain off-line access to mainframe usage reports and source code. Particularly, we had data on the MSU consumption in the IMS-DB2 production environment, and the source code, that is it.

Moreover, in this particular portfolio small time delays potentially can have a large impact on the continuity of business operations. In the 246 systems hard coded abnormal terminations of transactions (so called ABENDs) were implemented if certain database operations took too long to operate, like an AICA ABEND. These hard coded boundaries were necessary to prevent queuing of old and no longer necessary queries from offices around the world. Typically, the user behavior of the organization's personnel was to stop with user interaction while leaving some transactions unfinished. Such behavior would leave a hanging transaction in queue. To avoid these transactions from stacking up queues and slowing down or even halting the systems hard coded upper limits for response time were implemented. Within this company a very little adaptation of the system time immediately created havoc and led many transactions to be canceled. On one occasion when an engineer set the system time slightly back because of detected deviations between real and system time the hard coded resets fired. Since applying profiling tools might have influence on performance it can also trigger these hard coded resets erroneously. Not a single IT-executive within the firm wanted to take such risks given the significant problems the system time adjustment incident caused.

Naturally, to improve CPU resource consumption one must resort to conducting some sort of measurements. One possibility is to instrument source code with debugging lines to enable, for instance, recording the execution times of particular actions. And, use the obtained measurements to determine which code fragments are likely to be CPU intensive. Obviously, such an endeavor could work if we dealt with several programs still in development but it becomes completely unrealistic for an operational portfolio of 246 systems.

By doing so we would have been taking an unknown risk for the production environment. Furthermore, there is a prohibitively high cost involved. In [94] the authors discuss the cost realities of large scale software modifications. Simple single-site releases of business-critical systems easily cost 20 person days, which amounts to $20,000$ USD when you take a daily fully burdened rate of 1000 USD. So a release of the 246 systems portfolio, let alone any code changes, can cost $4,920,000$ USD ($246 \cdot 20 \cdot 1,000 = 4,920,000$). Clearly such costs are intolerable to executives especially when considering a cost-reduction project at low-risk.

In our research we had the opportunity to analyze a portfolio of 246 applications on top of which a large financial institution operates. Changing the code was an evolutionary process: as soon as one of the 246 systems was due for maintenance, also the performance issues were taken into account. Exceptions were candidates that posed serious performance issues. The idea of automated changes and a big bang of recompiling, installing and testing 246 systems simultaneously is hugely expensive and risky whereas our approach was low-risk, low-impact and evolutionary.

Summarizing, in our setting any approach that influences the performance of the applications was out of the question. This is not only due to the hard coded ABENDs that are present but also due to the effort relating to monitoring the execution behavior in such a large IT entity. Clearly, usage patterns of the 246 systems change and in order to monitor them one would need to deploy profiling tools or enable detailed logging (e.g. collecting specialized SMF records). Such actions might contribute to lowering performance and increase usage costs as a result of extra CPU load. So observing the behavior of systems using standard means bears the risk of not being easily able to cover the entire portfolio.

3.1.3 A light-weight approach

In this chapter we present an approach to dealing with reduction of costs attributed to CPU resource usage. Our approach is suited for deployment in circumstances where low risk and low cost are required. Our approach allows obtaining insight into the portfolio by pinpointing source code worth scrutiny from the perspective of CPU usage. It gives executives an option to plan a source code improvement project without having to engage many resources and take unnecessary risks. For instance, in the investigated industrial portfolio we found a relatively small number of source files which were likely to be responsible for higher than necessary CPU resource consumption. In fact, these programs constituted approximately 0.5% of all the programs in the portfolio. And, we found these in a single day by combining best practices for DB2, static code analyses, and historical mainframe usage data.

As it turned out our light-weight approach pinpointed the same hot spots for SQL improvements which were identified in an earlier small-scale pilot SQL-tuning project. The pilot encompassed some selected applications supporting operations of a single business unit. It was executed by an expert team specializing in code performance improvements. In this pilot profiling tools were used to determine inefficient code. The identified SQL code was altered and progressed to the production environment. The longitudinal data showed reduction of 9.8% in annual MIPS related costs for the optimized part of the portfolio. Due to the sensitive nature of the data we dealt with we cannot provide any

monetary figures that characterize cost savings resulting from this project, or any other costs involved in operating the studied mainframe. According to the management the value of the estimated savings significantly outweighs the cost of the project.

Since the pilot showed large cost reduction potential it was decided to scale up the SQL related performance improvements to the portfolio level. Executives were convinced that expanding the approach used in the pilot on a much larger portion of the portfolio could imperil the business operations. Therefore, we designed our approach. Our proposition incorporates heuristics for possibly inefficient SQL expressions. We discuss the set of heuristics we proposed and present the results of applying these across the entire portfolio. Moreover, the input from the small-scale SQL-tuning pilot gave us details concerning the actual code changes and their effect on the CPU consumption. We discuss the inefficient code identified by the expert team and the improvements they introduced. We present the results of applying our approach to a much larger portion of the portfolio which span across multiple business units. We analyzed two scenarios for code improvements that were determined on the basis of our approach. Based on the data from the small-scale SQL-tuning pilot we estimated the possible effects on the CPU consumption for both scenarios. Our approach turned out to be fully transparent for business managers; even those with no deep IT background. After writing this chapter we found out that our proposition is being applied by others in the industry [40].

In our approach we rely on finding opportunities for improving applications' source code so that CPU usage can be decreased. CPU usage is a major component of mainframe costs, however, it is not the only component. IT savings are only real if CPU cost reduction techniques are not eclipsed by increases in other resource consumers such as disk space, I/O, journaling, logging, back-ups and even human activities. While CPU cycles are expensive so are data storage costs. Reducing one while increasing the other can be counterproductive. For example, suppose you reduce the CPU cycles by eliminating `ORDER BY` clause in some SQL query and adding an index. This requires physical I/Os and disk storage. Whether such operation will result in net savings for IT depends on modification decisions made in each particular cases. Therefore, in any code optimization project based on our approach it is crucial to consider the financial implications that code alterations may have on the overall IT costs.

3.1.4 Related work

Research devoted to control of CPU resource usage in mainframe environments is essentially conducted in the commercial sector. In [19] two factors that impact CPU resource consumption are given: the inefficient applications' code and recurring applications' failures. We use these findings to steer our work and therefore in this chapter we concentrate on an approach dealing with code monitoring. Many industry surveys and guidelines are available, for instance, in [17, 18, 19, 37], and provide valuable insights into CPU resource consumption issues relating to DB2. We incorporate this knowledge into the approach we propose and, in addition, we share the insights obtained from analysis of the case study.

Meeting stringent performance goals is inherent for industry and therefore in our research we focus on delivery of an approach which is implementable in a real-world industrial setting. In [177] the author presents a solution to the problem of time-optimized

database queries execution. That university project reached the industry in the form of a commercially distributed database product, known as MonetDB, to enable delivery of IT-systems with improved database response time. Their research was geared with speed, ours deals with costs. We address the omnipresent IT management issue of controlling MIPS attributed costs and deliver executives with adequate managerial tooling. Similarly as in [161] we also investigate an industrial portfolio. Our approach is, in fact, fitted into the realities of large organizations. Not only was it developed on top of an exceptionally large case study but also results were presented and discussed with the executives.

Application of source code analysis to extract information is omnipresent in the literature. In [94] the authors show how source code analysis supports reduction of costs in IT transformation projects. Literature provides examples of its application in supporting recovery of software architecture [7] or migration of the IT-portfolio to a service oriented architecture (SOA) model [46]. There are also numerous instances of automated software modifications [95, 163, 162] aided with code analysis. In our work we also rely on source code analysis to extract information relevant from the MIPS control perspective. In that aspect our approach is similar to a technique for rapid-system understanding presented in [160]. It turned out that sophisticated parser technology is not necessary to reach our goals. A lexical approach to analysis is accurate enough for our purposes.

3.1.5 Organization of this chapter

This chapter is organized as follows: in Section 3.2 we present CPU usage realities of the mainframe production environment which we used as our case study. We provide fact-based argumentation behind focusing on the improvements in the area of DB2. In Section 3.3 we embark on the problem of inefficient usage of SQL language in the source code of the applications. We introduce an extraction method for locating potentially inefficient DB2 related code. In Section 3.4 we show how we identify Cobol modules which host the interesting, from the analysis point of view, code fragments. In Section 3.5 we discuss the MIPS-reduction project which was carried out by a DB2 expert team and involved SQL code tuning in the IT-portfolio we used as our case study. In Section 3.6 we present our approach in the setting of the entire IT-portfolio. We apply it to the case study and show how to estimate savings from DB2 code improvements. In Section 3.7 we examine the practical issues relating to the implementation of our approach within an organization. In Section 3.8 we discuss our work in the context of vendor management and redundancies in mainframe usage. Finally, in Section 3.9 we conclude our work and summarize findings.

3.2 MIPS: cost component

Mainframe usage fees constitute a significant component in the overall cost of ownership. The fees are directly linked to the application workloads deployed on the mainframe. Measurements of the mainframe usage costs typically involve two terms: MIPS and MSUs. Although mainframe usage measures are colloquially called MIPS, and often used as a rule of thumb for cost estimation, the actual measure is expressed by means of

MSUs. The two measures function in parallel on the mainframe market and constitute input for mainframe usage pricing models.

In this section we first explain the two measures. Next, we discuss transactions as they constitute the prime subject of our analyses. And finally, we embark on the analysis of the MSU consumption for the IMS production environment in which the MIS applications of the studied IT-portfolio were deployed. We present our findings concerning DB2 usage and MSU consumption. We argue that improvements in the interaction between client applications and DB2 have the potential to yield savings for the organizations. We also emphasize that DB2 related source code alterations are characterized with low-cost and low-risk for the business.

3.2.1 MIPS and MSU

MIPS was originally used to describe speed of a computer's processor [91, p. 136]. Since MIPS are dependent on the CPU architecture they are hardly useful in comparing the speed of two different CPUs. For instance, multiplication of two numbers takes a different number of CPU instructions when performed on particular mainframe and PC processors. For this reason some computer engineers jocularly dubbed MIPS a *misleading indicators of performance* [70, 105]. Despite the fact that nowadays Misleading Indicator of Performance are somewhat arbitrary figures they still find their application in the industry since they play a role in determining usage fees.

MIPS is a measure of processor speed alone and for this reason it has come under fire for its inaccuracy as a measure of how well a system performs. How software executes within a mainframe depends not only on the CPU but also on other factors, such as memory usage or I/O bandwidth. To embrace these extra factors IBM began licensing its software according to MSUs. MSU stands for Million Service Units and expresses the amount of processing work a computer performs in an hour which is measured in millions of z/OS service units [91, p. 136], where z/OS is the operating system on IBM mainframes. MSU is a synthetic metric which superseded MIPS for its accuracy as it embraced aspects such as hardware configuration, memory usage, I/O bandwidth, complexity of the instruction set, etc.

MIPS and MSUs are not independent from each other. Originally, when MSUs were introduced they were comparable to MIPS. One MSU was approximately 6 MIPS [91, p. 136]. This relationship has disappeared over time and nowadays MSUs hardly track consistent with the MIPS. In fact, if they did there would be no real need for them.

The fact that MIPS is, in principle, a CPU speed measure has led to confusion over its use as a measure for mainframe usage. A CA senior vice president, Mark Combs, explains it this way [42]:

> MIPS can't measure the actual consumption of work, while MSUs can. MIPS are also capacity based, meaning that users who pay according to MIPS are often paying for capacity they don't need. With MSUs, users can choose capacity- or consumption-based pricing. Shops that run close to 100% utilization most of the time might go with capacity-based pricing, while those

who run only at 40% most of the time would go with consumption based to save money.

Regardless of the chosen pricing model both MIPS and MSUs have a direct financial implication. For the purpose of our study we rely on the fact that either increase in MIPS capacity or accrual of MSUs results in the growth of mainframe usage fees which businesses have to include in their operational costs. In our chapter we assume the use of the consumption-based charging model and interpret the available MSU figures as a measure of consumed MSUs. We will consider reduction in MSUs consumed by a particular mainframe executed object as equivalent to the reduction of mainframe usage costs.

3.2.2 Transactions

A software portfolio is typically partitioned over a number of information systems. Each system implements some functionality which is deemed necessary to support some business operation. To illustrate this, let us consider an IT-portfolio supporting operations of a mobile network provider. Let one of the IT supported business operations be registration of the duration of a phone call made by a particular subscriber. Additionally, let us assume that for implementation of this operation two systems are needed: one providing functionality to handle client data, and another one enabling interaction with the technical layer of the mobile network infrastructure such that calls can be reported. Implementation of a call registration operation involves a number of atomic computer operations each of which is accomplished through functionality provided by one of the information systems available in the portfolio. One would refer to such a bundle of computer operations serving a particular purpose, which is implemented through the available IT infrastructure, as a transaction.

In the portfolio we investigated we dealt with 246 Cobol systems. From discussion with the experts it became known to us that IT-systems in the portfolio follow the SOA model. In a portfolio which adheres to the SOA model certain systems are meant to provide functionality in a form of shared services. These services are then used by other systems in order to implement a specific operation, such as a transaction. On mainframes transactions can be instantiated through, so called, IMS transactions. IMS stands for Information Management System and is both a transaction manager and a database manager for z/OS [33]. The system is manufactured by IBM and has been used in the industry since 1969. Mainframe usage resulting from code execution is reported through MSUs. In order to keep track of the processor usage for each execution of a transaction the number of consumed MSUs is measured and reported. A collection of these reports provides a repository which enables embracing the overall MSU consumption incurred by the IMS transactions in the IT-portfolio. We used this pool of data as one of the essential inputs in our analyses.

3.2.3 Database impact

The database is one of the components of the environment in which information systems operate. For mainframe environments the most frequently encountered database engine

is DB2. DB2 is IBM's software product which belongs to a family of relational database management systems. DB2 frequently refers to the DB2 Enterprise Server Edition which in the mainframe environments typically runs on z/OS servers. Although DB2 was initially introduced for mainframes [33], it has gained wider popularity since its implementation also exist for personal computers (Express-C edition) [56].

In the context of MIS systems most written programs are client applications for DB2. The SQL language is the common medium to access the database. Interaction with DB2 is, in particular, part of the IMS transactions. This is due to one of the characteristics of IMS transactions. They aggregate computer operations to accomplish some complex task. And, a DB2 operation is one of the many possible operations performed on the mainframes. DB2 utilization is known for being a resource intensive operation. Given that the use of CPU cycles has effect on the associated mainframe usage fees the SQL code run on the mainframes should be, in principle, optimized towards CPU cycles utilization.

MSU consumption In order to get insight into how the database usage participates in the MSU consumption in the studied IMS production environment we analyzed the available mainframe usage reports. We had at our disposal characteristics concerning the top 100 most executed IMS transactions. The characteristics formed time series in which observations were measured on a weekly basis for each top-ranking transaction. For each reported transaction the following data was available: the total number of executions (volume), the total number of consumed MSUs and database calls made. The time series covered a consecutive period of 37 weeks.

For our analysis we distinguished two groups of IMS transactions: those which trigger calls to the database and those which do not. We carried out a comparison of the average cost of executing the IMS transactions belonging to the two groups. We expressed the cost of execution as the average number of MSUs consumed per transaction measured on a weekly basis. In order to make a clear cut between the transactions which make database calls and those which do not we used the numbers of database calls reported for each transaction. We then computed the average weekly ratios of MSUs to transaction volumes for both groups. This way we formed two time series which we analyzed.

In Figure 3.1 we present two plots of the time series. The horizontal axis is used to express time in weeks. Each tick indicates a week number. The vertical axis is used to present the ratios of the average weekly MSUs to transaction volumes. The range of values covered by the vertical axis is restricted to the values present in both of the time series. The solid line is a plot of the time series constructed of the ratios for IMS transactions which made calls to the database. The dashed line shows the ratios for transactions which did not make calls to the database.

Analysis of the ratios reveals that the average number of MSUs required to execute an IMS transaction differs between the groups. In Figure 3.1 this fact is clearly visible by the relative position of the time series plots. Nearly during the entire time the ratios expressing the average MSU consumption by the transactions not using the database are below the other ratios. Only between weeks 31 and 33 we see that these ratios are above those expressing the average MSU consumption by the database using transactions. We found this case interesting and examined closer the MSU figures for rankings covering the period between weeks 31 and 33. Our analysis revealed one IMS transaction which

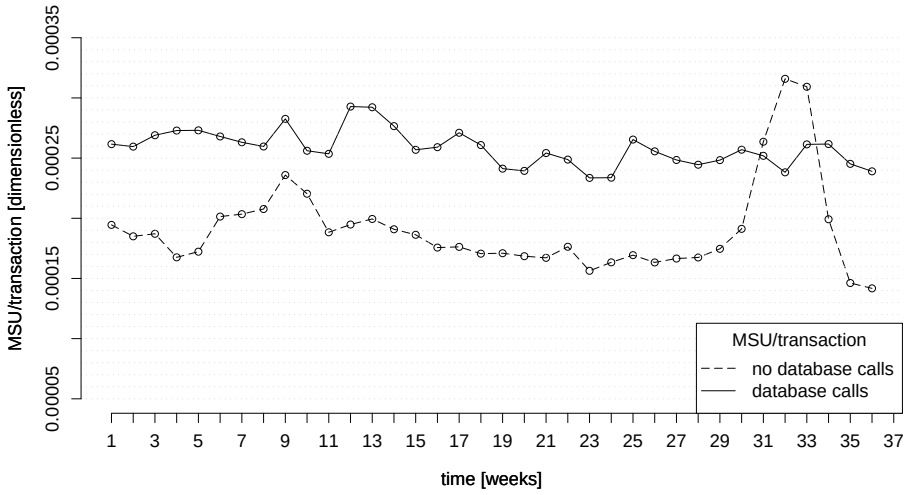

Figure 3.1: Time series of the average weekly ratios of MSUs to transaction volumes for two groups of transactions: those which trigger database calls and those which do not.

had an exceptionally higher than usual MSU consumption reported in those weeks. For this transaction the average weekly number of MSUs consumed in the period from week 1 until 30 was approximately 1299.227. In weeks 31 through 33 the reported consumptions were 5327.130, 8448.990, and 7022.570, respectively. In each case these values were at least four times the average usage between weeks 1 and 30. We did not have enough information to investigate why the temporary peaks occurred. We suspect that one reason could be some non-optimal change to the transaction's implementation and its migration to the production environment. We noticed that the transaction did not occur in the top 100 rankings for the weeks 35 and 36. This might suggest its removal from the production environment.

We compared how the average weekly MSU consumption by the IMS transactions from the two groups differs. We took the long-term average of the ratios for both of the time series. For the transactions which trigger calls to the database the average was 0.0002573. For the transactions which do not, 0.0001914. After we removed the outlier, the transaction which caused the peak between weeks 31 and 33, the average number of MSUs consumed to process an IMS transaction which does not trigger calls to the database came down to 0.000158. These calculations clearly show that on the average the database interacting transactions are more MSU intensive than those which do not interact with the database. Considering the computed averages we observe a difference by nearly as much as 63%.

Database importance The single fact that IMS transactions that perform database calls use more resources than transactions that do not is in itself not a proof that calls to the database are responsible for the major resource usage. While it is likely a more thorough analysis of the proportion of MSUs consumed as a result of executing the Cobol's object code and the MSUs related to DB2 operations within the transactions is necessary. Otherwise, there are a number of other possibilities, for example, it could be the case that the database-involving transactions are simply more complicated than the non-database transactions. Nevertheless, as it turned out the importance of database in the production environment is significant. To investigate this we analyzed the proportion of the database interacting transactions volume in time. Again, we considered the data from the weekly top 100 rankings. For each week we totaled the reported aggregate transaction volumes of those transactions which triggered calls to the database. We then computed the percentages of the total transaction volumes in each week.

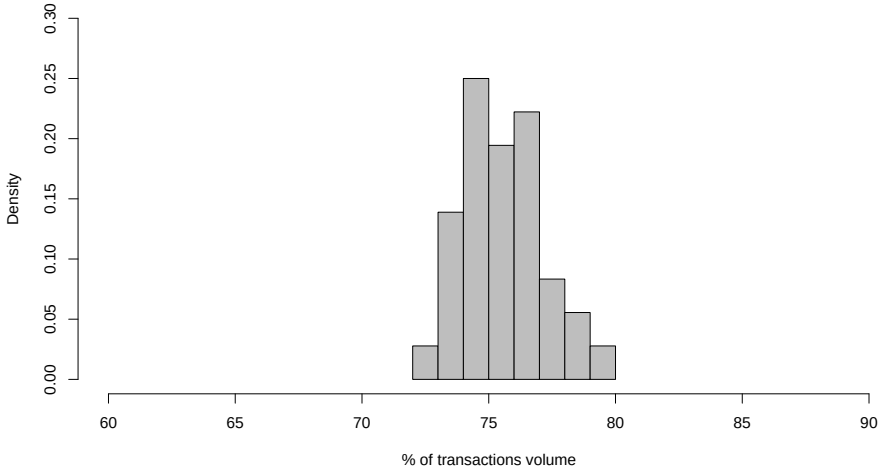

Figure 3.2: Distribution of the percentages of the total weekly aggregated transaction volumes of the database interacting IMS transactions.

In Figure 3.2 we present the distribution of the percentages of the total weekly aggregated transaction volumes of the database interacting IMS transactions. We restricted the range of values presented on the horizontal axis to 60% through 90% in order to clearly present the distributions. We did not find any percentages in the data sample outside this range. As we see in the histogram the bars are concentrated to the middle part of the plot. This clearly exhibits that the transactions interacting with the database occupy the majority of the top executed transactions. A great portion of the core business applications which code we inspected are used to handle customer data which are stored in the database. So, our findings are in line with that.

3.2.4 CPU usage sources

There are all kinds of CPU usage sources. For instance, screen manipulations, algorithms characterized by high computational complexity, expensive database queries, etc. When facing the task of reducing CPU resource consumption any of these elements is a potential candidate for optimization. However, from the perspective of maintenance of a business critical portfolio it is unlikely that an executive is interested in a solution that could imperil operations of a functioning IT-portfolio. It is commonly desired that an approach embodies two properties: low-risk and low-cost. By choosing to improve interaction with DB2 at source code level it is possible to deliver these two properties.

Improvements done in the area of DB2 are low-risk. By low-risk improvements we mean software maintenance actions which do not introduce significant changes to the IT-portfolio. Especially with regard to source code. Our approach does not encourage a major overhaul. As we will show in most cases minor modifications in a few lines of code or configuration changes in the database are sufficient to achieve changes in the MSU consumption. This small scope of alterations is partly due to the fact that DB2 engine provides a wide range of mechanisms which allow for affecting the performance of execution of the arriving database requests. Also, the SQL language allows for semantic equivalence. This opens vast possibilities to seek for other, potentially more efficient, expressions in the source code than the existing code. Due to the fact that database execution performance improvement deals with relatively small changes usually little labor is required. This makes the DB2 related improvements low-cost approach to reducing CPU resource consumption.

Based on the analyzed mainframe usage data we have found evidence that in terms of the average number of MSUs consumed the database interacting IMS transactions cost more than other transactions. Also, these transactions are among those most commonly executed. These observations suggest that by embarking on improvements of the DB2 interacting transactions we address a meaningful cost component on the mainframe.

3.3 DB2 bottlenecks

In this section we focus on communication between DB2 and the client applications. First, we present what factors impact performance of DB2. In particular, we concentrate our attention on the way SQL code is written. Next, we show cases of inefficient SQL constructs and propose a set of source code checking rules. The rules are syntax based and allow for isolation of code fragments which bear the potential to hamper the CPU when processed by the database engine. Finally, we discuss how we implemented the source code checking process to facilitate the analysis.

3.3.1 Performance

Performance of DB2 depends on various factors such as index definitions, access paths, or query structure, to name a few. A DB2 database engine provides administrators and programmers with a number of facilities which allow to influence these factors [64]. However, these facilities require accessing the production environment. When approaching re-

duction of CPU resource consumption from source code perspective it becomes necessary to examine the SQL code. Therefore in our approach we employ DB2 code analysis to seek for possible improvement opportunities.

The way one writes SQL code has a potential to significantly impact performance of execution of requests sent to DB2. This phenomenon is not different from how performance of the execution of programs written in other programming languages is affected by coding style. Virtually any code fragment is inevitably destined to perform inefficiently when inappropriate language constructs or algorithms are used. Writing efficient SQL code requires extensive experience from the programmers, solid knowledge of the language constructs, and also familiarity with the mechanics inside a database engine. In most cases following recommendations of experienced DB2 programmers and fundamental SQL programming guidelines allows obtaining code which runs efficiently. Even though it is the functional code that the consumers are after these days code efficiency cannot be neglected. This is particularly important in the face of growing complexities of queries encountered in, for instance, data warehousing applications.

For business owners inefficient SQL code is highly undesired in the IT-portfolio at least from one perspective; it hampers the speed in which operations are accomplished for the customers. And, of course, in case of mainframe environments it costs money since it wastes CPU resources. Even though static analysis of the programming constructs used in SQL is not sufficient to conclude whether the code is inefficient, it certainly leads to finding code which has the potential of being inefficient. Such code is a good candidate for an in-depth analysis, and if determined as inefficient, for optimization. SQL code optimizations involve, for instance, rewriting, changing the manner in which it is executed, its removal in case it turns out to be redundant, or reconfiguration of the database so that the time needed to execute the query is improved. Application of any of these depends on particular instances of the SQL code. The bottom line is that these instances must first be found.

3.3.2 Playground

We concentrate on the analysis of the use of the SQL language in the source code of the software applications. SQL facilitates support for various data manipulation tasks. It provides for a whole range of operations. For instance, creation of database objects, maintenance and security, or manipulation of data within the objects. SQL's syntax is based on statements. The statements are commonly categorized according to the type of function they perform. Normally the following three categories of the language are distinguished: Data Definition Language (DDL), Data Manipulation Language (DML), and Data Control Language (DCL) [117, p. 18]. Statements classified as DDL allow to create, modify or delete database objects. The DML statements allow for manipulation of the data within objects through operations such as insertion, deletion, update or retrieval. The last subset, DCL, allows controlling who, among the database users, has rights to perform specific operations.

Apart from those major three subsets there also exist auxiliary statements which do not fall into any of the above categories. An example of such a statement is the `SET` statement of DB2 which assigns variables with values. Although this statement is typically used

inside a body of an SQL's stored procedure it is also encountered used independently. In such cases it typically serves as a means to copy values of the database registers into a program's local variables, also known in the context of embedded SQL as host variables.

The division of SQL into subclasses provides a way to map specific subsets of statements to programmers' tasks. For instance, a programmer writing client applications is very likely to limit the scope of SQL statements to the DML group. From the perspective of business applications, which by their nature are clients of the database, the DML subset constitutes the great majority of SQL's vocabulary used in the programs. In fact, the internal documentation for SQL coding standards, which belongs to the organization that provided us with the data, states that only usage of the `SELECT`, `INSERT`, `DELETE` and `UPDATE` statements is permitted in the code embedded in Cobol programs.

Statement category	**Statement example**	**Instances**	**Percentages**
DCL	GRANT	0	0.00%
DCL	REVOKE	0	0.00%
DDL	CREATE	0	0.00%
DDL	DROP	0	0.00%
DML	SELECT	14387	68.91%
DML	DELETE	1948	9.33%
DML	INSERT	2147	10.28%
DML	UPDATE	2397	11.48%

Table 3.1: Examples of the types of SQL statements encountered in the IT-portfolio used as a case study.

In Table 3.1 we give examples of the SQL statements for each listed subset and, whenever available, provide numbers of their occurrences we found in the organization's portfolio. In the DML category we list statements which allow querying (`SELECT`), adding (`INSERT`), removing (`DELETE`) or updating (`UPDATE`) the data. Interestingly, the `SELECT` statement is generally claimed to be the most frequently used SQL statement [110]. This claim turns out to be true at least for the source code we studied. Occurrences of the `SELECT` statement account for almost 69% of all DML statements.

In the DDL category we show two examples of statements: `CREATE` and `DROP`. These statements allow for creation and deletion of database objects such as tables, indexes, users, and others, respectively. They hardly ever occur inside the code of client applications since manipulation of database objects is taken care of normally during the database setup process, or on some other occasions. In fact, we found no occurrences of these statements in the IT-portfolio under study.

A similar situation holds for the DCL subset of statements. In Table 3.1 we give examples of two such statements: `GRANT` and `REVOKE`. They are used to grant rights to users to perform specific operations on database objects or to revoke these rights, respectively. Due to the nature of the operations these statements perform they are used occasionally by database administrators.

In our analysis we concentrate on the SQL code which is embedded in the applications' source code. In particular, we primarily focus on investigating the `SELECT`

statements. While it is possible to formulate inefficiency heuristics for other types of statements, the SELECT statement offers a highly desired feature. It is characterized by a complex structure which offers vast capabilities to code queries in numerous ways so that semantic equivalence can be preserved. Obviously, this property is very much sought after from the code optimization perspective. Other than that, the statement is commonly addressed by the DB2 experts community for inefficiency related problems. And, as it turned out, it is the most frequently occurring statement in the studied portfolio. In SQL code analysis we also concentrate on prohibitive use of SQL code in applications. Let us recall that according to the company's proprietary coding standards the embedded SQL is meant to be limited to DML type of statements. Therefore, when formulating characteristics of the possibly inefficient SQL constructs we look also at these deviations.

3.3.3 Potentially inefficient constructs

Given an SQL statement at hand we want to analyze its structure and determine whether it carries any signs of being potentially expensive for DB2 processing. Let us emphasize that on the basis of the SQL code we are only able to detect signs of potential inefficiency. Thorough analysis of the actual environment in which the statement is executed allows to determine whether the statement is indeed running inefficiently. For instance, despite the fact that the operation of sorting datasets is in general deemed to be expensive its execution is likely to go unnoticed when performed on a data set comprising a hundred of records. However, it is expected to take considerable amount of time when performed on a data set with over one million records. Nevertheless, analysis of SQL code delivers information which when considered at the level of an IT-portfolio enables a low-cost and reasonably accurate assessment of the amount of code suitable for improvements.

In order to exemplify how analysis of an SQL's statement code brings us to finding potentially inefficient constructs, let us consider a general coding guideline that recommends avoiding the lack of restriction on the columns which are to be fetched as a result of processing a SELECT statement passed to DB2 [64, p. 215]. On the level of an SQL query's code lack of restriction on the columns is implemented by the use of the $\star$ character in the SELECT clause of the SELECT statement.

Let us now consider two semantically identical programs with SELECT statements where one carries an unrestricted and the other a restricted SELECT clause. Semantic equivalence in SQL is possible since as a high level language it enables specification of relational expressions in syntactically different but semantically equivalent ways. In consequence of this a query that obtains the required data from the database has possibly many forms. As an illustration let us assume that there exists a DB2 table called PEOPLE which consists of 50 columns, among which two are named FNAME and LNAME. Suppose also that the programs require for processing a list of all pairs of the FNAME and LNAME that exist in the table. The following are two possible SQL query candidates which allow fetching the required data.

```
1. SELECT * FROM PEOPLE
```

```
2. SELECT FNAME, LNAME FROM PEOPLE
```

In the first query the `SELECT` clause contains the $\star$ character. Execution of the query results in fetching all rows from the table with all possible columns defined in the table PEOPLE. In a real-life scenario this table possibly contains hundreds of thousands of rows (e.g. a table with names of clients of a health insurance company). In the second query instead of using the $\star$ character the needed column names are listed explicitly between the keywords `SELECT` and `FROM`, as per SQL language convention. In this case the database will only return the specified values. Generally, restricting what columns are to be fetched during query execution allows DB2 to retrieve only the needed data and thus constraining usage of the hardware resources to the necessary demand. In a real-life scenario the second query is expected to consume less resources than the first one. And, in case of execution on a mainframe be more efficient cost-wise.

There are more syntax based signs in the SQL statements which allow classifying them as potentially expensive. We used the organization's proprietary guideline on internal SQL coding standards, IBM's recommendations concerning DB2 SQL queries tuning, experts recommendations, and also our experience in order to identify practices which are known to have a negative effect on DB2 performance [61, 64, 135, 134]. Based on these sources of knowledge we determined a set of syntactic rules which point at potentially inefficient SQL statements. The point of the rules was to have the means to check if the coding recommendations are followed. Namely, for a given SQL statement some recommendation is not followed if we recognize that the statement's code complies with some syntactic rule. For instance, the aforementioned recommendation concerning explicit listing of the column names which are to be fetched by a query is violated if in the code of the corresponding `SELECT` statement there exists a $\star$ character in the `SELECT` clause.

Each syntactic rule is associated with a programming construct. For each such programming construct we assigned an identifier in order to allow for simple reference throughout the chapter. We first summarize the selected programming constructs in Table 3.2, and then treat them in greater detail.

Table 3.2 presents the list of SQL programming constructs which potentially lead to inefficient use of hardware resources when sent to DB2 for processing. In the first column we list programming constructs identifiers. In the second column we explain how each selected programming construct is detectable at the code level of an SQL statement.

The first two programming constructs listed are AGGF and AGGF_2. Both of them are related to the presence of aggregate functions in an SQL query. In SQL, aggregate functions constitute a special category of functions that return a single value which is calculated from values present in a selected table column. SQL provides a number of those functions. The actual set differs depending on the version of the database engine. For instance, the SQL reference manual for DB2 9 for z/OS lists the following functions as aggregate: `AVG`, `COUNT`, `COUNT_BIG`, `COVARIANCE`, `COVARIANCE_SAMP`, `MAX`, `MIN`, `STDDEV`, `STDDEV_SAMP`, `SUM`, `VARIANCE`, `VARIANCE_SAMP` and `XMLAGG` [62, p. 252]. Previous versions of DB2 support smaller subsets of these functions [71, 66]. Evaluation of a query containing an aggregate function is deemed costly if the aggregate function is not used in a manner which enables DB2 to carry out processing efficiently. For most of the functions there exist conditions that must be satisfied to allow for evaluation

Construct ID	Meaning
AGGF	Aggregate function present in a query. Under the AGGF identifier all aggregate functions are included except for `STDDEV`, `STDDEV_SAMP`, `VARIANCE`, and `VARIANCE_SAMP`.
$AGGF_2$	SQL query contains one of the following aggregate functions: `STDDEV`, `STDDEV_SAMP`, `VARIANCE`, and `VARIANCE_SAMP`.
COBF	SQL statement is used to load to a host variable a value obtained from one of the database's special registers.
DIST	The `DISTINCT` operator is present in a query.
GROUP	Query contains `GROUP BY` operator.
$JOIN_x$	Join operation present in a query. The subscript x is used to indicate the number of tables joined.
NIN	`NOT IN` construction applied to a sub-query.
NWHR	Missing `WHERE` clause in a query.
ORDER	Query contains `ORDER BY` operator.
UNION	`UNION` operator is used.
UNSEL	No restriction on the column names in the `SELECT` clause of a `SELECT` statement.
$WHRE_x$	`WHERE` clause contains a predicate which contains host variables and constants only. The x provides the total number of such predicates in the `WHERE` clause.
$WHRH_x$	`WHERE` clause contains a predicate which contains host variables and column names. The x provides the total number of such predicates in the `WHERE` clause.

Table 3.2: Programming constructs potentially leading to inefficient use of hardware resources during DB2 processing.

with minimal processing overhead [61, p. 753]. Based on the SQL manual we split the aggregate functions into two groups, namely those functions for which there exist conditions which when satisfied enable DB2 to evaluate efficiently and those for which evaluation is costly regardless of the way the aggregate functions are used. The latter group is formed by the four functions: `STDDEV`, `STDDEV_SAMP`, `VARIANCE`, `VARIANCE_SAMP`, and identified by $AGGF_2$. For the remaining functions, identified by AGGF, there exist conditions under which cost-effective evaluation is feasible. Since verification on the syntax level whether the conditions are met requires not only full parsing of a query but also additional information on the tables and defined indexes for our purposes we restrict code analysis only to reporting existence of an aggregate function in a query.

Another programming construct we list in Table 3.2 is COBF. COBF refers to the redundant use of SQL in Cobol programs. By redundant use we mean those inclusions of the SQL code which are easily replaceable by semantically equivalent sets of instructions written in Cobol. An example of it is a situation when SQL code is used to access the DB2's special registers with the pure intention of copying their values into host

variables of the Cobol program. DB2 provides storage areas, referred to as special registers, which are defined for an application process by the database manager. These registers are used to store various information, such as current time, date, timezone, etc., which can be referenced inside SQL statements [66, p. 101]. Four of these special registers involve information which is retrievable through an invocation of the built-in, or also referred to as intrinsic, Cobol function called `CURRENT-DATE`. For the following special registers `CURRENT DATE`, `CURRENT TIME`, `CURRENT TIMESTAMP` and `CURRENT TIMEZONE` the stored values are retrievable through that function. Let us recall that in the studied portfolio occurrences of non-DML statements were not permitted. From the CPU resource usage perspective while it is not clear that replacing any SQL code by semantically equivalent Cobol code guarantees reduction in CPU resources usage it is still worth while considering such code alteration. Especially, in cases when relatively non-complex operations, such as current date retrieval, are performed. Therefore we chose to include COBF construct on the list of possibly inefficient programming constructs.

One of the main CPU intensive database operations is sorting. Sorting is a common operation in data processing. SQL provides a number of programming constructs designed to be used as part of the `SELECT` statement which increase the probability that the database will perform sorting at some stage of query's execution. An obvious candidate is the `ORDER BY` clause which literally tells DB2 to perform sorting of the result set according to columns specified as parameters. Therefore detection of the presence of the `ORDER BY` clause is an immediate signal that there might exist a possibly high load on the CPU. Typically, elimination of sorting, if possible, is attainable through adequate use of indexes. Although, this move is likely to reduce load on the CPU it does affect data storage costs which after all might eclipse savings resulting from MSU consumption. This fact should be kept in mind when introducing code changes. Whereas the `ORDER BY` construct is the explicit way to tell the database that sorting is required there are also other constructs which implicitly increase the likelihood of sorting. These constructs are: `UNION`, `DISTINCT`, `GROUP BY` and the join operation [135, 134, 64]. In Table 3.2 they are labeled by UNION, DIST, GROUP, and JOIN$_x$, respectively. The `UNION` construct is a set sum operator. Whenever DB2 makes a union of two result sets it must eliminate duplicates. This is where sorting may occur. Similarly, when the `DISTINCT` construct is present elimination of duplicates takes place. In fact, presence of the `SELECT DISTINCT` construct typically suggests that the query was not written in an optimal manner. This is because the database is told that after the rows have been fetched duplicate rows must be removed. According to [64, p. 584] the sort operation is also possible for the `GROUP BY` clause, if the join operation is present, or the `WHERE` clause contains a `NOT IN` predicate. We report any occurrences of the sorting related constructs encountered in queries. In case of the join operation we additionally report the total number of tables involved. The number serves as the subscript in the JOIN$_x$ identifier.

The `WHERE` clause is a very important component of the `SELECT` statement. In principle, each SQL query should be restricted by the `WHERE` clause to limit the number of rows in the result set. For this reason we check for the presence of this clause and label any SQL query with NWHR in case the `WHERE` clause is missing. Whenever the `WHERE`

clause is used it is important that the predicates it contains are structured and used in an manner which enables DB2 to evaluate them efficiently. There are a number of conditions which govern the structuring of the content of the WHERE clause. Most of them require an in-depth analysis of how the predicates are used and structured inside the WHERE clause, the configuration of the database, and also require considerations as to the data to which the query is applied to.

We focus on several checks in the WHERE clause which already give an indication of the potential overhead. One commonly addressed issue is the ordering and the use of indexes for predicates which contain host variables. In [135, 134] the interested reader will find details concerning this subject. In order to capture the opportunity for improvements in that respect we check the WHERE clauses for the presence of predicates which contain host variables and each time the total number of those predicates exceeds one we report it with the WHRH$_x$ identifier. The subscript is used to specify the number of occurrences of such predicates in the query.

Similarly as COBF, WHRE$_x$ refers to potentially redundant invocations of SQL statements. This time we look at those predicates in the WHERE clause for which evaluation does not involve references to the database. In order to explain this let us consider the following SQL query embedded inside a Cobol program:

```
1 EXEC SQL
2     SELECT c1, c2, c3
3     FROM table1
4     WHERE :HOST-VAR='Andy'
5         AND c1>100
6 END-EXEC
```

It is a simple SQL query which returns a set of rows with three columns: c1, c2, and c3. The results are obtained from table1. The constraint imposed on the result set is given in the WHERE clause through a conjunction of two predicates: :HOST-VAR='Andy' and c1>100. Obviously, the only situation in which this conjunction is true is when both predicates evaluate to true. From the structure of the first predicate we see that its valuation has nothing to do with the values in the columns of the table1. The host variable :HOST-VAR is tested for equality against a constant 'Andy'. Only the value of the second predicate depends on the table's content. In this particular case it is possible to avoid invocation of the SQL query by rewriting the existing query and changing the way in which it is embedded inside a program. The following is a semantically equivalent improvement:

```
1 IF HOST-VAR = 'Andy' THEN
2     EXEC SQL
3         SELECT c1, c2, c3
4         FROM table1
5         WHERE c1>100
6     END-EXEC
7 END-IF
```

In the rewritten code fragment the WHERE clause from the original query was truncated by removing the :HOST-VAR='Andy' predicate. Also, the new query was wrapped into the Cobol's IF statement. The removed predicate from the SQL query was used as the predicate for the IF statement. In the new situation the query is passed to DB2 for pro-

cessing only when the Cobol's `IF` statement predicate evaluates to true. Of course, in the presented example the improvement was relatively simple. In real-world SQL statements predicates such as `:HOST-VAR='Andy'` may occur in more complex `WHERE` clauses and their improvements be more sophisticated. Nevertheless, the discussed constructs in the `WHERE` clauses ought to be avoided. The number of occurrences of such constructs within the `WHERE` clause are reported in the subscript of the WHRE_x identifier.

In Table 3.2, we also listed the UNSEL construct which corresponds to the use of unrestricted `SELECT` clause. This construct follows the general recommendation found in [64, p. 215] and was explained earlier in the chapter.

3.3.4 Getting the data

In order to conduct analysis of the SQL code embedded in the source code of a large software portfolio an automated process is a necessity. Let us recall that we did not have access to the mainframe and therefore we could not easily rely on the facilities offered by z/OS to carry out our analysis. Due to the fact that the source code was made available to us on a unix machine we considered the use of tools provided by that platform.

To detect whether an SQL statement contains any of the potentially inefficient SQL constructs listed in Table 3.2 we must examine the SQL code. To accomplish this task there are essentially two venues to consider: parsing or lexical scanning. The parsing option embodies SQL parser selection, SQL statements parsing, and slicing through the resulting objects to check whether the sought after SQL constructs are present. With lexical scanning it is necessary to construct regular expressions capable of matching the desired constructs within an SQL statement. Due to the fact that the SQL constructs listed in Table 3.2 followed simple textual patterns it was sufficient to employ regular expressions to detect them within SQL statements. To do this we required a technology that is suited to process efficiently a large volume of source files. Let us recall that we dealt with a software portfolio of a large size (23,004 Cobol programs that implement 246 applications). Therefore we opted for incorporating *Perl*. *Perl* is a programming language (and a tool) primarily meant for text processing [172, 173]. In particular, it is well suited for a light-weight analysis of the source code. It performs well when processing sheer amounts of data. *Perl* contains a strong regular expressions processing engine and therefore it is adequate to accomplish our tasks. In this way we allowed for an inexpensive, highly scalable, and simple solution to facilitate our analysis.

Tooling In order to accomplish SQL code analysis we distinguish two phases. First, isolation of the SQL code from the applications' sources. Second, verification of the extracted SQL statements against the presence of the programming constructs listed in Table 3.2. To facilitate these phases we developed a toolset comprising two *Perl* scripts. To illustrate the low complexity of our toolset in Table 3.3 we provide some of its key characteristics.

In Table 3.3 we provide characteristics of the developed toolset used to examine the SQL code. In the first column we list the scripts by filename and in the second we explain their role. In columns 3 and 4 we provide the total number of lines of code and the estimated time it took to develop them, respectively. Clearly the 470 lines of code represent

Script	Role	LOC	Development time
sql-explorer.pl	Used to scan the contents of the Cobol programs and extract the embedded SQL code.	121	4 hours
sql-analyzer.pl	Used to analyze the extracted SQL code. Particularly, to determine the presence of the potentially inefficient SQL constructs listed in Table 3.2.	349	8 hours
Total		470	12 hours

Table 3.3: Characteristics of the developed toolset used to examine the SQL code.

a relatively small amount of *Perl* code. The scripts were delivered in as little as 12 hours. That included design, implementation and testing. Our toolset sufficed to obtain the data required for our analysis.

Embedded SQL In case of Cobol programming language a DB2 program is characterized by the presence of `EXEC SQL` blocks in the code [63][p. 245–248]. Each SQL statement must begin with `EXEC SQL` and end with `END-EXEC`. We will refer to any Cobol module containing the `EXEC SQL` block as DB2 module. The following is a code fragment taken from one of the Cobol programs from the portfolio we study in this chapter.

```
1 00415 *    GET NUMBER OF ROWS
2 00416 EXEC SQL
3 00417   SELECT  COUNT(*)
4 00418     INTO  :USERREC-COUNT
5 00419     FROM  USERSBATCHST
6 00420 END-EXEC
```

To extract the embedded SQL code it is sufficient to apply lexical scan of the source lines of code. Of course, the goal is to extract the actual content embraced by the `EXEC SQL` and the `END-EXEC` statements. This is not too difficult, however, it takes some steps to separate the wheat from the chaff. Namely, we want to obtain only the SQL code without any additional elements such as SQL comments, Cobol comments, or the Cobol's editor line numbers. For instance, in the above code fragment we see the editor line numbers prefix every line. In order to address these issues we developed a light-weight SQL code extractor for Cobol programs which we deployed for our analysis. Figure 3.3 presents the implementation.

In Figure 3.3 we present the body of a Perl routine we wrote to tackle SQL code extraction from the Cobol sources. The routine takes as input a path to a Cobol file and produces as a result an array of strings where each string holds content of each `EXEC SQL` block found inside the file. Each string represents the extracted SQL expression as a single line of code free from comments. In lines 1–5 initialization of the routine variables

```
1  my(@sql_code)=();
2  my($inside_exec)=0;
3
4  open(FILE, $_[0]) || die "Cannot open file: $_[0]\n";
5  @_=<FILE>;
6  foreach(@_){
7      s/^.{6}\*.*//g;                          # Remove Cobol comments
8      if(/EXEC\s+SQL/) {$inside_exec=1}        # Signal entry to an EXEC SQL block
9      if(/END\-EXEC/) {$inside_exec=0}         # Signal exit from an EXEC SQL block
10     if($inside_exec){                        # Remove SQL comments
11         s/^.{6}\s+\-\-//g;
12     }
13     # Clean up
14     s/^.{6}\s*(.*)$/ $1/g;
15     s/^([^']*?)\s*$/$1/g;
16     s/\n//g;
17 }
18 $_ = join('',@_);
19 @sql_code = /EXEC\s+SQL(.+?)END\-EXEC/g;     # Extract the embedded code
20 foreach(@sql_code){ s/^\s*(.+?)\s*$/$1/g; } # Clean up
21 close(FILE);
```

Figure 3.3: Perl procedure used to extract SQL code from a Cobol module.

is done, the Cobol file is opened, and loaded into an auxiliary variable `@_`. In lines 6–19 the actual extraction process takes place. Each line of the Cobol file is analyzed line after line, which is indicated by the loop statement in line 6. First, Cobol comments are removed (line 7) using Perl's substitution expression. This step boils down to checking for presence of the $\star$ character in the 7th column. If such character is found the line is emptied. Second, SQL comments are dealt with. They occur inside the `EXEC SQL` block and are prefixed by the following sequence `--`. To remove them we first determine whether the line which is being analyzed is inside such a block or not. This is done in lines 8 and 9 where an auxiliary variable `$inside_exec` is set and unset depending on the presence of either `EXEC SQL` or `END-EXEC` keywords in the line, respectively. If the line is inside the `EXEC SQL` block it is checked whether it is an SQL comment and if so it is emptied. This check is done in lines 10–12. Third, since the content of `EXEC SQL` is to be transformed into a single line it is important to remove redundant characters from the line. Therefore for each line the characters from the, so called, editor's column (columns 1–6) are substituted into a single space (line 13) and the end-of-line characters (EOL) are removed (line 14). Once all Cobol lines of code have been analyzed they get concatenated (line 17). The content of the `EXEC SQL` blocks is extracted in line 18 using the Perl's *greedy* array operator. Finally, for each extracted SQL expression the leading and trailing spaces are removed (line 19) and the Cobol file is closed (line 20).

Code preparation After extraction of the SQL code each DB2-Cobol program is associated with a number of SQL statements. For the purpose of analysis of the `SELECT` statement we perform preparatory steps. In a DB2 program, the `SELECT` statement occurs in two contexts: as a plain query (a `SELECT INTO` statement [62, p. 1456]) or inside a mechanism called a *cursor*. For detailed information on cursors the interested reader is

referred to the literature [68, p. 148]. We distinguish the two forms of occurrences since in case of cursors the `SELECT` statement must first be extracted from the cursor declarations before its code is analyzed using our *Perl* scripts [66, p. 496]. The extraction is accomplished by performing a match of the `EXEC SQL`'s content to the following regular expression.

```
^\s*DECLARE\s+[A-Za-z0-9\.\-\_]+\s+(?:CURSOR\s+FOR|CURSOR\s.+?\sFOR)\s(.+)
```

We applied the above regular expression to all SQL statements associated with the analyzed DB2-Cobol programs. This way we isolated the code of the `SELECT` statements whenever it was encountered inside cursor declarations.

Finding the constructs We now present how we implemented the tools for SQL code analysis. We assume that the SQL statements which are passed for analysis adhere to the DB2 SQL syntax specification. In particular, the implementation we present is based on the SQL DB2 dialect version 9.1. In case of the studied portfolio we had certainty that the SQL code is structured correctly since it originated from Cobol sources which were successfully compiled. For the sake of explanation we assume that the statement, which is being analyzed, has been loaded into Perl's `$_` auxiliary variable. The statement is then passed through the following Perl command: `s/'.+?'/''/g`. This command allows to expunge the content of any string constants which may occur in the code of the statement. By doing so we assure that the objects we search are not found inside strings.

With each statement that is taken for analysis we associate a list of labels which are taken from the first column in Table 3.2. With each programming construct we associate a Perl routine which enables verification whenever the given construct is present in the SQL statement. Each time the verification confirms the presence of such a construct its label is appended to the SQL statement's labels list. Eventually, after the analysis is complete a single SQL statement is associated with a list containing potentially many labels of the construct identifiers.

We now treat in detail the implementation of the Perl routines used for finding the programming constructs listed in Table 3.2. Due to a large number of the programming constructs we strove for a compact presentation of the implementation. For this reason we grouped the programming constructs according to the programming template followed in the implementation. We distinguished five groups.

Group	Programming constructs
I	AGGF, AGGF_2, COBF, DIST, GROUP, NIN, ORDER, UNION
II	UNSEL
III	JOIN_x
IV	NWHR
V	WHRE_x, WHRH_x

Table 3.4: Grouping of the programming constructs.

Table 3.4 presents the grouping of the programming constructs. In the first column we enumerate the groups. In the second column for each group we list the programming constructs. We now explain each enumerated group in detail.

Group I In order to identify the presence of the AGGF, $AGGF_2$, COBF, DIST, GROUP, NIN, ORDER, UNION constructs in an SQL statement it is sufficient to search in the statement's code for the corresponding SQL keywords. The entire procedure boils down to matching the given SQL statement against one of the following regular expressions. In all cases, except for COBF, we must assure that the regular expression is matched against an SQL query.

```
1  [AGGF]
2  [\s,\>\<\=](?:AVG|CORRELATION|COUNT|COUNT_BIG|COVARIANCE|↩
       COVARIANCE_SAMP|MAX|MIN|SUM|XMLAGG)[\s\(]
3
4  [AGGF2]
5  [\s,\>\<\=](?:STDDEV|STDDEV_SAMP|VARIANCE|VARIANCE_SAMP)[\s\(]
6
7  [COBF]
8  ^\s*SET\s+\:[A-Za-z0-9\.\-\_]+\s*=\s*.*?CURRENT\s+(?:DATE|TIME|↩
       TIMESTAMP|TIMEZONE).*?$
9
10 [DIST]
11 [\(\s]DISTINCT\(?\s
12
13 [GROUP]
14 \s+GROUP\s+BY\s+
15
16 [NIN]
17 \s+NOT\s+IN\s+\(\s*SELECT\s
18
19 [ORDER]
20 \s+ORDER\s+BY\s+
21
22 [UNION]
23 \s+UNION\s+
```

The above listing is organized as follows. Each construct discussed here is surrounded by `[]` characters. In the line following the identifier of the construct we find the corresponding regular expression. In the regular expressions it is possible to distinguish SQL keywords. In all cases we see that the keywords are surrounded by additional characters. These additional characters are used to specify the context in which the keywords are allowed to occur. Each time we find a match against any of the above listed regular expressions the appropriate label is appended to the SQL statement's labels list.

Group II The UNSEL programming construct concerns the `SELECT` clause of the `SELECT` statement. In order to perform the UNSEL check we first assure that the tested SQL statement is a query. The following is a fragment of a Perl script used to check for the presence of the unrestricted `SELECT` clause.

```
1  my(@select_clauses)=/(?:^\s*|[\(\s])SELECT\s(.+?)\s(?:FROM|INTO)\s/g;
2  foreach my $c (@select_clauses){
```

```
3       if($c =~ /^\s*\*\s*$|[,\.]\s*\*\s*/){
4           # Report UNSEL
5       }
6   }
```

Let us discuss the above code fragment in detail. Our objective is to locate any occurrence of the `SELECT` clause which lacks explicit listing of the columns. This boils down to testing if the `SELECT` clause contains the general column filter, the $\star$ character. For a single `SELECT` statement we must take for analysis not only the main clause but also those `SELECT` clauses which, if defined, occur in sub-queries. This is accomplished in line 1. The, so called, *greedy* search for a string that matches regular expression `(?:^\s*|[\(\s])SELECT\s(.+?)\sFROM\s` is made to isolate contents of all `SELECT` clauses present in the given query. The result of the search, which is a list of contents of the found `SELECT` clauses, is stored in an array called `@select_clauses`. In lines 2–6, the analysis of the found `SELECT` clauses is done. For each element in the `@select_clauses` array a match to a regular expression, given in line 3, is made. This expression matches all possible contexts in which the $\star$ character occurs and indicates a lack of restriction on column names. When a match is found the UNSEL programming construct is reported. The reporting operation is assumed to be implemented between the brackets in lines 3 and 5.

Group III By JOIN_x we indicate the presence of the join operation in an SQL query. The x subscript gives the total number of tables which participate in the join. For the sake of simplicity we only encompass joins which do not use sub-queries to build tables. The procedure of detecting joins is illustrated by the following Perl code fragment.

```
1   my(@from_clauses)=/\s+FROM\s([^\(\)]+?)\s(?:WHERE|ORDER|GROUP|HAVING|↩
        FETCH|$)/g;
2   foreach my $stmt (@from_clauses){
3       $_=$stmt;
4       my(@joins)=/\sJOIN\s|,/g;
5       if(scalar(@joins)>0){
6           # Report joins
7       }
8   }
```

Group IV For NWHR we test if the `WHERE` clause is missing in an SQL query. This test applies to the main query and all potential sub-queries. In order to verify the presence of the `WHERE` clause it is sufficient to count the number of occurrences of the `SELECT` and the `WHERE` keywords. This follows from the fact that each `SELECT` must be associated with at most one `WHERE` clause. In case the number of found `SELECT` keywords differs from the number of found `WHERE` keywords we conclude that some `WHERE` clause must be missing. The following fragment of the Perl code illustrates the process.

```
1   my(@select_stmts)=/(?:\(\s*|\s|^\s*)SELECT\s+/g;
2   my(@where_clauses)=/\s+WHERE\s+/g;
```

```
3
4  if(scalar(@select_stmts)!=scalar(@where_clauses)){
5      # Report NWHR
6  }
```

In line 1 we find all occurrences of the `SELECT` keyword. In line 2 we find all occurrences of the `WHERE` keyword. We compare the number of occurrences of the two keywords in line 4 and if they differ we report an NWHR.

Group V For the WHRE$_x$ and WHRH$_x$ constructs we analyze predicates in the `WHERE` clause. In both cases the x parameter is used to provide the number of instances of objects found in the `WHERE` clause. Since the process of handling the two searches is analogous we present implementation of only one. The following Perl code fragment illustrates the steps taken to find occurrences of the WHRE$_x$ constructs.

```
1  my(@where_clauses)=/\s+WHERE\s(.+?)(?:ORDER|GROUP|HAVING|FETCH|$)/g;
2  foreach my $clause (@where_clauses){
3      $_=$clause;
4      my(@preds)=/[\s(]?(?:[A-Z][A-Za-z0-9\-\_\.]+\s*(?:=|>|<|BETWEEN|↩
           LIKE)\s*\:[A-Z][A-Za-z0-9\-\_\.]+|\:[A-Z][A-Za-z0-9\-\_\.]+\s↩
           *(?:=|>|<)\s*[A-Z][A-Za-z0-9\-\_\.]+)[)\s]?/g;
5      if(scalar(@preds)>=2){
6          # Report WHRH
7      }
8  }
```

We begin with isolation of the contents of the `WHERE` clauses from the body of the `SELECT` statement. In line 1 we perform a greedy match of the query against a regular expression which isolates the contents of the `WHERE` clauses. The resulting contents are stored in the `@where_clauses` array. Contents of each found clause are then matched against a regular expression which detects occurrences of predicates which involve a Cobol host variable and a table column name. The match is restricted to finding =,>,<,`BETWEEN`, and `LIKE` predicates. The predicates that meet criteria specified in the regular expression are stored in the `@preds` array for later counting. If the number of predicates stored in the `@preds` array exceeds 2 the WHRE$_x$ constructs is reported. The number of predicates found becomes the value of x.

We discussed the implementation. It turned out that this is not too difficult except careful programming with regular expressions. Such approach helps in keeping things simple. Later on we will present the results of applying the Perl scripts to the portfolio. Now we continue with presentation of how to identify in the portfolio those Cobol modules which host the relevant code for analysis.

3.4 Reaching for code

In approaching CPU usage reduction from source code perspective we must reach for relevant source files. In this section we elaborate on the demarcation of the applicable

source files given essential information on the IMS environment. First, we will explain how the mapping between the IMS transactions and their implementation is organized. Next, we discuss the process of analyzing Cobol code which leads to the extraction of data required to identify the relevant source files. Finally, we present properties found for the source code associated with the IMS transactions from the studied IT-portfolio.

3.4.1 Implementation of transactions

IMS transactions bound to a particular mainframe environment are typically identified by some name. In the studied IT-portfolio the IMS transactions were identified by the names of Cobol programs which served as starting points in the execution of transactions. The source code of the Cobol program which identifies a transaction commonly represents, however, only a small part of the implementation of the transaction. What is typical for Cobol environments, but also encountered in other programming languages, is that a single program invokes a number of other programs when it is executed. Those relationships among programs are known as call dependencies. For this reason in order to find source modules which implement a given IMS transaction it is essential to carry out a call dependency analysis.

Formally speaking, implementation of an IMS transaction is best represented by a directed graph $T_{\text{IMS}} = G <\text{V,E}>$. In the context of an IMS transaction V denotes the set of nodes which represent Cobol modules. E denotes the set of edges which represent call relations between the Cobol modules. The edges are characterized by a direction since a call relation always originates at one Cobol module and leads to another. We refer to the T_{IMS} graph as a call-graph.

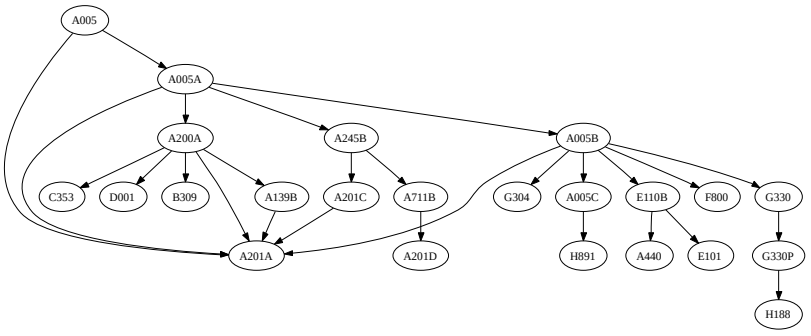

Figure 3.4: Call-graph of an IMS transaction.

Figure 3.4 depicts a call-graph of one of the IMS transactions taken from the portfolio under study. The plot of the graph was prepared by means of the `dot` program which is part of the Graphviz package, an open source graph visualization software [43]. The

data required to feed the `dot` program to construct the graph originated from our analysis of the portfolio's source code. Nodes in the graph represent Cobol programs. The edges of the graph, depicted by means of arrows connecting pairs of different nodes, indicate call dependencies amongst Cobol programs. The direction of the arrow indicates how two modules depend on one another. An arrow originating from node A and reaching node B indicates that a Cobol program (A) contains a reference to a Cobol program (B). In Figure 3.4 the labels conform to a pattern: a letter followed by a number, and an optional letter in the end. The letters in the first position are used to associate Cobol modules with IT-systems which they implement. The remaining characters are used to identify different modules within the systems. In the graph presented in Figure 3.4 we see call-graph of an IMS transaction `A005`. The node `A005` represents the Cobol module which serves as the starting point in the transaction's execution. At runtime executables associated with the Cobol modules represented by the nodes, which appear in the call-graph, have the potential to be executed. Their execution is governed by the program's logic and the flow of data. By analyzing the labels of the nodes in the call-graph it is also apparent that the implementation of the presented IMS transaction spans across 8 different IT-systems.

In a production environment it is not unusual to encounter IMS transactions which involve large numbers of Cobol programs. In Figure 3.5 we present a call-graph of one of the most executed IMS transactions found in the studied production environment. It contains 89 nodes what corresponds to 89 different Cobol programs. The programs are part of 15 IT-systems. Given the large number of nodes and relations in the call-graph in Figure 3.5 it is obvious that it is infeasible to manually analyze the graph. Similarly, as in case of SQL code analysis, in order to determine the relevant Cobol modules we had no choice but to rely on automation.

3.4.2 Portfolio exploration

Reaching for the implementation of the IMS transactions boils down to carrying out source code dependency analysis. In our context we operated at the level of a portfolio of applications, in particular, the IMS transactions. The granularity level at which we explored the dependencies in the source code was limited to the individual source files. Specifically, we were interested in the call relations amongst the Cobol modules.

In dependency analysis two types of dependencies are distinguished: static and dynamic. Static dependencies between two source modules arise when one module contains in its code an explicitly defined call to the other module. Dynamic dependencies arise when a reference to a module is established at runtime. On the code level this typically means that the control-flow is determined on the basis of the value of some variable.

We restricted ourselves to finding those call relations which are retrievable from source code. With the kind of access to the portfolio we had we chose to analyze source code using lexical means. Such approach bears certain limitations. In principle, lexical analysis does not allow for the full exploration of dependencies established at runtime. To do this one would require construction of a control-flow graph for each studied transaction in order to explore all possible paths traversable during execution. Accomplishing this demands, however, access to the environment in which the transactions are deployed. Given the constrains under which we operated we chose to limit the relevant source files

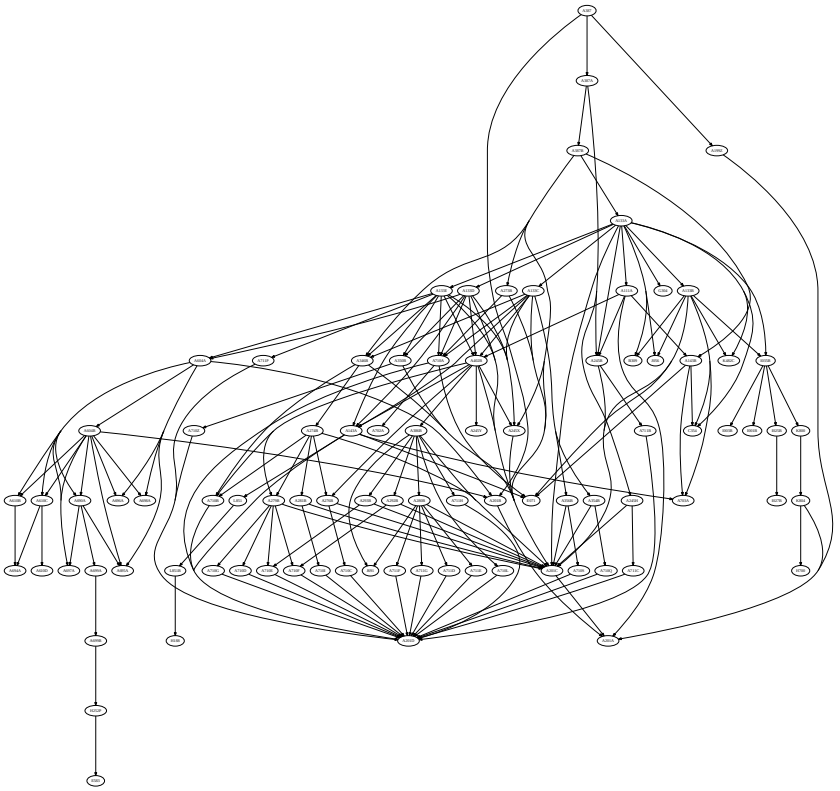

Figure 3.5: Call-graph of one of the most executed IMS transactions found in the studied production environment.

for SQL analysis to those which are retrievable in a lexical manner.

In the studied Cobol sources we distinguish two types of source files: modules and include files. The latter type of files is also known as the copybooks. In our call-dependencies exploration we analyzed the code of the Cobol modules and ignored the copybooks. In the Cobol programming language copybooks are typically used to store data definitions. Their essential role is to allow for simple sharing of the data definitions among various Cobol applications. Given the context of our work we are only interested in locating the source files which contain the embedded SQL. Since it is not anticipated from the copybooks to contain either the executable Cobol statements or the embedded SQL we decided not to involve them in the analysis.

With the taken approach we were able to capture the static code dependencies. Also, we were able to extract certain potential dynamic dependencies. This was possible by

capturing the names of the variables, whenever they were used in the implementation of the calls, and retrieving from the code values of the string constants assigned to them. We treated the values as names of the potentially called modules.

Cobol dependencies In the Cobol programming language implementation of call relations is done by means of two statements: `CALL` and `COPY` [103]. The `CALL` statement transfers control from one object program to another within the run unit [65]. The statement has an elaborate structure that allows for inclusion of various parameters which affect the way the call is made, the result returned, and the exceptions handled. The only mandatory parameter that the `CALL` statement requires is a reference to the program to be called. There are two possibilities to provide this reference, either through a literal or an identifier. In the first case the literal is simply the name of another Cobol program which is embraced by the `'` characters. Cobol calls which are implemented in this way are referred to as *static* since at the compilation time it is known upfront what Cobol programs must be available to run the compiled program. The other way to provide reference to a Cobol program in the `CALL` statement is through an identifier. The identifier holds a name of some variable which is defined in the Cobol program. The variable is expected, at a certain point during execution of the program's code, to hold the name of the Cobol program to be called. This type of `CALL` statement usage implements what is known as a dynamic call.

The other statement, `COPY`, is a library statement that places prewritten text in a Cobol compilation unit [65]. The prewritten text is included in a text file (a copybook). The statement begins with the word `COPY` and ends with a `.` character. In between the `COPY` keyword and the `.` character there is room for parameters. Among the parameters that the `COPY` statement takes there is only one which is mandatory and that is the name of the file to be included. The remaining parameters affect the way in which the statement is processed, in particular, what transformations are done to the text which is included. The effect of processing a `COPY` statement is that the library text associated with the name is copied into the compilation unit, replacing the entire `COPY` statement, beginning with the word `COPY` and ending with a period, inclusive. `COPY` statements are typically used to include into Cobol programs predefined data structures. As explained earlier, we skip copybooks in the process of retrieval of the embedded SQL.

Implementation We approach extraction of call relations on per Cobol module basis. For each scanned module we obtain a set comprising names of the referenced modules. In order to create such a set it is indispensable to analyze the `CALL` statements encountered in the module's code. By extracting the literals from the `CALL` statements we obtain the static call relations. By extracting the identifiers and analyzing assignments of constants to them we obtain names of the modules potentially called. Such retrieval of names provides for approximation of the dynamic call relations. We accomplished the extraction of the call relations from Cobol modules by means of a self-developed Perl script.

Code demarcation The extracted data enabled us to associate each Cobol module with a set containing names of the Cobol modules participating with it in call relations. For the purpose of our work for each selected IMS transaction we are interested in constructing a

set comprising all the modules which form its implementation. By obtaining the set of all call relations for all of the modules in the portfolio we have enough data to analyze the call-graph of any transaction in the portfolio.

Given a specific IMS transaction we begin analysis of its call-graph by finding the Cobol module which implements the starting point for the transaction. Let us refer to this module as the root module. Starting from the root module we explore all the reachable modules on the basis of the call relations. This process is called a graph traversal. Graph traversal is typically done using Breadth-First Search (BFS) or Depth-First Search (DFS) algorithms [20]. Both algorithms begin at some chosen root node and explore all the reachable nodes. As a result they produce a list of visited nodes. We chose DFS since it was best suited for our internal representation of the call relations data. Applying the graph traversal procedure to the main modules of the IMS transactions enabled us to obtain the source files relevant for SQL code analysis.

3.4.3 Code properties

Call dependencies are important when approaching CPU usage reduction through code improvements. Naturally, alterations to the source code of programs which are dependencies for other programs have impact not only on the overall functionality but also the execution performance. The impact that changes to an individual source file have on the performance depends on the frequency of executions of its object code. When considering a call-graph of an application it is clear that improvements done to the code of those modules which object code is frequently executed have higher impact on the performance than alterations made to the modules which object code is executed less frequently. For the purpose of selecting a possibly small set of modules in which code improvements can allow attaining a significant reduction in CPU usage having the information on the number of executions of individual programs is in principle desired. However, obtaining such information requires a meticulous analysis of the environment in which the programs are executed. Given the constraints under which our approach was applied we had to resort to reaching for an alternative information.

In an effort to obtain some approximation to the information on the frequency of program executions we examined call relationships among the source files. This way we were able to get insight into the intensity of reusability of individual programs in the implementations of the applications. We captured the intensity by counting the number of references to the modules. Clearly, by considering the number of references instead of the number of executions we allow for imprecision. For instance, we might deem as a good candidate for optimization a source code module which is referenced by 100 programs each of which only executes its object code one time per day. Nevertheless, with our goal to support CPU usage reduction through code improvements in a large portfolio this information gives us some indication of importance of the module in the execution sequence. And, with the constrains under which our approach is deployed the information is obtainable without much effort.

Source code analysis resulted in each module being associated with a set of names of modules which it references. Having this data at our disposal we carried out two analyses. The first analysis dealt with investigation of the distribution of the number of call relations

of the modules. The second dealt with measuring the relationship between the modules altered and the number of IMS transactions these alterations affect.

Modules fan-in In the first analysis we studied the distribution of the number of references made to modules. In software engineering this number is commonly referred to as fan-in and used as a structural metric. Our analysis of the distribution of the fan-in metric was restricted to those Cobol modules which implement the top 100 most executed IMS transactions. Following the code demarcation process each IMS transaction was associated with a set of modules. Performing the union operation on these sets resulted in isolating 768 distinct Cobol modules. For those modules we determined the fan-in metric. The modules counted as references were limited to the pool of the selected modules. This way we inspected the distribution of the fan-in within a particular group of IMS transactions.

Fan-in [#]	**Frequency [# of modules]**	**Relative frequency**
0	100	0.13021
1	447	0.58203
2	109	0.14193
3	44	0.05729
4	13	0.01693
5	15	0.01953
6..10	20	0.02604
11..15	11	0.01432
16..25	4	0.00521
26..50	1	0.00130
51..55	1	0.00130
56..60	0	0.00000
61..70	1	0.00130
71..76	0	0.00000
77	1	0.00130
78	1	0.00130
$>$78	0	0.00000
Total	768	1.00000

Table 3.5: Distribution of the fan-in metric for the set of Cobol modules implementing the top 100 IMS transactions.

Table 3.5 presents the distribution of the fan-in metric for the set of Cobol modules implementing the top 100 IMS transactions. In the first column we list the classes for the values of the fan-in metric. In the second and third column we provide the frequencies of module occurrences in each class and the relative frequencies, respectively. Out of the 768 Cobol modules the value of the fan-in was greater or equal to 1 for 668 of them. The remaining 100 modules were the starting points for the analyzed IMS transactions with a fan-in value of 0.

The fan-in metric values range from 0 until 78. From Table 3.5 we see that the frequency decays sharply along with the increase in the value of the fan-in metric. We clearly see from the distribution that the vast majority of modules have very few incoming call relations. Those with fan-in metric of value one constitute over 58%. We also find outliers. For the fan-in metric values greater than 26 we find only 5 modules. The top most referenced module has as many as 78 incoming call-relations. The distribution we observe suggests that in the implementation of the top 100 IMS transactions there are only very few modules for which code alterations have impact on a large number of other modules.

Change impact Next to the analysis of the distribution of the fan-in metric among the modules which implement the top 100 IMS transactions we also studied the relationship between the IMS transactions and the modules. Analysis of the fan-in metric revealed to us that in the portfolio we find a group of modules in which each is referenced at least once by some other module. From the CPU resources usage reduction point of view the following question is relevant: how are these modules distributed among the IMS transactions? For each of the 768 modules we counted how many times it occurs in implementations of the IMS transactions. We then analyzed the distribution of the obtained counts.

IMS transactions [#]	Frequency [# of modules]	Relative frequency
1	409	0.53255
2	118	0.15365
3	56	0.07292
4	34	0.04427
5	64	0.08333
6	10	0.01302
7	8	0.01042
8	10	0.01302
9	17	0.02214
10	17	0.02214
11..15	13	0.01693
16..20	7	0.00911
21..25	0	0.00000
26..28	2	0.00260
29..31	2	0.00260
32	1	0.00130
>32	0	0.00000
Total	768	1.00000

Table 3.6: Distribution of the modules among the IMS transactions.

Table 3.6 presents the distribution of the module counts among the IMS transactions. In the first column we list the classes with the numbers of IMS transactions affected through alterations to an individual module. In the second and third column we provide

the frequencies of module occurrences in each class and the relative frequencies, respectively. The distribution characterized in Table 3.6 resembles the one in Table 3.5. In this case we observe that the frequency decays along with the increase in the number of IMS transactions. We find that nearly half of the modules (46.74%) belong to implementations of at least two different IMS transactions. We also find a few outliers. There is one module which when altered has a potential to affect as many as 32 IMS transactions.

The facts revealed through this analysis are meaningful from the perspective of planning a code improvement project. In a scenario in which the goal is to improve performance of IMS transactions it is generally desired to change a few Cobol modules and yet impact as many IMS transactions as possible. Based on the analysis we see that in the portfolio under study there exist many opportunities for choosing modules in such a way that multiple IMS transactions get affected. Exploitation of such opportunities yields ways to seeking configurations of modules which allow maximizing reduction in CPU resources usage while limiting code changes.

3.5 MIPS-reduction project

A team of experts specializing in optimizations of mainframe related costs carried out an improvement project, dubbed as the MIPS-reduction project, on the source code of the IT-portfolio under study. In this section we present our analysis of the code changes and the impact of the project on the MSU consumption. The objective of the project was to reduce the consumption of MSUs of the selected IMS transactions. The reduction was to be obtained through optimization of the DB2 related fragments in the source code which implement functionality of the transactions. The project was evaluated by comparing the average ratios of MSUs to transaction volumes before and after the changes were made. Our analysis shows that in the course of the project relatively innocent looking changes were made to either the code of the Cobol modules involved in the inspected IMS transactions or to the database configuration, e.g. index modification. Nevertheless, these changes were sufficient to cause a noticeable decrease in the ratio of MSUs to transactions volume. This effect was observable for both the IMS transactions initially selected for optimization and those impacted by the changes through interdependencies in the code.

3.5.1 Project scope

The portion of the IT-portfolio's source code covered in the project was determined on the basis of the selected IMS transactions. The expert's selection was limited to the pool of IMS transactions which were part of IT services of a particular business domain in the organization. In the selection process the primary point of departure was the ranking list of the top most executed IMS transactions in the production environment. After restricting the ranking to the relevant IMS transactions the choice fell on those transactions for which the average weekly ratio of the MSUs to transactions volume was the highest. From this ranking five IMS transactions were selected. The experts also found an additional IMS transaction, which was not in the ranking. Due to reported excessive CPU usage and the fact it was part of the business domain, for which the project was commissioned, it was

selected for code improvements. All of the six selected IMS transactions were interacting with the production's environment DB2 database.

IMS transactions inspected	6
Cobol modules covered	271
DB2 modules	87
DB2 modules with SQL statements	81
Modules with the potentially inefficient SQL constructs	65
# of instances of the potentially inefficient SQL constructs	424

Table 3.7: Characteristics of the portion of the IT-portfolio's source code covered in the MIPS-reduction project.

In Table 3.7 we provide some characteristics of the portion of the IT-portfolio's source code which was involved in the MIPS-reduction project. For the 6 IMS transactions selected for the project we obtained the list of modules which formed their implementation. This process boiled down to finding the source files associated with the implementations of the transactions. As a result 271 Cobol modules were retrieved. Amid those modules 87 ($\approx 32\%$) were identified as *DB2 modules* (Cobol programs with embedded DB2 code). It means that in each of those modules at least one `EXEC SQL` code block was present. The total number of modules containing SQL statements which were relevant from the MIPS-reduction project perspective was 81. This represents approximately 0.35% of all Cobol programs (23,004) in the portfolio's source code. In the remaining 6 modules we found only the SQL's `INCLUDE` statements. From the modules relevant for the SQL code analysis we isolated 65 in which at least one potentially inefficient SQL construct was identified. The constructs were identified with the SQL analysis tool, which is part of our framework. In total 424 instances of the potentially inefficient SQL constructs were found. The figure encompasses all the potentially inefficient SQL constructs listed in Table 3.2.

3.5.2 Selected modules

The expert team made the selection of the code that underwent improvements in a semi-automated process. The process included analysis of the time spent on execution of the DB2 modules involved in the implementation of the selected IMS transactions. The measurements were carried out with a tool called STROBE. STROBE for DB2 is part of the Strobe Application Performance Management product line. It measures the activity of z/OS based on-line and batch processing applications and produces detailed reports on how time is spent [57]. In the course of this analysis 6 Cobol-DB2 programs were selected. We scanned the selected programs to acquire characteristics of the SQL code.

In Table 3.8 we list in the first column the identifiers of the SQL constructs defined in Table 3.2. We list only the identifiers of those constructs for which at least one instance was found in the inspected code. In the inspected portion of the portfolio's source code we found no SQL statement which contained $AGGF_2$, DIST, GROUP, NIN, UNSEL, or UNION programming constructs. The next six columns are used to list instances of each

Construct ID	Module 1	Module 2	Module 3	Module 4	Module 5	Module 6
AGGF	1	7	2	0	3	0
COBF	0	0	0	2	0	0
$JOIN_2$	0	2	0	0	0	0
$JOIN_4$	1	0	0	0	0	0
ORDER	1	4	2	1	1	0
$WHRE_1$	0	1	0	0	0	0
$WHRE_2$	0	1	0	0	0	0
$WHRE_3$	0	7	2	0	1	0
$WHRE_{4..10}$	0	0	1	0	0	0
$WHRH_2$	1	5	2	0	4	2
$WHRH_3$	1	4	2	3	3	4
$WHRH_4$	1	1	1	0	0	0
$WHRH_{5..10}$	0	3	0	0	0	0
Instances in total	6	35	12	6	12	6
EXEC SQL blocks	8	40	20	12	14	28

Table 3.8: Characteristics of the SQL statements found in the DB2-Cobol modules which implement IMS transactions selected for the MIPS-reduction project.

construct encountered in the SQL code of the modules selected for optimizations. In the last two rows we summarize occurrences of the instances of the potentially inefficient programming constructs and the `EXEC SQL` blocks in the modules. From Table 3.8 it is clearly visible that in every module we find some SQL statement which exhibits the presences of the programming constructs dubbed by us as potentially inefficient. As we will show in all of these modules, except for *Module 1*, at least one of the recognized programming constructs was the direct cause of the triggered improvement actions, which resulted in changes to the code.

3.5.3 Changes

We now discuss the code changes done to the Cobol modules selected in the MIPS-reduction project by the expert team. We present the situation encountered in the code before and after the alterations were made by the expert team. For most of the modules we illustrate the code changes by providing code snippets taken from the original modules. For confidentiality reasons the variable, column and table names occurring in the snippets were changed.

Module 1 The original module *Module 1* contained three SQL queries embedded inside cursors. In these queries we found occurrences of the AGGF, $JOIN_4$, ORDER, and a series of $WHRH_x$ programming constructs. The only aggregate function found was present inside a sub-query of one of the cursors. It was used to return the maximum value. Two of the cursors contained `ORDER BY` clauses. The expert team discovered that the scenario in which these potentially expensive looking queries were used inside the program's code led to inefficient hardware resource usage. They identified what is known as a *nested cursor situation*. Presence of such a situation means that for two different cursors one is opened inside the other one. In case of the three cursors each time a row was fetched using

one of the cursors, one of the two remaining cursors was opened, values were retrieved and the cursors closed. Let us recall that for `SELECT` statements embedded inside cursors the query's invocation takes places each time a cursor is opened. In the scenario found in the program the nested cursors were opened and closed several times.

The improvement to the encountered situation involved changes to both the queries and the program's logic. According to the experts it was possible to supplant the three queries with a single one and obtain a result equivalent to those in the original program setting. Based on this conclusion all the three cursors were combined to form a single cursor, and the relevant Cobol code fragments were adequately adapted.

Module 2 In the original module *Module 2* we found fifteen SQL queries. Inside the code of the queries we found programming constructs of AGGF, JOIN_2, ORDER, and various instances of WHRE_x and WHRH_x. After the analysis the experts concluded that the `ORDER BY` clauses present inside two of the `SELECT` statements, embedded inside cursors, were the cause of the burden on the CPU. The `ORDER BY` clauses were removed after the existing cluster index on the table was extended with two additional column names used in the original `ORDER BY` clauses.

Module 3 In the original module *Module 3* a cursor was declared with the following SQL query:

```
1  SELECT DOCID       ,
2         NUMBER      ,
3         DATE_CREATED ,
4         CREATED_BY
5   FROM  FILEDOCUMENTS
6   WHERE NUMBER = :HOST-A
7     AND DOCID >= :HOST-B
8   ORDER BY DOCID ASC;
```

The `ORDER BY` clause in this deceptively simple query was identified as a performance hampering element. The solution was two-fold. First, the existing cluster index was extended with the field `DOCID`. And secondly, the `ORDER BY` clause was removed from the query.

Module 4 In module *Module 4* the SQL programming constructs of COBF, ORDER, and WHRH_3 were found. In this module code changes were triggered by the two instances of the COBF programming construct. In this module there were two `EXEC SQL` blocks which contained code dealing with the retrieval of a value from the DB2's special register: `CURRENT DATE`. It is starkly redundant to make DB2 calls for pure retrieval of these values while they are obtainable in a far cheaper way, in terms of CPU usage, through calls to the Cobol intrinsic functions. The expert team made a decision to replace the found `EXEC SQL` blocks with equivalent Cobol code. The following code fragment illustrates what changes were made to the code in module *Module 4* with respect to one of the two `EXEC SQL` blocks. Changes concerning the other block were done in an analogous manner.

```
1  ---- CODE ADDED ------------------------------------------------------
2  59:       01  WS-TEMP-DATE-FORMAT.
3  60:           03  WS-TEMP-YYYY           PIC X(4).
4  61:           03  WS-TEMP-MM             PIC X(2).
5  62:           03  WS-TEMP-DD             PIC X(2).
6  ---- CODE ADDED ------------------------------------------------------
7  63:       01  WS-FINAL-DATE-FORMAT.
8  64:           03  WS-FINAL-DD            PIC X(2).
9  65:           03  FILLER                 PIC X(1) VALUE ".".
10 66:           03  WS-FINAL-MM            PIC X(2).
11 67:           03  FILLER                 PIC X(1) VALUE ".".
12 68:           03  WS-FINAL-YYYY          PIC X(4).
13 ---- CODE DELETED ----------------------------------------------------
14 339:024000        EXEC SQL
15 340:024100            SET :DCLSERVICEAUTH.SA-STARTING-DATE
16 341:024200                = CURRENT DATE
17 342:024300        END-EXEC
18 ---- CODE ADDED ------------------------------------------------------
19 360:              INITIALIZE WS-TEMP-DATE-FORMAT
20 361:                         WS-FINAL-DATE-FORMAT
21 362:              MOVE FUNCTION CURRENT-DATE  (1:8)
22 363:                                   TO WS-TEMP-DATE-FORMAT
23 364:              MOVE WS-TEMP-DD          TO WS-FINAL-DD
24 365:              MOVE WS-TEMP-MM          TO WS-FINAL-MM
25 366:              MOVE WS-TEMP-YYYY        TO WS-FINAL-YYYY
26 367:              MOVE WS-FINAL-DATE-FORMAT
27 368:                TO SA-STARTING-DATE    IN DCLSERVICEAUTH
```

Whereas the above illustrated code transformation is a valid optimization solution it does, however, pose a risk for migration of this code to a newer version of a Cobol compiler. According to [60] the semantics of the `CURRENT-DATE` function changes in the Enterprise Cobol for z/OS. This will imply that in case of a compiler upgrade additional changes will be inevitable for this module.

Module 5 *Module 5* contained the following `SELECT` statement:

```
1       SELECT  START_DATE
2              ,DATE_CREATED
3              ,CREATED_BY
4              ,DATE_RISK_ASS
5              ,RESULT
6              ,DATE_REVISION
7              ,ISSUE_ID
8              ,PROCESS_ID
9              ,STATUS
10             ,DC_ACTIONCODE
11             ,ENTRY_ID
12             ,DATE_AUTH
13             ,AUTHORISED_BY
14        INTO  ....
15              ....
16         FROM TABLE001
17        WHERE NUMBER       = :NUMBER-DB2
18          AND START_DATE  =
19                ( SELECT MAX(START_DATE)
20                    FROM TABLE001
21                   WHERE NUMBER = :NUMBER-DB2
22                     AND START_DATE < :H-START-DATE-DB2
23                     AND ( ( :H-DF-IND-DB2      = '0'  AND
24                             STATUS         = '0'  AND
25                             DC_ACTIONCODE <> '0'
26                           )
27                          OR
```

```
28                              ( :H-DF-IND-DB2      = '1'  AND
29                                STATUS IN ('1','9')
30                              )
31                             OR
32                              ( :H-DF-IND-DB2      = '2'  AND
33                                ( STATUS        <> '0'  OR
34                                  DC_ACTIONCODE <> '0')
35                              )
36                            )
37                  )
```

In this query we found programming constructs labeled by us earlier as AGGF, ORDER, $WHRE_3$, $WHRH_2$ and $WHRH_3$. The experts focused on the `WHERE` clause inside the nested sub-query. The particularly interesting fragment of the SQL code is between lines 23 and 36 where three predicates concatenated with `OR` operators are present. In each `OR` separated block (lines: 23–26, 28–30, 32–35) a host variable `:H-DF-IND-DB2` is compared with different constants. The condition specified in the sub-query's `WHERE` clause will hold true if and only if the value of the host variable is either 0, 1 or 2. Such use of the above presented query in the code leads to potentially redundant invocation of a call to DB2. Regardless of the actual value held by the host variable `:H-DF-IND-DB2` the query will always be executed. To circumvent this situation from happening it is desired to first make appropriate comparisons with the host variable `:H-DF-IND-DB2` and only if any of them holds true make a DB2 call to execute the query. Such logic is programmable by means of Cobol conditional statements.

In this module, the presented query was rewritten in such a way that three new queries were introduced. Each new query differed from the original with the `WHERE` clauses. The comparison of the host variable `:H-DF-IND-DB2` with the constants was removed from the new queries. The control over invocation of the queries was embedded into a ladder of Cobol `IF` statements.

Module 6 In the original *Module 6* six cursors were found each of which contained the `WHERE` clause. All the `WHERE` clauses were analyzed and the potentially inefficient programing constructs of type $WHRH_2$ and $WHRH_3$ were found. In five of these cursors the `WHERE` clauses contained a predicate which involved comparison of a database retrieved value with a country code. In all five cursors this predicate was placed as second in a chain of `AND` operators. According to the experts inefficient ordering of the predicates in the `WHERE` clauses was negatively affecting the speed of execution of the query. The country code check, which was carried out as first, was not restricting the dataset well enough and thus hampering the performance. All five cursors had their `WHERE` clauses rewritten by changing the order in which the predicates are checked. The following code fragment illustrates the conducted transformation.

```
1  ---- OLD WHERE CLAUSE-----------------------------------------
2   WHERE ( PH_COUNTRY = :H-COUNTRY )
3      AND
4   ( PH_FONETICNAME   = :H-FONETICNAME )
5      AND
6   ( PH_YEAR_OF_BIRTH = :H-YEAR_OF_BIRTH)
7
8  ---- NEW WHERE CLAUSE -----------------------------------------
9   WHERE ( PH_FONETICNAME  = :H-FONETICNAME )
```

```
10      AND
11    ( PH_COUNTRY            = :H-COUNTRY )
12        AND
13    ( PH_YEAR_OF_BIRTH      = :H-YEAR_OF_BIRTH)
```

The code fragment shown above presents the `WHERE` clause of one of the five queries found in the Cobol program. In the original situation in all of the queries the `(PH_COUNTRY=:H-COUNTRY)` predicate was listed as first in the sequence of the `AND` operators (line 2). After changes to the code were made this predicate was replaced with the second one. In addition to the changes made in the code two indexes were created in the database to satisfy two of the cursors.

3.5.4 Impact analysis

We now present the analysis of the impact that the MIPS-reduction project had on the IMS transactions in the portfolio. Let us recall, we have observed that modules in the studied IT-portfolio are interdependent on one another. This observation turned out to hold true, in particular, for the Cobol modules which implement the IMS transactions selected for the MIPS-reduction project. Interdependencies in the code cause that the effects of code changes propagate to other parts of the portfolio. And, as explained earlier, this has implications for both the functionality and the execution performance of applications. As a consequence apart from the IMS transactions initially selected for the MIPS-reduction project also other IMS transactions became affected due to the carried out code changes. The expert team identified in total 23 IMS transactions which were affected in the aftermath of the project and used them for assessment of the project's performance.

In Table 3.9 we present a transactions-modules dependency matrix which shows the relationships among the project affected IMS transactions and the altered Cobol modules. The data required to construct the matrix was obtained by retrieving from source files implementing the transactions. The retrieval was accomplished by means of our tools. The columns correspond to modules changed in the course of the project and are labeled *Module 1* through *Module 6*. The rows, labeled T1 through T23, represent transactions pinpointed for the project assessment. Presence of a module i in the implementation of a transaction j is indicated with the symbol `X` placed on the intersection of the corresponding column and row.

The first observation which follows from the obtained transactions-modules dependency matrix is that a relatively low number of Cobol programs has a wide range of coverage of the IMS transactions. Also, the range in which the changed modules affect the transactions differs a lot. *Module 6* is only part of implementation of the transaction T18, whereas *Module 2* is part of implementation of nearly all listed transactions except for T11 and T21. This observation suggests that the *Module 2* turned out to be a shared bottleneck for nearly all affected transactions. One transaction for which our analysis did not reveal any relationship with the changed Cobol modules is T11. We were unable to disclose the reason for the transaction's inclusion in the impact analysis of the MIPS-reduction project. We believe that one of the reasons it was listed, despite the fact no code relationship was discovered, is that the DB2 indexes created or expanded in the course of the project had some effect on the transaction's performance.

Transactions	Module 1	Module 2	Module 3	Module 4	Module 5	Module 6
T1		X	X			
T2		X				
T3		X				
T4		X				
T5	X	X	X			
T6	X	X	X			
T7		X				
T8	X	X	X			
T9	X	X		X	X	
T10		X				
T11						
T12		X	X			
T13		X				
T14		X				
T15		X				
T16		X				
T17	X	X				
T18	X	X		X		X
T19		X				
T20	X	X		X	X	
T21	X					
T22	X	X	X			
T23		X				

Table 3.9: Transactions affected by MIPS-reduction project changes: code based impact analysis.

3.5.5 MSU reduction

The primary goal of the project was to reduce MIPS usage fees in the production environment which were linked to execution of the IMS transactions. The performance metric used for evaluation of the project's impact on the MSUs was the ratio of the MSUs consumed to the transactions volume in the given time frame for a given set of IMS transactions. Decrease of the ratio in time was considered meeting the project's goal. Obviously, the higher the decrease the higher the reduction of MSUs, and thus the better the result of the project. The expert team evaluated the actual impact of the MIPS-reduction project by comparing the weekly ratios of the MSUs to transaction volumes in two points in time: before and after the changes were migrated to the production environment. For this evaluation the experts considered the 23 transactions listed in Table 3.9.

We analyzed the effectiveness of the MIPS-reduction project in a time frame spanning across several weeks before and after the project changes were migrated to the production environment. We did this by studying the behavior of the performance metric in time. We took the available weekly data concerning the top 100 most executed IMS transactions. In the considered time frame 19 out of 23 IMS transactions affected by the MIPS-reduction project were consistently occurring in the weekly rankings. For these 19 transactions we had at our disposal totals for their weekly MSU consumption and transaction volumes. We aggregated these figures to obtain weekly totals for the MSU consumption and transaction

volumes for the bundle of the 19 transactions. From these totals we calculated the weekly MSUs to transactions volumes ratios.

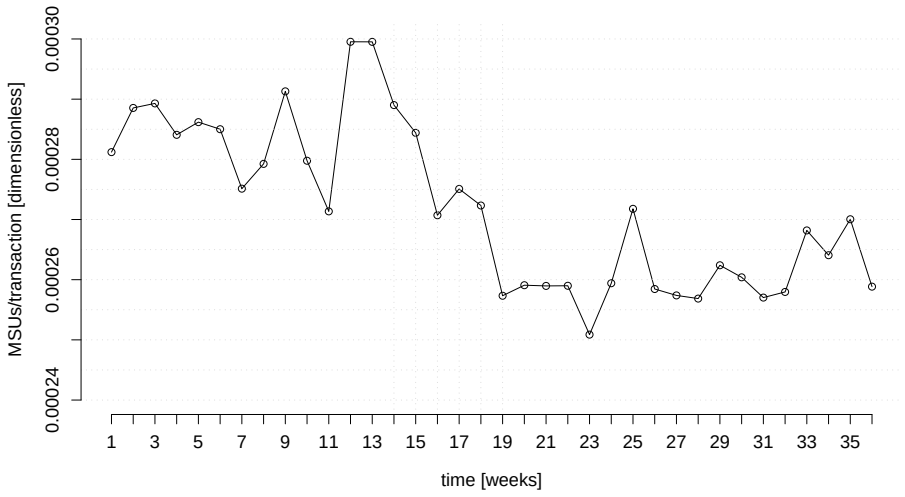

Figure 3.6: Time series of the average weekly ratios of the MSUs to transactions volume for the transactions affected by the MIPS-reduction project.

Figure 3.6 presents the obtained time series of the average weekly ratios of the MSUs to transactions volume for the set of IMS transactions affected by the MIPS-reduction project. The horizontal axis denotes time. Each tick on the axis represents a week number. The time series covers the period of 36 consecutive weeks. The vertical axis is used to express the average weekly ratios of MSUs to transaction volumes. The period in which the MIPS-project evaluation took place is distinguished in the plot by the vertical dashed lines which range from week 14 until 19. From the plot it is clearly visible that between week 14 and 19 the average ratio of the MSUs to transaction volumes decreased. Before week 14 the ratio ranged from 0.000271 until 0.000299. After week 19 the ratio decreased and oscillated between 0.000250 and 0.000271. The expert team estimated that the project reduced the annual MIPS related costs attributed to the optimized IMS transactions by 9.8%. The figure was provided by the expert team. Due to the confidentiality agreement we are unable to provide the monetary value of this reduction. According to the management the value of the estimated savings significantly outweighs the cost of the project.

External influence In order to verify that the reduction in MSUs was solely due to the MIPS-reduction project we checked the portfolio source code for changes. In particular, we scrutinized whether the source code which implements the functionality of the affected IMS transactions was altered between weeks 14 and 19. Let us recall, the initial

number of Cobol source modules covered by the project was 271. This number was the total number of Cobol modules which implemented the set of the six selected transactions targeted for MSU consumption improvements. In addition to those six transactions the expert team identified other 17 IMS transactions which also became affected by the MIPS-reduction project. To obtain the list of Cobol modules which implemented the 23 IMS transactions we analyzed their call-graphs. In total we found 328 Cobol modules. For each of those modules we looked up in a version control system whether any code alteration were reported and, if altered, migrated to the production environment. The check concerned alterations to source code in the time frame between weeks 14 and 19. Apart from the changes with respect to the modules altered as part of the project we found three other modules which were modified and brought into the production environment in the time. We studied carefully what the nature of modifications in these modules was. Two of the modules contained SQL code but it was not altered. All the modifications we found referred to minor changes in the Cobol code.

Production environment We also studied transaction and database calls volumes in the production environment in the period when performance of the project was measured. The objective of this analysis was to investigate whether there were any observable changes in the volume of transactions or the number of calls made to the database by the IMS transactions before and after project changes were migrated to the production environment. One variable which has effect on the MSU consumption by a transaction is the number of database calls made by the transaction. Decline in the number of calls has a positive effect on the MSUs since it lowers their consumption. For the analysis we considered the time series of the weekly total transaction volumes and the associated total volumes of database calls for the IMS transactions affected by the MIPS-reduction project. The time series covered the same time frame as in the analysis of the average weekly ratios of the MSUs to transaction volumes. Given that the data we had at our disposal were consistently reported for 19 out of 23 transactions we restricted our analysis to those 19 transactions. We aggregated the weekly transaction and database volumes to obtain weekly totals of these properties for a bundle of the 19 transactions. The obtained totals formed two time series which we analyzed.

In Figures 3.7 and 3.8 we present plots of the time series of the total transactions volume and the total volume of database calls made by the IMS transactions affected by the MIPS-reduction project, respectively. In both plots the horizontal axes are used to denote time. Each tick represents a week number. In Figure 3.7 the vertical axis denotes the total number of transactions, and in Figure 3.8 the total number of database calls. Given that the number of database calls is dependent on the number of transaction invocations it is not surprising that the two plots look alike. In fact, the Pearson's correlation coefficient is high (0.8951222) what confirms the strong correlation. The plots in Figures 3.7 and 3.8 do not exhibit any significant changes neither to the transaction nor to the database calls volume. In fact, the number of database calls is on the rise during the entire time frame.

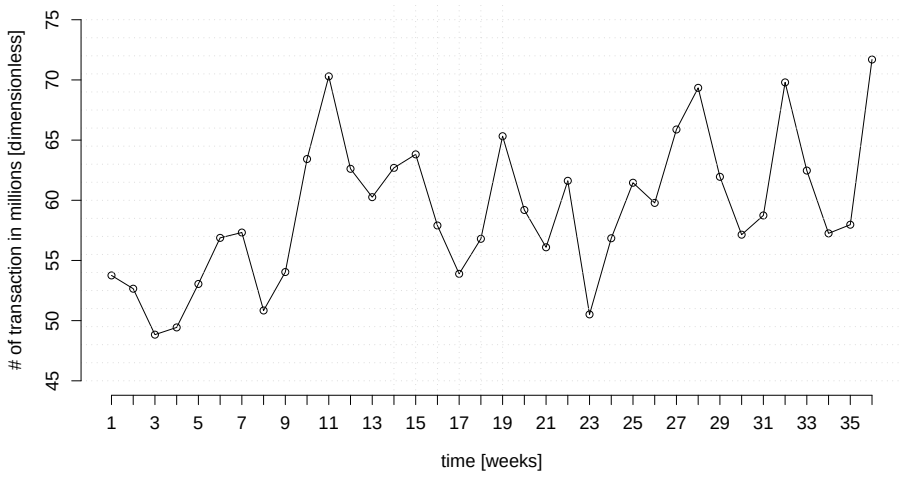

Figure 3.7: Time series of the total volume of IMS transactions affected by the MIPS-reduction project.

3.5.6 Summary

The MIPS-reduction project has shown that changes carried out to the DB2 related code had impact on the MSU consumption. We have observed a decline in the average weekly ratios of the aggregated MSUs and the transaction volumes after the project took place. According to the expert team the estimated annual savings related to the execution of the IMS transactions affected by the project were approximately 9.8%. It turned out that amid the 87 DB2-Cobol programs, which were relevant from the project perspective, changes to only six of them were sufficient to arrive at substantial savings. As our analysis has shown in all of those modules the SQL constructs deemed as potentially inefficient were present. Furthermore, in all of the modules their presence was directly or indirectly linked with the improvements carried out.

Based on our analyses the observed reduction in the MSU consumption was linked with the MIPS-reduction project. The decrease did not happen due to a decline in either the number of database calls or transactions invocations. Also, other changes done to the code in the portfolio did not exhibit signs which would question the project's role in reducing MSU consumption. The observations derived from the MIPS-reduction project provide a real-world evidence that analysis of the code from the angle of the presence of potentially inefficient SQL constructs provides for discovering meaningful candidates for code optimization.

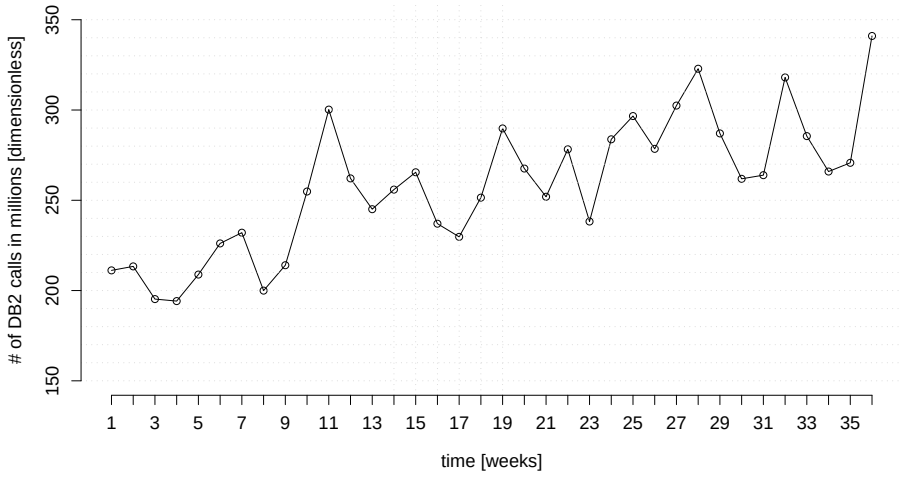

Figure 3.8: Time series of the total volume of database calls generated by the IMS transactions affected by the MIPS-reduction project.

3.6 Portfolio-wide control

In this section we present how we approach control of the CPU resource usage at the level of a portfolio of mainframe applications. First, we show that for a large mainframe environment control of the CPU resource usage requires a structured approach. On the basis of the case study we show that both the proportions of the DB2 related code in the portfolio and the potentially inefficient SQL statements are high; hence making it clearly far from trivial to effectively carry out improvements. Next, we present how by combining the data available from the mainframe usage reports along with the source code extracted facts it is possible to narrow down the amount of sources that require scrutiny. For the portfolio under study we were able to restrict ourselves to less than 1% of all Cobol modules. To aid the process of locating the likely performance hampering Cobol modules we use the code derived data on the potentially inefficient SQL programming constructs and data concerning code interdependencies. Finally, we present how to assess the expected savings in MSU consumption for particular code improvement projects. To enable quantification of the potential savings we constructed two formulas which are based on the data available from the evaluation of the MIPS-reduction project. These formulas serve as rules of thumb for estimating the percentage of the average monthly MSUs saved for a given IMS transaction given the number of altered DB2-Cobol modules. We illustrate how to plan code improvement projects by presenting two scenarios for source code improvements.

3.6.1 DB2 code

We analyzed the source code of the IT-portfolio to obtain insight into the state of the SQL code. The analysis covered 246 IT-systems which were built of 23,004 Cobol modules (the number does not include copybooks). Of these modules, DB2-Cobol programs constituted nearly 25% (5,698). The DB2-Cobol programs were identified by seeking the source code for the presence of the `EXEC SQL` blocks. In total we found 55,643 `EXEC SQL` blocks. Inside the blocks there were 14,387 `SELECT` statements which were used either as plain SQL queries or were embedded inside cursor declarations. The content of the remaining `EXEC SQL` blocks comprised SQL communication areas (SQLCA), `INCLUDE` statements, and the remaining SQL commands permitted in the Cobol programs' code.

We also measured the extent of the potentially inefficient programming constructs in the portfolio. In order to accomplish this task we applied our code analysis tools. Detailed results of this analysis are presented in Table 3.10.

Construct ID	**DB2 modules**	**% of DB2 modules**	**Instances**	**% of Instances**
ORDER	1953	34.28	3232	23.72
WHRH_2	1327	23.29	2488	18.26
AGGF	945	16.58	2017	14.80
WHRH_3	744	13.06	1175	8.62
NWHR	704	12.36	1038	7.62
WHRH_4	379	6.65	573	4.20
JOIN_2	345	6.05	575	4.22
UNSEL	345	6.05	591	4.34
$\text{WHRH}_{5..10}$	345	6.05	637	4.67
COBF	276	4.84	369	2.71
DIST	160	2.81	266	1.95
GROUP	82	1.44	189	1.39
JOIN_3	79	1.39	141	1.03
UNION	41	0.72	54	0.40
$\text{WHRH}_{11..20}$	33	0.58	82	0.60
$\text{JOIN}_{4..7}$	27	0.47	53	0.39
WHRE_3	13	0.23	48	0.35
NIN	12	0.21	26	0.19
WHRE_1	11	0.19	22	0.16
$\text{WHRH}_{21..50}$	7	0.12	11	0.08
WHRE_2	6	0.11	19	0.14
$\text{WHRE}_{4..10}$	3	0.05	5	0.04
$\text{WHRE}_{11..30}$	2	0.04	8	0.06

Table 3.10: Potentially inefficient programming constructs found in all of the portfolio Cobol sources.

In Table 3.10 we present results of the SQL code analysis which covered all the DB2-

Cobol sources. In the first column we list the identifiers of the potentially inefficient SQL programming constructs. These identifiers are taken from Table 3.2. For the constructs: JOIN$_x$, WHRE$_x$, and WHRH$_x$ we created classes that cover ranges of instances. The ranges are determined by the minimum and maximum values for x. For instance, in case of the WHRH$_x$ one of the classes that we created is WHRH$_{5..10}$ which covers all instances of the WHRH$_x$ where x ranges from 5 until 10. By doing so we eliminated the large number of identifiers of constructs which occurrences were scarce; hence making the presentation of our table clearer. In the second column we provide the total numbers of Cobol modules in which the particular programming constructs were found. The following column, labeled *% of DB2 modules*, provides the percentage of the DB2 modules containing instances of the programming constructs with respect to all DB2 modules. In the column labeled *Instances* we give the total number of instances of the programming construct. And, in the last column we provide the percentage of the instances with respect to all instances of the potentially inefficient constructs. The counts and percentages relating to the instances are given only for information purpose. We did not utilize this information in our work. Rows of the table were sorted in a descending order according to the number of DB2-Cobol programs.

The primary observation that follows from Table 3.10 is that there is a large number of DB2-Cobol modules which contain the potentially expensive constructs in the embedded SQL code. We found approximately 68% (3,874) of the DB2-Cobol programs with SQL statements in which at least one instance of the constructs listed in Table 3.2. In total we found 15,924 instances of such constructs. The top most occurring construct is ORDER. It is present in 1,952 modules (34.26% of all DB2 programs). It tops the list both in terms of the total number of instances and modules it occupies. The high percentage of modules with the ORDER construct is somewhat alarming given that the corresponding SQL `ORDER BY` clause is known for increasing the probability of performing the CPU intensive operation such as sorting. From the MIPS-reduction project we know that presence of this construct did trigger improvement actions. Of course, the high incidence of ORDER constructs does not necessarily mean that the mainframe environment suffers from a large unnecessary MSU consumption. We did not conduct an analysis which would enable us to confirm or rule out such a possibility. However, this construct should be used with care.

Another construct which is worth discussion, for its relatively high position in the ranking, is the COBF. Let us recall that this construct refers to redundant calls made to the database. We have seen that due to the presence of this construct one module was altered during the MIPS-reduction project. Since use of the COBF is starkly redundant we found it surprising that 369 instances of this construct occur in 276 DB2-Cobol modules. Despite the fact that 276 constitute roughly 1.2% of the Cobol files it still appears as an overuse of a *bad* programming practice.

In the bottom of the Table 3.10 we see several variations of WHRE$_x$ constructs. These constructs occur in a relatively small number of DB2-Cobol programs but due to their nature, which is nearly pathological, they deserve special attention. The positive information derivable from the SQL analysis is that we find only 24 modules which exhibit presence of WHRE$_x$ constructs. Let us note that the given number is not the total of the numbers listed in the second column of Table 3.10 next to the WHRE$_x$ identifiers. This derives from the fact that multiple variations of the construct may occur in a single module. Thus

the actual reported total number of modules is lower than the total derivable from the table. Despite the low number of occurrences in the code these constructs are interesting from the perspective of CPU performance. In fact, in the MIPS-reduction project one module had a query with a WHRE_3 construct decomposed into three queries. These three queries were later embedded in the Cobol code in such a way that their invocations were controlled through a sequence of Cobol's `IF` statements.

From the analysis of the embedded SQL code we see a high incidence of potentially inefficient SQL programming constructs in the portfolio source code. The amount of modules, which on the grounds of presence of these constructs, speak for an in-depth analysis is huge. Such an amount constitutes a vast playground for SQL tuning experts. Of course, analyzing all the encountered cases and carrying out improvements, if necessary, is advisable from the software engineering perspective. The observations from the MIPS-reduction project show that presence of the constructs found in the portfolio poses a threat to efficient usage of the hardware resources. However, it is nearly infeasible for a portfolio of this size and such importance for the business to conduct such an overhaul of the source code. In fact, such a project also poses risks since code might become erroneous. Therefore, our advise is to carry out the impact analysis portfolio-wide, and mark the revealed code as such. Then if the code has to be changed anyway due to some business reason it is a good idea to combine this effort with code optimizations. By aggregating information about earlier code optimization efforts and their effects on the MSU consumption, this will become more and more routine and less risky. So we propose an evolutionary approach towards portfolio-wide control of CPU resource usage.

3.6.2 Major MIPS consumers

We now focus on a way to narrow down the search space for the modules with potentially inefficient SQL constructs. We propose to go after the low-hanging fruit and concentrate improvement efforts on the source code implementing the major MIPS consumers in the mainframe environment. Such an approach enables spending effort on those parts of the portfolio which provide for a relatively high savings potential. And, it also allows keeping improvements low risk for the business by limiting the extent of changes to the portfolio source code.

Accounting data In order to determine major MIPS consumers in a portfolio it is indispensable to have access to mainframe usage figures. Mainframe usage data, also known as the accounting data, are the essential input for fees calculation. The type of data which is required to conduct such calculations is collected by default. Apart from this essential data, z/OS also enables collection of all sorts of other data to provide detailed characteristics of particular aspects of the system or software products. Whether this is done or not depends on the demand for such data by the organization utilizing the mainframe. It is important to note that collection of the accounting data is a workload on its own, and also involves MSU consumption. This fact implies that careful selection must be made as to what is collected. For our analyses we had at our disposal weekly IMS transaction rankings of the top 100 most executed IMS transactions. This set of rankings was our primary source of data on the MSU consumption in the environment.

Transactions volume We propose to use the IMS transactions volume ranking to steer reduction of the MSU consumption in the IT-portfolio. The argumentation behind this decision is two-fold. Firstly, the general tendency is that the transactions volume is on the rise. This phenomenon finds its justification, for instance, in the fact that organizations gradually process more data using their IT infrastructure. Besides that clients of the organizations also get more used to the typically offered web front-ends to the IT-systems and as a result more data gets processed by the computers. In our case study the top used IT services have been experiencing increased CPU resource consumption over a long time frame. Management's attention has fallen mainly on five of those systems since more than 80% of MIPS were deemed to be used only there. Except for the increasing MIPS usage these systems are characterized by one other property. They are all interacting with the database. Nearly 57% of the transactions found in the top 100 IMS transactions were linked to those five systems. In a period of one year we have observed a 12.22% increase in the volume of transactions which involved these systems. We found no indication that this increase was temporary. In fact the systems which we analyze are the core systems of the organization. The increase of 12,22% most likely is a result of doing more business.

The other argument behind relying on the transactions volume to steer reduction of the MSU consumption is its relationship with the actual major MIPS consumers. We carried out an analysis of the MSU usage associated with the top most executed transactions. It turns out that the position of a top-ranking IMS transaction is a reliable indicator of the level of participation in the MIPS related costs. For this analysis we considered data reported in a period of six months for the top 100 most executed IMS transactions. For each transaction we calculated the totals for the number of transaction executions and the consumed MSUs in that period. We sorted the list of transactions in terms of the total number of executions in a descending order and plotted the associated MSU figures.

In Figure 3.9 we present the observed relationship between the transactions position in the ranking and the associated total MSU usage in the period of six months. The horizontal axis is used to indicate the transactions position in the ranking. The values range from 1 until 100. The lower the value the higher the position in the ranking. A transaction with a higher transaction-volume obtains a higher ranking. The vertical axis is used to present the total number of MSUs consumed. From the plot it is visible that the observed MSU consumption tends to decrease along with the position in the ranking. Although we see several exceptions, this observation suggests that the transaction's frequency of execution is key in the transaction's contribution to the MIPS related costs.

The fact that MSU usage is positively related to the volume of executions is not surprising. Execution of each transaction is associated with some consumption of MSUs. The amount of consumed MSUs varies from transaction to transaction. Varied usage derives from the fact that CPU utilization depends on the actual number of instructions passed to the processor. This is ultimately determined by the logic embodied in the code of the executed programs implementing the transaction and data flow. The peaks in the ranking which are noticeable for several transactions in the plot in Figure 3.9 are due to the varied MSU consumption. Nevertheless, we assume the ranking as a sufficiently good tool to locate major MIPS consumers in the IMS environment and rely on it in the process of pinpointing source code for the DB2 related improvements.

Based on our observations, from this and other case studies, a software portfolio typ-

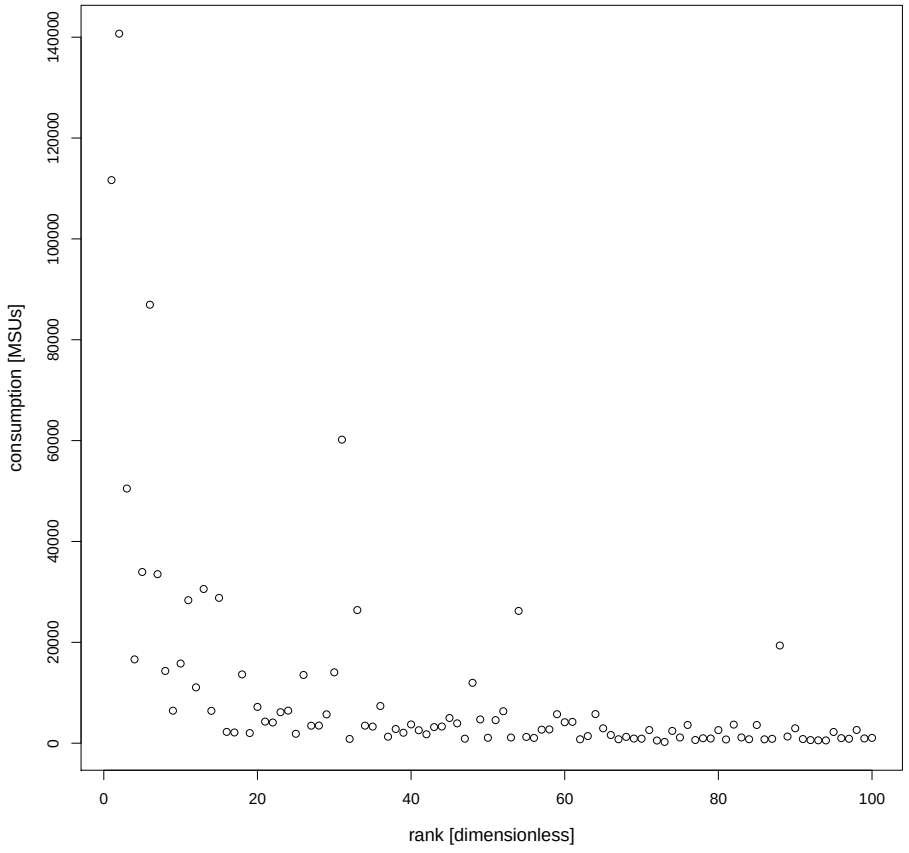

Figure 3.9: Relationship between the transactions position in the ranking and the MSU consumption.

ically contains only few applications which share some common property, e.g. play a major role in supporting business. This phenomena follows the well-known 80-20 rule, or Pareto principle, which roughly translates into: 80% of the effects come from 20% of the causes [87]. Whereas there is no hard proof to support this claim this observation appears to hold true for most portfolios. For the case study used in this chapter we found that the top 100 IMS transactions contain nearly 57% of transactions linked to IT-systems which are attributed to more than 80% of MIPS usage. Whereas the situation encountered is not exactly in line with the Pareto principle, application of the principle to the investigated problem is straightforward.

DB2 transaction Among all IMS transactions that we find in the top 100 ranking we concentrate, for obvious reasons, on those which interact with the database. We will refer to an IMS transaction as a DB2 transaction each time we find in its implementation code which is related to DB2. Given that we approach the IT-portfolio from the source code perspective to determine which IMS transactions are DB2 transactions we screen the content of source files. Namely, we first map the IMS transaction with source files that implement its functionality. Next, we scan the files and seek for the presence of particular programming constructs which are meant to implement interaction with the database. A typical DB2-client program written in Cobol contains embedded SQL code which is enclosed in `EXEC SQL` blocks. We use this fact to classify Cobol programs as DB2-clients. Finally, once at least one source module containing DB2 related code is found in the implementation of an IMS transaction it is marked as DB2 transaction.

Given that the set of DB2 transactions was chosen on the basis of static code analysis we allow in the set those IMS transactions which in runtime never establish any connection with the DB2. This situation derives from the fact that logic in the implementation of the transactions ultimately dictates when a DB2 related code is executed. This lack of accuracy is, however, unavoidable given the kind of analysis we deploy in our approach. Nevertheless, by distinguishing between DB2 and non-DB2 related IMS transactions we restrict the number of source modules to those that are relevant for further analysis.

3.6.3 Low-hanging fruit

Conducting DB2 related improvements inevitably boils down to reaching for the relevant part of the source code. So far we have explained that for the purpose of focusing improvement efforts we restrict ourselves to the top 100 IMS transactions which we dubbed the major MIPS consumers. Having at our disposal the set of Cobol modules which implements the transactions we carried out analysis of the embedded DB2 code. We now present results of this analysis. As it turns out the code which is relevant from the perspective of improvements constitutes, in fact, a small portion in the portfolio.

Top executed DB2 code In an earlier part of this chapter we presented results of the analysis of the SQL code which was spanned across all the Cobol modules found in the portfolio. In the context of this section we restrict our analysis only to those DB2 Cobol modules which are associated with the implementation of the top 100 IMS transactions. Let us recall, in total the implementation comprises 768 Cobol modules. Of these modules 258 contain DB2 related code.

In Table 3.11 we present results of the SQL code analysis from those DB2-Cobol sources which implement the top 100 IMS transactions. In the first column we list the identifiers of the potentially expensive SQL programming constructs. These identifiers are taken from Table 3.2. Again, for the constructs: JOIN_x, WHRE_x, and WHRH_x we created classes that cover ranges of instances. In the second column we provide the total numbers of Cobol modules in which the particular programming constructs were found. The following column, labeled *% of DB2M*, provides the percentage of the DB2 modules containing instances of the programming constructs with respect to all DB2 modules. In the column labeled *inst* we give the total number of instances of the programming

Construct ID	DB2 modules	% of DB2 modules	Instances	% of Instances
ORDER	72	1.26	134	0.98
$WHRH_2$	71	1.25	155	1.14
$WHRH_3$	50	0.88	84	0.62
AGGF	43	0.75	136	1.00
$WHRH_4$	24	0.42	36	0.26
NWHR	22	0.39	44	0.32
$JOIN_2$	21	0.37	44	0.32
$WHRH_{5..10}$	15	0.26	34	0.25
COBF	14	0.25	17	0.12
UNSEL	13	0.23	22	0.16
DIST	10	0.18	33	0.24
$WHRE_3$	9	0.16	34	0.25
$JOIN_3$	8	0.14	26	0.19
$WHRE_1$	8	0.14	18	0.13
GROUP	5	0.09	7	0.05
$JOIN_{4..max}$	5	0.09	17	0.12
UNION	4	0.07	5	0.04
$WHRE_{4..10}$	2	0.04	2	0.01
$WHRE_2$	1	0.02	1	0.01
$WHRE_{11..max}$	1	0.02	6	0.04
$WHRH_{11..20}$	1	0.02	6	0.04

Table 3.11: Potentially inefficient programming constructs found in the Cobol sources which implement the top 100 IMS transactions.

construct. And, in the last column we provide the percentage of the instances with respect to all instances of the potentially inefficient constructs. Rows of the table were sorted according to the number of DB2-Cobol programs.

In Table 3.10 the modules which contain the ORDER construct constitute over 34% of all DB2-Cobol modules whereas in Table 3.11 we see roughly over 1%. Table 3.11 exhibits similarity with Table 3.10 in terms of order of the programming constructs. Again, the ORDER construct tops the list, the variations of the $WHRH_x$ constructs occupy its upper part, the COBF resides around the middle. Also, majority of the positions in the bottom of the list is, again, occupied by the $WHRE_x$ constructs. Of course, in Table 3.11 the proportions of the modules and instances are much lower but this is due to the fact that we look at a much smaller portion of the portfolio. What is particularly interesting are the modules labeled with $WHRE_x$. In the entire DB2-Cobol part of the sources we found 24 modules which contain instances of those constructs. This time we screened 258 files and we see 21 modules with this property. Given that the $WHRE_x$ constructs are pathological it is surprising to learn that 87.5% of source files implementing the top IMS transactions contain such code. Despite the presence of nearly a full array of SQL constructs deemed as potentially inefficient the overall number of DB2 modules with such constructs is rather low (180). It is fair to say that in the worst case scenario code improvements are restricted

to at most 180 Cobol programs. Given the total number of Cobol modules in the analyzed portfolio (23,004) this constitutes approximately 0.78%.

The small portion Analysis of the mainframe applications' SQL code revealed that there are plenty of opportunities to fine-tune interaction with the database. As we have already explained when dealing with a business critical portfolio a preferred approach is to focus on achieving savings at low-risk. In general, the lower the amount of changes applied to the code the lower the risk of encountering defects in the applications. In case of changes to the DB2-related code, alterations are normally restricted only to a few lines of the embedded SQL code. These minor changes can be sufficiently tested without too much effort prior to migrating modules to the production environment; hence making them relatively low-risk alteration of the applications.

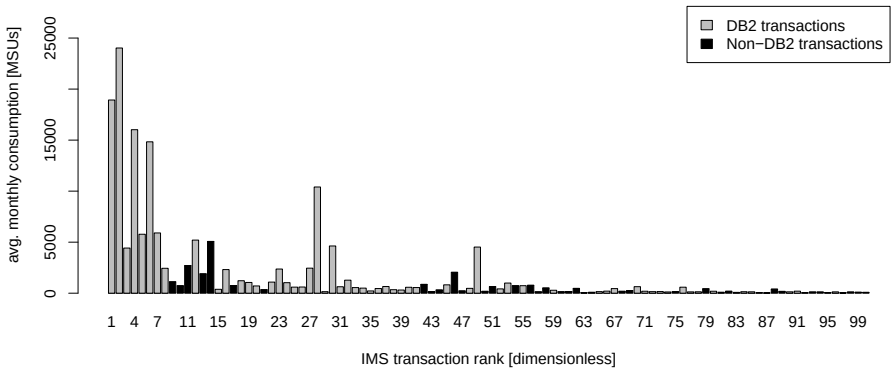

Figure 3.10: Bar plot of the monthly MSU consumption for each IMS transaction in the top 100 ranking.

In Figure 3.10 we present a plot of the monthly MSU consumption for each transaction in the top 100 ranking. The horizontal axis is used to provide the rank of an IMS transaction according to monthly transactions volume. The data presented in the plot were captured in a particular month. The vertical axis is used to present the observed consumption of MSUs by the transactions. In the bar plot we distinguish two types of transactions: those which interact with the database (gray) and those which do not (black). In the studied pool of IMS transactions 64% interacts with the database. Similarly as in Figure 3.9 we see that the high-ranking IMS transactions tend to have higher MSU consumption. This is not a one-to-one correspondence. We see some deviations. What is interesting in this plot is that it is visible that the DB2 interacting IMS transactions have a distinctly higher MSU consumption than other IMS transactions.

We took the top 100 IMS transactions listed in Figure 3.10 and analyzed them from the perspective of the amount of source modules present in their implementation. In the

first step we mapped with each IMS transaction a set of modules used for their implementation. Each set was associated with a position number identical to the rank the corresponding IMS transaction occupied in the ranking. Then, for each position i we computed the cumulative number of distinct modules which exist in the implementations of transactions ranked 1 through i. This operation was accomplished by calculating the union of the corresponding sets. In addition, for each position we obtained cumulative counts of the DB2-Cobol modules and the DB2-modules with the potentially inefficient programming constructs. Eventually, we had at our disposal three vectors of data containing the cumulative counts.

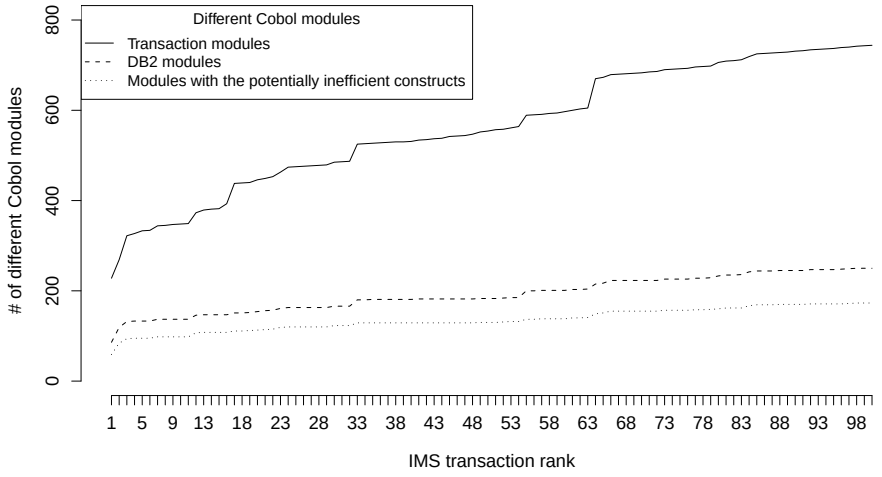

Figure 3.11: Properties of the Cobol code implementing the top 100 IMS transactions.

In Figure 3.11 we present the results of our analysis of the code implementing the top 100 used IMS transactions. The horizontal axis lists the IMS transaction rank in terms of execution volume and the vertical axis provides the Cobol modules cumulative counts. The counts of the modules implementing the transactions reflect: the total number of Cobol modules (solid line), the DB2 interacting modules (dashed line), and the DB2 interacting modules containing the potentially inefficient SQL code (dotted line).

What is observable from the plot is that a relatively large number of the major IMS transactions are implemented by a rather small number of Cobol programs (768). In the global portfolio view these modules represent approximately 3.2%. An even smaller number of code is responsible for interaction with the DB2 (258 modules, 1.07%). This follows from the fact that there exist many code interdependencies among the implementations of the IMS transactions. Finally, we see that among the DB2 modules implementing the top 100 IMS transactions, 180 (0.78%) contain the potentially inefficient SQL constructs. Bearing in mind the plot in Figure 3.10 we see that in order to address the top

MSUs consuming DB2-transactions it is sufficient to take into consideration a relatively small portion of the code. By taking into account different subsets of the IMS transactions it is possible to arrive at various configurations of Cobol modules which are worth inspection. These configurations will possibly differ in the numbers of modules, and what follows from it, the risk associated with optimizing them and migrating to the production environment. We will now show how we utilize the code extracted data in planning a MIPS-reduction project.

3.6.4 Improvement scenarios

We use the data gathered during the analysis of the code and show how to use it to plan source code improvement projects. We illustrate portfolio code improvements on the basis of two what-if scenarios. For each scenario we present estimations of the possible savings.

Estimation tools In order to be able to estimate the potential savings in MSU consumption resulting from carrying out a code improvement project we need some benchmark data. The only data that were at our disposal on that kind of projects came from the MIPS-reduction project evaluation report. We chose to rely on it in estimating the potential MSU savings for the similar future projects. Our analyses so far have been revolving around optimization of the MSU consumption by the IMS transactions through improvements to their implementation. Using the available data we drafted two formulas for estimating the percentage of the monthly MSUs saved for executing an IMS transaction given the number of modules that were altered in its implementation. One formula provided us with the average expected monthly percentage of savings, the other, with the maximum.

We considered savings calculations provided in the evaluation of the MIPS-reduction project by the expert team. We had at our disposal percentages of the MSUs saved and the average monthly numbers of MSUs consumed by the transactions. These were obtained from the historical yearly data. In order to associate this data with the code alterations, for each transaction we counted the number of modules changed in the implementation of the transaction during the MIPS-reduction project. The counts ranged from 0 up to 4 (0 applied to the one exception we reported earlier). We grouped the transactions according to the number of modules altered. We eliminated groups which contained less than one data point. Eventually, we arrived at 3 groups of transactions with 1, 2, or 3 modules improved. We characterized each group with two values: the average and the maximum percentage of MSUs reduced. For each group the average was calculated as the weighted average of the reported percentages with the historical average monthly MSU consumption as weights.

We chose to do extrapolation on these small sets of three points. Although, the linear relationship appears to be appropriate given the points we had we did not find it suitable. Whereas we expect the increase in MSU reduction as we improve more and more Cobol modules we do not expect this growth to be proportional. We decided to choose logarithms as the most appropriate family of functions to extrapolate. Logarithmic functions are increasing, if the base is larger than one, and their growth is very slow. This property stays in line with our intuition. Having assumed nonlinear behavior we used the available data points to extrapolate. We found the following relations.

$$R_{avg}(m) = 0.010558 \cdot \ln(3561.230982 \cdot (m+1)). \tag{3.1}$$

$$R_{max}(m) = 0.0512401577381 \cdot \ln(4.138711 \cdot (m+1)). \tag{3.2}$$

Formulas 3.1 and 3.2 provide the tools we used for estimating the expected MSU reduction in altered transactions. The R_{avg} stands for the average expected percentage of reduction. And, the R_{max} gives us the maximum expected. Both formulas use as input m which is the number of Cobol modules changed in a transaction's implementation.

In our approach to estimating savings we rely on the fact that source code is altered. We size the extent of code alterations with the number of modules changed. We do not take into account the actual scope of changes, for instance, the number of SQL statements improved within a particular module. This is obviously a simplification. As an alternative one might consider counting the number of changed lines of code. However, such approach bears its own drawbacks and given the small number of data points does not guarantee achieving any better estimates. In our case by relying on limited data we were able to equip ourselves with tools which enable us to make estimations based on the past experience. Let us note that the formulas were derived from characteristics of a specific production environment and therefore it is possible that for other environments one must consider devising their own using different data. Nevertheless, in the face of having no approximation tools we came up with formulas that we use as rules of thumb for estimating the size of savings obtainable from DB2-related code improvements.

What-if scenarios Naturally, there exist various scenarios in which it is possible to achieve reduction in MSU consumption through code improvements. Each scenario is characterized with a level of potential savings and a scope with which the applications' source code is affected. The first element has obvious financial consequences. The latter relates to IT risk. Two scenarios for code improvement in the implementations of the major IMS transactions were analyzed. To estimate savings we used the formulas 3.1 and 3.2. For the sake of analysis we assumed that the basic improved code unit is a DB2 Cobol module. The costs relating to code improvements are assumed to be low compared to potential savings. The following are the analyzed scenarios:

Major overhaul We focus on the top 100 IMS transactions. All DB2 modules bearing the potentially inefficient SQL constructs found in code analysis are improved. The emphasis in this scenario is on code quality improvement. Savings in MSU consumption are expected to follow as a consequence of improvements to the code.

Minor changes The number of DB2 modules dispatched for improvements is kept low yet the impact of the alterations made to these modules allows embracing IMS transactions which offer potential for high savings. Selection of modules for this scenario was based on facts such as presence of potentially inefficient SQL constructs and code interdependencies among implementations of the IMS transactions. The

emphasis in this scenario is on achieving significant savings in MSU consumption through relatively small number of alterations in the applications' code.

Scenario	**Modules (#)**	**Est. monthly savings (%)**	
		Avg	**Max**
Major overhaul	180	9.61%	16.83%
Minor changes	14	6.07%	9.56%
MIPS-reduction project	*6*	*3.84%*	-

Table 3.12: Summary of the analyses of the scenarios.

In Table 3.12 we present a summary of the scenarios from the perspective of the total potential savings and the amount of source code which is affected. In the first column we list the scenarios. In the second we provide the number of Cobol source modules which undergo improvements. In columns 3 and 4 we present the estimated savings. The savings are expressed as the percentage of the average monthly MSU consumption by the top 100 IMS transactions. The scenarios are summarized on the basis of calculations obtained from formulas 3.1 and 3.2. The columns contain labels *Avg* and *Max* to distinguish figures obtained with the two formulas. In the last row of Table 3.12 a summary of the MIPS-reduction project is provided. The figures given are the actuals and therefore cells in columns labeled with *Max* do not contain any value.

It is clear from Table 3.12 that in all scenarios MSU consumption savings are to be expected. In principle, the more changes to the code the higher the projected yields. The highest expected savings are generated in the *Major overhaul* scenario. These savings come for the price of in-depth analysis and alterations to a large number of modules. An interesting result is revealed through the *Minor changes* scenario in which only 14 modules are considered yet the expected savings are substantial. In the last row estimates regarding changes to the same Cobol modules that were changed in the MIPS-reduction project are presented. Let us note that the value is expressed as the percentage of the average monthly MSU consumption by all the top 100 IMS transactions.

Which scenario is the best to depart from depends on a number of factors. For instance, on the desired level of savings, the risk tolerance, or the degree to which the source code is to be improved. Nevertheless, the analyses presented here allow for rational decision making based on facts.

The fact that the savings in the MIPS-reduction project were lower than those we estimated for the two analyzed scenarios is primarily due to differences in the scope of the portfolio source code involved. Our approach is unlike the expert team. We departed from screening the code which implements the top 100 IMS transactions and marked the potentially inefficient code. The MIPS-reduction project team was constraint by the business domain which commissioned the project. What follows from it is that they focused on a particular subset of the IMS transactions. From there they made the selection of the code to be improved. Taking the IT-portfolio management point of view, we would recommend not to constrain code improvements by the business views, such as the business domains. In a mainframe portfolio of applications serving multiple business domains the

source code constitutes a uniform space for searching improvement opportunities. The mainframe usage reports are already a reliable guide which points to the major cost components. There is no need, from the technology perspective, to impose artificial constraints when the goal is to lower the operational IT costs.

3.7 Practical issues

In order to control CPU resource consumption and generate savings, the control efforts must be embedded within the organization and in the ongoing software process. We already alluded to that by proposing an evolutionary approach towards MSU consumption reductions: monitoring, marking and changing in combination with other work on the code. Since there will be new modules and enhancements, there will also be new venues for reductions in CPU resource usage, and we find those during monitoring. It would be good to analyze new code upfront for this to preventively solve the problem before it occurs. Of course, in order to take advantage of all those opportunities there must exist a clearly defined software process.

Within the software process two phases should be distinguished. The first phase involves code improvements to the existing implementations of the MIPS consumers. The second phase involves ongoing monitoring of changes in the production environment MSU consumption, and also, monitoring of the quality of the code in the pre-production stage.

3.7.1 Phase I

As the MIPS-reduction project has shown, code improvements yield significant savings. The MIPS-reduction project was restricted to a particular group of IMS transactions. We have demonstrated that by carrying out code improvements on a portfolio scale it is possible to achieve higher reduction in the MIPS related costs. These facts speak for themselves in managerial terms. In addition, what follows from the code improvements is also the increase in the quality of the source code. For an organization this aspect translates, for instance, into prolonged longevity of the portfolio, better response time of the applications, or better control of the IT-assets. In order to commence with the CPU resource usage control the current condition of IT must undergo screening.

Implementation of phase I could take form of one of the source code improvement scenarios analyzed in this chapter. Of course, for a particular mainframe environment there needs to be data available which would allow regeneration of the results used to construct the improvement scenarios presented in here. Implementation of a portfolio-wide code improvement project should take into account lessons learned from analysis of the entire Cobol written portfolio, the MIPS-reduction project, or recommendations from the portfolio experts.

To facilitate analysis of the Cobol portfolio we developed a number of tools which allow for extraction of the relevant data from the source code. These tools are capable of delivering an analysis each time changes to the portfolio take place. In fact, these tools are suited for analysis of many Cobol sources. Minor modifications might be necessary. Of course, except for the collection of the code related data it is also necessary to have at

your disposal the mainframe usage data. This data, however, is usually available for all mainframe environments. If not their collection needs to be enabled.

3.7.2 Phase II

There is a clear need to monitor CPU resource usage on an ongoing basis. That is the essential role of phase II. It has been observed that source code in IT-portfolios evolves. According to [112] each year Fortune 100 companies add 10% of code through enhancements and updates. In the Cobol environments alone, which are estimated to process approximately 75% of all production transactions, the growth is also omnipresent [8]. In 2005 experts at Ovum, an IT advisory company, estimated that there were over 200 billion lines of Cobol deployed in production [10]. They claimed this number was to continue to grow by between 3% and 5% a year. There is no sign that the Cobol's growth has stopped. Our own study revealed that in the production environment we investigated in a single year the number of Cobol modules grew by 2.9%.

Apart from changes to code there are also changes relating to the systems usage patterns. For instance, the MIPS capacity attributed to the one system in the studied portfolio has increased nearly 5 times in just 3 years time. For this particular system it was observed in time that many IMS transactions, which were linked to the system, entered the ranking of the top 100 executed IMS transactions.

The following are major elements involved in the ongoing monitoring of CPU resource usage in Phase II with respect to DB2 linked IMS transactions:

- Enforcement of the verification of the SQL code in the testing phase to eliminate the inefficient code from entering the production environment.
- Utilization of the mainframe usage data for identification of changes in the IMS transactions usage patterns.
- Deployment of a mechanism which enables measuring MSU consumption or CPU time spent on execution of particular SQL statements by the IMS transactions.

There are multiple possibilities to realize those elements. For instance, the verification of the SQL code prior to its entry to the production environment can be accomplished by conducting checks of the embedded SQL code as it has been proposed in this chapter. Our list of potentially inefficient programming constructs was derived based on the DB2 experts recommendations and the real-world examples taken from the MIPS-reduction project. The content of this list has been likely not exhausted. In fact, we would recommend an evolutionary approach here. Each time any kind of project which involves alterations to the mainframe applications code is carried out some increment to the knowledge on the inefficient code constructs could be made. For particular mainframe environments the insights based on organization's internal experience provide the most accurate information on the actual bottlenecks in the source code.

To enable analysis of the IMS transactions usage patterns it is important to assure that relevant mainframe usage data are collected. This is typically done by enabling adequate mechanisms on the z/OS. In case of the organization which provided us with the case study

the collection of the accounting data was well established. Partly due to the regulatory framework in which the business operates. Nevertheless, having the data available is not sufficient to take advantage of it. There must be clearly defined paths which allow exploitation of its abundance. For instance, ongoing measurement of the average MSU consumption by the transactions in the IMS environment would set up a foundation for enabling automated reporting of the deviations from the historical means.

Being able to capture the performance of execution of particular statements is somewhat more involving. It boils down to deployment of monitoring techniques which rely on the feedback from the operating system. z/OS provides facilities which enable capturing all sorts of detailed data bits through the so-called IFCID (Instrumentation Facility Counter ID) record blocks. For instance, IFCID 316 captures metrics with respect to SQL statements executed in DB2. Among these metrics we find information on the CPU time [67]. By capturing and storing the IFCID 316 data it is possible to deploy a mechanism to measure CPU usage by the particular SQL statements. Later on it is possible to use this data for performance analysis, and eventually planning improvements. Of course, all sorts of monitoring deployments must be carefully arranged with the mainframe experts to make sure that such mechanisms alone will not result in undesired CPU resource leaks.

3.8 Discussion

In this section we consider the source code based control of CPU resource usage in a broader spectrum of IT-management issues. First, we position our approach in the context of vendor management. In particular, we discuss how this domain of IT-management can benefit from the work presented in this chapter. Second, we discuss mainframe utilization from the perspective of potential redundancies. We focus on the usage of various environments within mainframes and discuss how to address the potential inefficiencies.

3.8.1 Vendor management

Trust is the foundation of any business venture. However, when the stakes are high additional means must be considered to assure success. For business, which is critically dependent on IT, software assets are priceless. Therefore, organizations which outsource IT activities to external parties must keep track of what happens. Lax or no control over outsourced IT is likely to lead to loss of transparency over its condition. It is possible to base evaluation of IT deliverables on data provided by the outsourcing vendor. However, such data is at clear risk of being distorted and not present a true picture. The bottom line is that it is in the best interest of the business management and shareholders to possess reliable means to control the state of IT.

Source code represents the nuts and bolts of the software assets. Maintaining its high quality is paramount since it has impact on numerous IT aspects. For instance, in our work we enabled the reduction of IT operational costs through improvements in the source code. We relied on the fact that inefficient code, or in other words, low quality code hampers the usage of hardware resource. Outsourcing IT frequently means outsourcing the alterations done to the source code. Of course, no CIO is expecting to degrade quality of the code

by outsourcing it. In order to prevent from such distortions to happen organizations must equip themselves with adequate quality assurance mechanisms.

Source code quality assurance at the level of vendor management typically boils down to two aspects: provisions in the service level agreements (SLAs) and methods for compliance verification. Provisions of an outsourcing SLA must stipulate in what condition the source code is expected to be delivered by the vendor. In order to make these conditions verifiable they better be expressed in source code terms. What we mean by that is a clear outline of the kind of code metrics, set of statements, or language dialects, to name a few, the delivered code should adhere to. This way it is possible to devise means to verify that the conditions of the SLA have been met.

Verification process can take two routes. The organization outsourcing its IT relies on the compliance assurances made by the vendor, or it chooses to verify the compliance on its own. We would recommend the latter. Having the source code quality conditions expressed in the source code terms makes it possible to deploy source code analysis techniques for verification. Such approach provides for the highest level of transparency on the source code. If the source code quality criteria are specified in such a way that lexical analysis of the code is sufficient than verification process is implementable in an inexpensive manner.

In our approach to reducing CPU resource usage we used the lexical code analysis to mark the potentially inefficient code in the Cobol files. In our case we focused only on the use of SQL. Of course, it is possible to check other aspects of the code following this very approach. The tooling we used for our purposes can be naturally deployed to verify SQL code in the process of controlling the outsourcing vendors.

A real-world example In the portfolio we study in this chapter we encountered a number of DB2-Cobol modules with instances of the WHRE$_x$ construct. We inspected all the modules which contain this construct in order to learn about the origins of the construct. We looked for the common properties these files shared. Since the number of modules was small it was feasible to analyze them manually. First, we analyzed the queries labeled with WHRE$_x$. As it turned out two thirds of the modules containing the queries originated from the implementation of the same system. We checked the queries for duplicates, which would suggest a *copy-paste* approach to programming, but out of 102 instances of the WHRE$_x$ we found only 1 case of a verbatim copy. This simple experiment suggests that the undesired WHRE$_x$ constructs did not propagate into the code through code duplications. Second, we scrutinized comments embedded in the 24 modules to track the history of changes. For 21 modules we found dates and identifiers of the programmers who were indicated as authors of those modules. We found 9 different authors among whom one was associated with as many as 13 modules. As it turned out the name of the author pointed to a company to which projects involving source code modifications were outsourced. It appeared as if the programmer (or programmers), who labeled these 13 modules with the name of the company, smuggled this coding malpractice into the portfolio. Presence of the WHRE$_x$ instances in the code and the comments embedded version history details expose interesting information. Of course, they neither prove that the WHRE$_x$ constructs were in the modules since their inception into the portfolio nor guarantee that the outsourcing company put them there. However, these observations

provide basis to raise questions as to the quality of work delivered by the vendor.

3.8.2 CPU resource leaks

Large organizations use mainframes to accommodate business processes with heavy workloads. There is a whole array of tasks that a typical mainframe accomplishes. Hosting of data warehouses to facilitate data mining, offering web-enabled services, processing batch-jobs, handling IMS transactions, storing data, to name a few. Each such task contributes in some degree to the overall mainframe usage bill. Within the mainframe we distinguish two types of workloads: production and non-production. The first type relates to all sorts of business critical tasks which require for processing the reliability offered by the mainframes. The second type refers to all other tasks which are executed on the mainframe but are not critical for the business processes or not necessarily require the top-notch execution performance. The second type of tasks constitutes the area with potential redundancies in MSU consumption. In other words, an area where CPU resources leak. Identification of such areas yields a way to further reduce mainframe utilization, and what follows, the related costs.

In this chapter we concentrated our analysis on the source code of IMS transactions executed in the production environment. All of them were highly critical for the business and without any doubt they had to be executed on the mainframe. On the mainframe the organization also developed and tested software which later ended up in production. Let us now illustrate the distribution of the MSU consumption among the environments designated for IMS. The MSUs were reported for four IMS transaction environments: PRODUCTION, TEST1, TEST2, and TEST3.

		Environment		
Month	**PROD**	**TEST1**	**TEST2**	**TEST3**
1	87.90%	10.82%	0.43%	0.85%
2	84.34%	14.52%	0.63%	0.51%
3	92.26%	6.02%	0.97%	0.74%
4	87.82%	10.62%	0.65%	0.90%
5	91.86%	6.35%	0.66%	1.13%
6	95.44%	3.03%	0.73%	0.80%
Monthly average	89.94%	8.56%	0.68%	0.82%

Table 3.13: Breakdown of the MSUs usage of the top 100 transactions into 4 IMS environments.

Table 3.13 shows the breakdown of the MSUs usage of the top 100 transactions into 4 IMS environments. In the first column we indicated the month number. In the remaining columns we provide percentages of the total monthly MSUs consumed in each IMS environment. In the last row the arithmetic average for each listed environment is provided to indicate an average monthly percentage of MSU consumption in the entire period.

Not surprisingly, the largest portion of the MSU consumption is associated with the production environment. On the average it accounts for nearly 90% of all MSUs con-

sumed. All the remaining environments consume roughly 10% with the `TEST1` environment consuming the majority of approximately 8%. We implicitly assumed that the workloads on the production environment are critical and cannot be removed. The workloads assigned to the test environments do not serve business processes in a direct way yet they take up approximately 10% of the IMS environments MSUs. These MSUs loom as an easy to eliminate target. An obvious approach would involve moving the test workloads onto environments which do not incur MSU related charges, such as PCs. It has been a reality in many IT shops since shortly after the invention of the PC to migrate from MSU consuming environments. Compilers, language parsers, test data generators, testing environments and source code libraries have all been available on PCs and servers for years. Many have syntax specifically geared to the IBM mainframe environment. In case of this portfolio migration of testing was complicated. So that this solution was not possible for this organization to implement. Nonetheless, insight like the one presented here certainly provides food for thought for mainframe cost reduction strategists.

3.9 Conclusions

Costs relating to mainframe usage are substantial. Despite this fact the CPU resource consumption is not managed in a granular fashion. We departed from the fact that the usage charges are directly dependent on the applications' code and leveraged the code level aspects up to the executive level of operational costs. In this chapter, we presented an approach to managing MIPS related costs in organizations which run their applications on mainframes.

Our approach relies on source code improvements relating to interaction between the applications and the database. Its distinct feature is that it does not require instrumentation of the mainframe running the applications, what allows eliminating risks that can jeopardize continuity of operations. Also, it allows obtaining the insight into the mainframe environment and conduct planning of code optimization projects without the actual need to access the mainframe. One of our assumptions was to be pragmatic so that facilitation of our approach in an industrial setting is feasible. We achieved it by relying on the type of data that a typical mainframe operating organization possesses: source code and the mainframe usage information. We showed that our approach is adequate to incorporate management of CPU resource usage at the portfolio level.

We investigated a production environment running 246 Cobol applications consisting of 23,004 Cobol programs and totalling to 19.7 million of physical lines of code. Approximately 25% of all the Cobol programs interacted with DB2. Our investigations were focused on the IMS-DB2 production environment. In particular, we studied the MSU consumption figures relating to the top 100 most executed IMS transactions. One of the characteristics of the IMS environment was that on a weekly basis between 72%–80% of all the major IMS transaction invocations involved database usage. As it turned out the MSU consumption for those transactions was on average higher by more than a half compared to the consumption reported for the non-database related invocations.

An earlier small scale MIPS-reduction project triggered our full-scale portfolio analysis. As the project showed, through SQL tunning it was possible to save approximately

9.8% of the annual cost linked to executing the optimized portion of the portfolio. Our approach enabled us to effectively constrain the search space for inefficient SQL code in a large set of source files. To achieve this we bridged the source code dimension with the financial dimension. We related the mainframe usage data to the source code implementing the IMS transactions.

The combination of SQL programing knowledge, findings from the MIPS-reduction project, and input from the experts gave us a set of syntactic rules which enable filtering out the potentially inefficient SQL constructs from the portfolio sources. We showed how to use those rules along with the code interdependencies information to narrow down the number of source files potentially worth optimization. With our approach we could identify as little as 0.78% of all modules as candidates for actual improvements. We presented our tooling in detail so that others can use our approach in their own context. As we showed the tooling is simple to implement.

We demonstrated two code improvement scenarios and calculated the potential reductions in MSU consumption. We showed that by selecting as little as 14 Cobol-DB2 modules there exists a possibility of cutting approximately 6.1% of the average monthly MSU consumption in the studied environment. By carrying out a more extensive code improvement project involving a potential 180 modules the savings can reach as much as 16.8%.

Our work presented here provides organizations with the following. First, it outlines a framework for incorporation of a structured approach to CPU resource management on the mainframe. Second, it provides for improvement of code quality through enforcement of usage of the SQL coding guidelines derived from a substantial real-world portfolio. Finally, it demonstrates how to use information obtained from source code analysis and mainframe usage data to plan projects aimed at reducing MSU consumption.

To conclude, identifying the few most promising Cobol modules that give opportunity to significantly reduce CPU resource consumption is a viable option for reducing operational IT costs. We provided an approach, tooling and an example to implement this. Our approach leads to fact-based CPU resource management. We hope that it will become a trigger for the IT executives to increase operational IT efficiency.

CHAPTER 4

Samenvatting

4.1 Broncode: een goudmijn van management informatie

Het onderzoek dat in dit proefschrift wordt beschreven is uitgevoerd onder de auspiciën van het EQUITY project. EQUITY staat als afkorting voor *Exploring Quantifiable IT Yields* en het project wordt gesubsidieerd door de Nederlandse Organisatie voor Wetenschappelijk Onderzoek (ZWO) en het Ministerie van Economische Zaken uit gelden die worden beheerd door de JACQUARD stichting. De doelstelling van EQUITY is een kwantitatieve basis te leggen onder de besluitvorming over IT. Veel beslissingen over de inzet en het onderhoud van IT middelen in het bedrijf worden thans gevoelsmatig genomen en ontberen een rationele (kwantitatieve) grondslag. In het bedrijfsleven is men zich nauwelijks bewust van het feit dat veel essentiële kwantitatieve management informatie ligt opgesloten in de broncode van de informatiesystemen die de bedrijfsactiviteiten ondersteunen. In dit proefschrift wordt verslag gedaan van twee toepassingen van broncode analyse die gericht is op het genereren van management informatie. In beide gevallen is gebruikgemaakt van praktijkdata en hebben we de resultaten van de case studies ter toetsing voorgelegd aan praktijkfunctionarissen.

Het hoeft nauwelijks betoog dat IT een onmisbare productiefactor is voor de meeste bedrijven en overheidsinstellingen voor het realiseren van hun organisatiedoelstellingen. De bekende IT service provider Gartner meldt dat het totale bedrag dat wereldwijd door het bedrijfsleven aan IT werd besteed in 2008 circa 3,4 biljoen euro bedroeg. De grote afhankelijkheid van de business van IT ondersteuning heeft niet alleen vergaande kosten consequenties, maar brengt ook grote risicos met zich mee. Voor veel bedrijven geldt dat de continuteit van de bedrijfsvoering ernstig in gevaar wordt gebracht indien bepaalde cruciale informatiesystemen langer dan een etmaal down gaan. Het onderstaande staatje geeft een goed beeld van de kostenimpact van een uur uitval van IT ondersteuning. Het overzicht is gemaakt door de Fiber Channel Industry Association en overgenomen uit [153]. Voor een aantal verschillende industrieën en bedrijfsactiviteiten is geschat hoe groot de

financiële schade is indien de ondersteunende IT systemen gedurende 1 uur niet beschikbaar zijn als gevolg van een hardware- of softwarestoring.

Business activity	Industry	Hourly downtime costs
brokerage operations	finance	$6,450,000
CC authorizations	finance	$2,600,000
pay-per-view	media	$150,000
home shopping (TV)	retail	$113,000
catalog sales	retail	$90,000
airline reservations	transportation	$90,000
tele-ticket sales	media	$69,000
package shipping	transportation	$28,000
ATM fees	finance	$14,500

Table 4.1: Information systems hourly downtime cost per business activity.

Zie dat de verliezen in sommige gevallen kunnen oplopen tot zes en een half miljoen dollar per uur! Indien men zich realiseert dat de IT systemen met grote regelmaat moeten worden aangepast en uitgebreid vanwege de steeds wisselende eisen die de omgeving (markt) aan de business stelt, dan is duidelijk dat een goed inzicht in de kosten, de kwaliteit en het aanpassingsvermogen van de bestaande IT- portfolio onontbeerlijk is. In dit proefschrift laten we zien dat de broncode van de informatiesystemen een schat aan informatie bevat die na een paar bewerkingsslagen aan het management kan worden aangeboden als basis voor besluitvorming. We laten zien hoe relevante feiten uit de broncode kunnen worden opgediept door analyse van compiler uitdraaien van recente productieversies van de informatiesystemen. Broncode analyse is op zich niet nieuw en kan bogen op een rijke ervaringsgeschiedenis. Voor onze doeleinden hebben we dan ook gebruikgemaakt van een aantal bekende geautomatiseerde hulpmiddelen voor broncode analyse. Onder meer hebben we gebruikgemaakt van gereedschap dat gebaseerd is op de zogenaamde lexicale analyse methode. Teneinde de analyse eenvoudig te houden en niet duurder te maken dan nodig hebben we in een aantal gevallen zelf analyse software geschreven. Deze kwam in de plaats van op de markt verkrijgbaar gereedschap dat voor onze analyse doeleinden een te zwaar en te duur hulpmiddel zou zijn geweest. Bij de uitvoering van de broncode analyse is het motto steeds geweest: maak het niet moeilijker en duurder dan nodig en volg een werkwijze die door andere bedrijven gemakkelijk te kopieren is.

Het grote voordeel van broncode als informatiebron is dat het een zuivere informatiebron is. Zuiver in de zin dat de feiten die aan het licht worden gebracht niet kunnen zijn gemanipuleerd. Dit in tegenstelling tot andere informatiebronnen die bestaan uit gegevens die door medewerkers zijn verzameld en ingevoerd. Bij het vastleggen van die gegevens kunnen persoonlijke waardeoordelen een belangrijke rol spelen. Vaak zijn de bestanden bovendien niet goed bijgehouden en daardoor onvolledig. Denk bijvoorbeeld aan documentatie betreffende functioneel onderhoud en uitbreiding van functionaliteit van systemen. Ook beschikken veel organisaties vaak niet over gegevens betreffende kosten en productiviteit bij systeemontwikkeling en onderhoud. Indien beschikbaar, dan is men vaak pas laat begonnen met het meten en vastleggen van dergelijke projectgegevens en is

tijdreeks analyse daardoor niet mogelijk. Voor informatiesystemen die lang geleden zijn gebouwd en een aantal keren zijn aangepast geldt vaak dat de systeemdocumentatie verloren is gegaan en de programmeurs die de bouw en onderhoudswerkzaamheden hebben verricht de dienst reeds lang hebben verlaten. Voor de broncode gelden deze bezwaren niet. De laatst bijgewerkte versie van de broncode is per definitie beschikbaar en vaak kan uit de commentaarregels die de programmeurs tijdens de bouw- en onderhoudswerkzaamheden aan de broncode hebben toegevoegd de historie van het systeemonderhoud voor het overgrote deel worden gereconstrueerd.

Zoals gezegd, doet dit proefschrift verslag van twee toepassingen van broncode analyse ten behoeve van het genereren van management informatie. In beide gevallen is gebruikgemaakt van praktijkdata en zijn de resultaten ter toetsing voorgelegd aan praktijkfunctionarissen. De praktijkdata is afkomstig van een internationaal opererende financiële instelling en bestaat uit de broncode van de informatiesystemen van een van haar grote business units. In hoofdstuk 3 wordt beschreven hoe we de plausibiliteit van de data hebben gecheckt door een aantal controles op de data uit te voeren. De testresultaten waren alle positief. We geven nu een aantal karakteristieken van de IT portfolio. De betreffende portfolio bestaat uit 47 informatiesystemen die in COBOL zijn geschreven en draaien op een mainframe. De 47 systemen bestaan in totaal uit 8198 modules (COBOL programmas). De code is voor ca 40% handmatig geschreven en voor het overige deel gegenereerd door COBOL codegeneratoren, zoals TELON, COOL:GEN en CANMAN. De totale code van de IT portfolio bestaat uit circa 18 miljoen regels. Voorwaar een astronomisch groot aantal! Het is duidelijk dat het niet doenlijk is om 18 miljoen regels code handmatig te analyseren, maar dat daarvoor geautomatiseerde hulpmiddelen nodig zijn. De oudste portfolio systemen stammen uit de jaren zestig van de vorige eeuw en de meest recente systemen zijn van 2009. De portfolio bestrijkt dus een tijdsperiode van vijftig jaar. In die periode hebben zich tal van ontwikkelingen op IT gebied voorgedaan die hun sporen in de broncode hebben achtergelaten. Zo heeft de programmeertaal COBOL een evolutie doorgemaakt. Als gevolg daarvan zijn vele verouderde, inefficiënte programmeerconstructies in de code te vinden. Ook de codegeneratoren hebben een ontwikkeling in de tijd doorgemaakt en de code die is gegenereerd door de ene codegenerator, verschilt van de andere. Een probleem is dat geen van de codegeneratoren kan worden afgestoten, zolang de portfolio nog systemen bevat waarvan modulen zijn gegenereerd door de codegenerator. De licentiekosten blijven doorlopen en de kennis over het gebruik van de codegenerator moet op peil worden gehouden. Vervanging van de gegenereerde code door handgeschreven code zou een oplossing zijn, maar is vaak een riskante en kostbare operatie, zeker als de software cruciale bedrijfsactiviteiten ondersteunt. Een ander probleem dat een gevolg is van de lange levensduur van de portfolio is de noodzaak verschillende COBOL compilers *up and running* te houden. Het besturingssysteem van het IBM mainframe is in de tijd sterk geëvolueerd. Van de meer dan acht duizend modulen waaruit de informatiesystemen zijn opgebouwd kan 40% alleen worden gecompileerd met de verouderde IBM OS/VS COBOL compiler, die niet meer door IBM wordt ondersteund. Niet alleen is de programmeertaal COBOL in de loop van de tijd sterk verbeterd en zijn steeds meer geavanceerde COBOL code generatoren beschikbaar gekomen, maar ook hebben de database management systemen en transactie monitoren een ontwikkeling doorgemaakt in de afgelopen 50 jaar. Voor gegevensverwerkende bedrijven is de opslag en integriteit-

bewaking van data in databases, het bijwerken van data en het efficiënt kunnen opvragen van data uit databases een cruciale bedrijfsactiviteit. Voor het opvragen van gegevens uit de database wordt gebruikgemaakt van speciaal daarvoor ontworpen opvraagtalen, zoals SQL, dat staat als afkorting voor Structured Query Language. Een efficiënt gebruik van SQL, gecombineerd met een efficiënte technische database infrastructuur, leidt tot grote kostenbesparingen en lage response tijden. Door verouderde en inefficiënt werkende SQL programmeerconstructies via broncode analyse op te sporen en vervolgens te optimaliseren, gebruikmakend van nieuwe inzichten en programmeerhulpmiddelen, kunnen grote besparingen worden gerealiseerd met een laag afbreukrisico voor de continuteit van de bedrijfsvoering.

We geven nu een samenvatting van de twee case studies die we hebben uitgevoerd. Bij beide case studies staat het distilleren van (kwantitatieve) management informatie uit de broncode centraal.

4.2 Case studie 1: strategisch IT management

De eerste case studie betreft IT besturing op strategisch niveau (vaak aangeduid met IT Governance). Een specifieke taak van de ondernemingsleiding is het ontwikkelen en uitdragen van een strategische visie op het toekomstige reilen en zeilen van de onderneming. Een belangrijk onderdeel daarvan is de visie op de inzet van IT voor het realiseren van de bedrijfsdoelstellingen. Op strategisch niveau gaat het om de beantwoording van vragen als:

- Wat is de economische waarde van de bestaande portfolio van IT systemen? Voor welk bedrag zouden deze kapitaalgoederen op de balans moeten worden opgevoerd?
- Welke kosten zullen in de toekomst bij ongewijzigd beleid zijn gemoeid met het in stand en operationeel houden van de informatiesystemen?
- Wat is de kwaliteit van de bestaande IT systemen? Hoe groot is het risico dat de uitvoering van essentiële bedrijfsactiviteiten stokt als gevolg van falende IT systemen? Kan dat risico worden verlaagd door herbouw of vervanging van de informatiesystemen die aan het risico debet zijn?

Dit soort vragen is aan de orde bij strategische management beslissingen over IT middelen. De ondernemingsleiding moet aangeven welke activiteiten voorrang krijgen bij het onderhoud en beheer van haar IT middelen.

In hoofdstuk 2 wordt uiteengezet hoe door analyse van broncode een antwoord kan worden gegeven op bovengenoemde typen management vragen. Als eerste wordt de vraag behandeld hoe de economische waarde van de portfolio van informatiesystemen kan worden bepaald. Door te schatten hoeveel de herbouw van de informatiesystemen vandaag de dag zou kosten (gebruikmakend van de meest geëigende productiewijze) wordt een goede indicatie gekregen van de vervangingswaarde van de portfolio. Met behulp van broncode analyse en gebruikmakend van de zogenaamde *backfiring* methode [80] [82, p. 79] wordt eerst een schatting gemaakt van de omvang van de bestaande informatiesystemen,

gemeten in functiepunten. Een functiepunt is een door de industrie en wetenschap algemeen geaccepteerde synthetische maat voor het meten van de omvang en complexiteit van een systeem. Zoals in de bouwwereld bij het opstellen van begrotingen de kubieke meterprijs wordt gehanteerd, zo kan men bij het opstellen van een kostenbegroting van een systeem ontwikkelproject uitgaan van de kosten per functiepunt. Vervolgens worden publieke *benchmark* formules gebruikt om op basis van de geschatte omvang van de informatiesystemen te schatten hoeveel kosten, tijd en risico er mee gemoeid zouden zijn om de systemen met de beschikbare kennis en hulpmiddelen van nu te bouwen. De benchmark formules worden in hoofdstuk 2 uitvoerig toegelicht en beargumenteerd wordt waarom ze mogen worden toegepast in ons specifieke geval. Om een beeld te krijgen van de verwachte aangroei van de portfolio in de toekomst wordt gebruikgemaakt van ervaringscijfers uit het verleden. Van een groot aantal modulen kan uit analyse van de broncode het groeipad worden gedestilleerd dat is doorlopen sedert de datum van hun creatie. In de loop van de tijd is de functionaliteit van de modulen uitgebreid en dat heeft geresulteerd in een toename van het aantal functiepunten. Door extrapolatie van deze groeicijfers wordt een redelijk betrouwbare schatting verkregen van de aangroei van de portfolio als totaal bij ongewijzigd beleid. Een jaarlijkse toename van het aantal functiepunten op portfolio niveau met 8,7% ligt in de lijn der verwachtingen. De kosten repercussies hiervan in termen van toename van onderhoud- en exploitatiekosten worden met behulp van publieke *benchmark* formules ingeschat.

Risico analyse is een chapiter apart. Zo blijkt uit broncode analyse dat 1237 modulen (ca 35% van alle handgeschreven modulen) het stempel complex verdient. Dat wil zeggen: voor 1237 modulen geldt dat de berekende McCabe indicator (een algemeen geaccepteerde maat om de complexiteit van software te meten [108]) meer dan 50 bedraagt, hetgeen volgens ervaring een niet te verwaarlozen risicofactor is bij onderhoud en vernieuwbouw. Duidelijk is dat het gebruik van verouderde Cobol compilers en codegeneratoren, behalve hogere kosten (licenties), ook extra risico met zich meebrengt. Uit onze broncode analyse blijkt dat veertig procent van de portfolio modulen voor compilatie afhankelijk is van de verouderde, niet meer door IBM ondersteunde IBM OS/VS COBOL compiler. De kennis over deze verouderde hulpmiddelen is normaliter schaars aanwezig, omdat de medewerkers die ermee hebben gewerkt de dienst hebben verlaten. Verder is duidelijk dat systemen van oude datum, die voor een belangrijk deel uit modulen bestaan die zijn geschreven in niet meer gebruikte dialecten van COBOL, een groter risico van falen lopen dan systemen van recentere datum. Indien zulke systemen dan ook nog met grote regelmaat worden aangepast (hoge volatiliteit) en cruciale bedrijfsactiviteiten ondersteunen, kan worden gesproken worden van een hoog risico gebied (hot spot) binnen de portfolio. In hoofdstuk 2 wordt beschreven hoe zulke *hot spots* met behulp van broncode analyse in kaart kunnen worden gebracht en kunnen worden afgebakend.

Met behulp van *what-if* scenario analyse wordt een beeld verkregen van het risico profiel van de portfolio in de toekomst bij ongewijzigd beleid. Ook worden met behulp van *what-if* scenario analyse mogelijke migratietrajecten, gericht op kostenverlaging en risicoverlaging, geëvalueerd. Daarbij worden de systemen gerangschikt in volgorde van kritisch belang voor de bedrijfsvoering. Een systeem wordt als bedrijfskritisch aangemerkt, indien het systeem de uitvoering van een bedrijfsactiviteit met een hoog transactievolume ondersteunt. Bij migratietrajecten gericht op kosten- en risicoverlaging blijken

de topkritische bedrijfsapplicaties in belangrijke mate te worden geraakt.

4.3 Case studie 2: verlaging operationele kosten door MIPS management

MIPS staat als afkorting voor Million Instructions Per Second. De kosten die het Rekencentrum de eindgebruiker in rekening brengt voor het draaien van een applicatie zijn voor het overgrote deel gebaseerd op het MIPS-verbruik door de applicatie. Duidelijk is derhalve dat het vanuit een kostenoogpunt loont te trachten het verbruik van het aantal MIPS te minimaliseren. Hoofdstuk 3 laat zien dat broncode analyse een belangrijk hulpmiddel is bij het opsporen van inefficiënt geschreven code, die debet is aan een hoog MIPS verbruik. Feitelijk is het de combinatie van programmatuur en database infrastructuur die de inefficiëntie veroorzaakt, maar door de *verdachte* programmeerconstructies op te sporen in de code wordt tegelijkertijd de schijnwerper gezet op mogelijke technische inefficiënties.

In de tweede case studie wordt ingezoomd op SQL code die voor IMS (Information Management System) transacties is geschreven om data op te vragen uit DB2 databases. DB2 is een ruim verspreid relationeel gegevenssysteem dat wordt gebruikt in de productieomgeving (massale transactieverwerking) van gegevensverwerkende bedrijven. In DB2 en andere relationele systemen wordt SQL (Structured Query Language) gebruikt als interface tussen de gegevensbank en toepassingsprogrammatuur en als zelfstandige vraagtaal. Ons onderzoek wijst uit dat wekelijks bij de uitvoering van 72% - 80% van de belangrijkste IMS transacties een database wordt geraadpleegd. De MIPS consumptie van deze transacties blijkt gemiddeld een factor 2 hoger te liggen dan die van transacties die geen database raadplegen. De vraagtaal SQL kent veel uitdrukkingen, waarvan gebruik kan worden gemaakt bij het opslaan, opvragen of bijwerken van gegevens. Wij hebben bij ons onderzoek de schijnwerper gezet op het SELECT statement. Binnen SELECT zijn WHERE en FROM voorbeelden van veelgebruikte operaties. De programmering in SQL luistert bijzonder nauw. Inefficiënte programmeerconstructies leiden snel tot een excessief MIPS verbruik bij de transactieverwerking. Indien het om massale transactieverwerking gaat kunnen de kosten daardoor hoog oplopen. In hoofdstuk 3 wordt beschreven hoe met behulp van eenvoudig uit te voeren broncode analyse *verdachte* SQL programmeerconstructies kunnen worden opgespoord en aan experts ter beoordeling kunnen worden aangeboden. Bij de broncode analyse wordt onder meer gebruikgemaakt van gereedschap dat is gebaseerd op de zogenaamde lexicale analyse methode. Zoals eerder gezegd gaat het hier om een porfolio die in het totaal een kleine twintig miljoen aan code regels omvat. Zonder geautomatiseerde hulpmiddelen is het ondoenlijk een dergelijk groot bestand te analyseren. Uit de analyse blijkt dat 668 van de 23004 modulen waaruit de informatiesystemen zijn opgebouwd SQL programmeerconstructies bevatten die mogelijk inefficiënt zijn geschreven. Een aantal van deze modulen heeft grote invloed op het MIPS verbruik, omdat ze bij hun uitvoering andere modulen aanroepen, die weer andere IMS transacties ondersteunen. Optimalisatie van de SQL code van deze modulen heeft dus potentieel een veel groter MIPS consumptie besparingseffect dan optimalisatie van de SQL code van modulen die weinig of geen relaties met andere modulen hebben en de uitvoering van transacties ondersteunen met een laag transactievolume. In hoofdstuk 3 wordt beschreven

hoe we met behulp van broncode analyse de aanroep (call) relaties tussen de modulen in kaart hebben gebracht. Het resultaat is onder meer samengevat in de vorm van een graaf. De potentiële besparing op het MIPS verbruik door optimalisatie van de SQL code wordt nog veel groter indien alle betrokken IMS transacties een groot omzetvolume kennen. Uiteindelijk kon het zoekgebied worden verkleind tot 0,78% van de 23000 COBOL programmas. We hebben de resultaten van onze *quick scan* geëvalueerd door ze te vergelijken met de resultaten van een kleinschalig MIPS besparingsproject dat eerder door het bedrijf was uitgevoerd. Met dit project werd een kostenbesparing van ca 3,8% op de maandelijkse MIPS kosten rekening gerealiseerd. Het bleek, dat alle SQL code die het team had herschreven (geoptimaliseerd), in onze scan als potentieel inefficiënt geprogrammeerde SQL code was onderkend. We hebben een schatting gemaakt van de kostenbesparingen die het team zou hebben kunnen realiseren, indien het team alle door ons gevonden potentieel inefficiënte SQL code onder ogen zou hebben gekregen en waar nodig had geoptimaliseerd. Het team had dan 180 modulen bekeken en een kostenbesparing van om en nabij 7% op de maandelijkse MIPS kostenrekening gerealiseerd. De belangrijkste conclusie die uit de tweede case studie kan worden getrokken is, dat door middel van broncode analyse op goedkope en snelle wijze een inventarisatie kan worden gemaakt van de SQL code die mogelijk inefficiënt is geschreven en daarom in aanmerking komt voor optimalisatie onderzoek. De uiteindelijke beoordeling is aan de SQL experts, maar veel tijd en kosten worden bespaard door de inzet van deze dure en schaarse expertise te richten op de meest veelbelovende activiteiten, waarvan het hoogste kostenbesparingsrendement kan worden verwacht.

Bibliography

[1] Advantage - Mainframe Pricing. Website. Available via http://www.ogs.state.ny.us/purchase/prices/7600021268PL_MainframeProducts.pdf.

[2] Meeting with Warren Buffett 28 Jan 05 - synthesized notes by topic. Available via http://www.tilsonfunds.com/BuffettVanderbiltnotes.pdf.

[3] MACRO 4. MACRO 4: Users 'slam' vendors for not helping to control mainframe processor consumption. *TradingMarkets.com*, January 2009. Available via http://www.tradingmarkets.com/.site/news/Stock%20News/2122225/.

[4] Joachim Ackermann, Miles Au Yeung, and Edwin van Bommel. Better IT management for banks. *The McKinsey Quarterly*, July 2007. Available via http://www.mckinseyquarterly.com.

[5] D.A. Adamo, S. Fabrizi, and M.G. Vergati. A Light Functional Dimension Estimation Model for Software Maintenance. In C. Verhoef, R. Kazman, and E. Chikofsky, editors, *IEEE EQUITY 2007: Postproceedings of the first IEEE Computer Society Conference on Exploring Quantifiable Information Technology Yields*. IEEE Computer Society, 2007.

[6] A.J. Albrecht. Measuring application development productivity. In *Joint SHARE/GUIDE/IBM Application Development Symposium*, pages 83–92, 1979.

[7] Nicolas Anquetil and Timothy C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance*, 11(3):201–221, 1999.

[8] Edmund C. Arranga, Ian Archbell, John Bradley, Pamela Coker, Ron Langer, Chuck Townsend, and Mike Wheatley. In Cobol's Defense. *IEEE Software*, 17(2):70–72,75, 2000.

[9] Kanika Bahadur, Driek Desmet, and Edwin van Bommel. Smart IT spending: Insights from European banks. *The McKinsey Quarterly*, January 2006. Available via `http://www.mckinseyquarterly.com`.

[10] Gary Barnett. The future of the mainframe, October 2005. Available via `http://store.ovum.com/Product.asp?tnpid=&tnid=&pid=33702&cid=0`.

[11] Michael W. Berry, Susan T. Dumais, and Gavin W. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Rev.*, 37(4):573–595, 1995.

[12] Bert Kersten and Chris Verhoef. IT Portfolio Management: A banker's perspective on IT. *Cutter IT Journal*, 16(4), 2003.

[13] Jesús Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. Legacy information systems: Issues and directions. *IEEE Software*, 16:103–111, 1999.

[14] Bluephoenix. CA Gen Modernization – Coolgen Migration. Available via `http://www.bphx.com/en/Solutions/ApplicationModernization/Pages/CooLGen.aspx`.

[15] Barry W. Boehm and Kevin J. Sullivan. Software economics: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 319–343, New York, NY, USA, 2000. ACM.

[16] Computer Associates (CA). Product Brief: CA TELON Application Generator, CA Telon Application Generator r5. Available via `http://www.ca.com/files/ProductBriefs/mp32418_telon_pb_us_en.pdf`.

[17] Computerworld. MIPS management at mainframe organizations, August 2007.

[18] Compuware. Reduce MIPS. Save money. Presentation available via `http://www.compuware-insight.com/pdfs/0307/Enterprise-MIPSManagementbrochure.pdf`.

[19] Compuware. Save money on mainframe hardware and software: Five simple steps to start managing MIPS, September 2008.

[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts London, England, 1990.

[21] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill Higher Education, 2006.

[22] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Enhancing an Artefact Management System with Traceability Recovery Features. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 306–315, Washington, DC, USA, 2004. IEEE Computer Society.

[23] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407, 1990.

[24] Carol Dekkers and Ian Gunter. Using 'backfiring' to accurately size software: more wishful thinking than science? *IT Metrics Strategies*, November 2000.

[25] Tom DeMarco. *Controlling software projects*. Yourdon Computing Series, Upper Saddle River, New Jersey, USA, 1982.

[26] Sheila Dennis and David Herron. FP lite - An alternative approach to sizing. Available from `http://www.davidconsultinggroup.com`.

[27] Jens Dibbern, Tim Goles, Rudy Hirschheim, and Bandula Jayatilaka. Information systems outsourcing: a survey and analysis of the literature. *SIGMIS Database*, 35(4):6–102, nov 2004.

[28] J. B. Dreger. *Function point analysis*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[29] S.T. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments and Computers*, 23:229–236, 1991.

[30] Susan T. Dumais. Enhancing Performance in Latent Semantic Indexing (LSI) Retrieval, 1992.

[31] Susan T. Dumais and Jakob Nielsen. Automating the assignment of submitted manuscripts to reviewers. In *SIGIR '92: Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 233–244, New York, NY, USA, 1992. ACM.

[32] Soumitra Dutta. Recognizing the True Value of Software Assets - Industry report. *Microfocus*, November 2007. `http://www.microfocus.com/000/RecognisingTheTrueValueofSoftwareAssets_tcm21-16042.pdf`.

[33] Mike Ebbers, Wayne O'Brian, and Bill Oden. *Introduction to the New Mainframe: z/OS Basics*. IBM's International Technical Support Organization, July 2006.

[34] eCube Systems LLC. Cool:Gen History. Available via `http://www.ecubesystems.com/coolgen.htm`.

[35] J. L. Eveleens and C. Verhoef. Quantifying IT forecast quality. *Science of Computer Programming*, 74(11-12):934–988, 2009.

[36] Brian Everitt and Torsten Hothorn. *A Handbook of Statistical Analyses Using R*. Chapman & Hall/CRC, 2006. ISBN 1-584-88539-4.

[37] Scott Fagen and David Luft. Optimizing Mainframe Success by Simplifying Mainframe Ownership. Available via `https://ca.com/Files/TechnologyBriefs/optimizing-mainframe-success-tb.pdf`.

[38] D. Faust and C. Verhoef. Software product line migration and deployment. *Software Practice and Experience, John Wiley & Sons, Ltd*, 33:933–955, 2003.

[39] Norman E. Fenton and Martin Neil. Software metrics: success, failures and new directions. *J. Syst. Softw.*, 47(2-3):149–157, 1999.

[40] Antonio Jose Ferreira. Performance tuning service for ibm mainframe. Available via `http://www.isys-software.com/TuningService(ajf).pps`.

[41] Micro Focus. No Respect: Survey Shows Lack Of Awareness, Appreciation For COBOL, May 2009. Available via `http://www.microfocus.com/aboutmicrofocus/pressroom/releases/pr20090528819202.asp`.

[42] Mark Fontecchio. CA aiming to ease mainframe software licensing costs, January 2008. Available via `http://searchdatacenter.techtarget.com/news/article/0,289142,sid80_gci1295640,00.html`.

[43] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19(3):214–230, 1993.

[44] David Garmus and David Herron. *Function Point Analysis: Measurement Practices for Successful Software Projects*. Addison-Wesley, 2001.

[45] P.R. Garvey. *Probability Methods for Cost Uncertainty Analysis – A Systems Engineering Perspective*. Marcel Dekker Inc., New York, 2000.

[46] Joris Van Geet and Serge Demeyer. Lightweight Visualisations of COBOL Code for Supporting Migration to SOA. In *Third International ERCIM Symposium on Software Evolution*, October 2007. to appear. available via `http://essere.disco.unimib.it/reverse/files/paper-re4apm/Paper10.05.pdf`.

[47] Tudor Girba, Stéphane Ducasse, and Michele Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 40–49, Washington, DC, USA, 2004. IEEE Computer Society.

[48] GNU. GNU Grep: Print lines matching a pattern. Technical report, GNU, 2010. http://www.gnu.org/software/grep/manual/.

[49] Robert B. Grady and Deborah L. Caswell. *Software metrics: establishing a company-wide program*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[50] Todd L. Graves, Alan F. Karr, J.S. Marron, and Harvey Siy. Predicting fault incidents using software change history. *IEEE Transactions on software engineering*, XX, 1999.

[51] Tom Groenfeldt. Morgan Stanley Takes On Big Data With Hadoop. Forbes.com LLC, May 2012. Available via `http://www.forbes.com/sites/tomgroenfeldt/2012/05/30/morgan-stanley-takes-on-big-data-with-hadoop/`.

[52] Francesco Guerrera. Citigroup looks to slash tech costs. *FT.com, Financial Times*, May 2009. Available via http://www.ft.com/cms/s/0/24808782-462b-11de-803f-00144feabdc0.html.

[53] Alex Handy. COBOL's future lies in the clouds. *SD Times on the WEB*, July 2009. Available via http://www.sdtimes.com/COBOL_S_FUTURE_LIES_IN_THE_CLOUDS/By_Alex_Handy/About_CLOUDCOMPUTING_and_COBOL/33587.

[54] Stefan Hinz, Paul DuBois, Jonathan Stephens, Martin 'MC' Brown, and Anthony Bedford. MySQL Reference Manual, 2009. http://dev.mysql.com/doc/.

[55] Starling David Hunter. Information Technology, Organizational Learning, and the Market Value of the Firm. Working papers 4418-03, Massachusetts Institute of Technology (MIT), Sloan School of Management, Aug 2003.

[56] IBM. IBM - DB2 Express-C. Available via http://www-01.ibm.com/software/data/db2/express/.

[57] IBM. STROBE for IBM DB2. Available via http://www-304.ibm.com/jct09002c/gsdod/solutiondetails.do?solution=6769&expand=true&lc=en.

[58] IBM. COBOL on the z/OS, OS/390, MVS, and VM Platforms. Website, February 1997. Available via http://www-03.ibm.com/servers/eserver/zseries/zos/le/history/cobmvs.html.

[59] IBM. IBM Rational Asset Analyzer Version 5.5. Technical report, IBM Corporation, July 2005. Available via http://publib.boulder.ibm.com/infocenter/rassan/v5r5/index.jsp?topic=/com.ibm.raa.doc/common/ccyccom.htm.

[60] IBM. Enterprise COBOL for z/OS Compiler and Runtime Migration Guide. Technical report, International Business Machines Corporation, 2007. Available via http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/index.jsp?topic=/com.ibm.entcobol.doc_4.1/MG/igymch1025.htm.

[61] IBM. *DB2 Universal Database for z/OS Version 8 – Administration Guide*. IBM, June 2009. Available via http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db2.doc.admin/dsnagj19.pdf?noframes=true.

[62] IBM. DB2 Version 9.1 for z/OS – SQL Reference. Technical report, International Business Machines Corporation, May 2009. Available via http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db29.doc.sqlref/dsnsqk16.pdf?noframes=true.

[63] IBM. DB2 Version 9.1 for z/OS, Application Programming and SQL Guide. Technical report, International Business Machines Corporation, June 2009. Available via http://publib.boulder.ibm.com/epubs/pdf/dsnapk14.pdf.

[64] IBM. DB2 Version 9.1 for z/OS: Performance Monitoring and Tuning Guide. Technical report, International Business Machines Corporation, October 2009. Available via http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db29.doc.perf/dsnpfk16.pdf?noframes=true.

[65] IBM. Enterprise COBOL for z/OS V4.2 Language Reference. Technical report, International Business Machines Corporation, August 2009.

[66] IBM. IBM DB2 Universal Database – SQL Reference Version 8. Technical report, International Business Machines Corporation, June 2009. Available via http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db2.doc.sqlref/dsnsqj17.pdf?noframes=true.

[67] IBM. IBM Tivoli OMEGAMON XE for DB2 on z/OS, Version 4.2.0. Technical report, International Business Machines Corporation, 2009. Available via http://publib.boulder.ibm.com/infocenter/tivihelp/v15r1/index.jsp?topic=/com.ibm.omegamon.xe_db2.doc/ko2rrd20228.htm.

[68] IBM. Introduction to DB2 for z/OS. Technical report, International Business Machines Corporation, May 2009. Available via http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db29.doc.intro/dsnitk13.pdf?noframes=true.

[69] IBM. DB2 Version 9.1 for z/OS: Performance Monitoring and Tuning Guide. Technical report, International Business Machines Corporation, March 2010. Available via http://publib.boulder.ibm.com/epubs/pdf/dsnpfk17.pdf.

[70] IBM. z/OS: Resource Measurement Facility Performance Management Guide. Technical Report V1R12.0, International Business Machines Corporation, March 2010. Available via http://publibz.boulder.ibm.com/epubs/pdf/erbzpm90.pdf.

[71] IBM. IBM DB2 Universal Database – SQL Reference Version 7. Technical report, International Business Machines Corporation, db2sql. Available via ftp://ftp.software.ibm.com/ps/products/db2/info/vr7/pdf/letter/db2s0e71.pdf.

[72] IFPUG. Function Point Counting Practices Manual, Release 4.1. Technical report, International Function Point Users Group (IFPUG), Mequon, Wisconsin, USA, January 1999.

[73] Gartner Inc. Gartner Says Worldwide IT Spending on Pace to Decline 6 Percent in 2009, July 2009. Available via http://www.gartner.com/it/page.jsp?id=1059813.

[74] The MathWorks Inc. Matlab. Technical report, The MathWorks, Inc., 2010. Available via http://www.mathworks.com/access/helpdesk/help/techdoc/.

[75] Software Engineering Institute. *C4 Software Technology Reference Guide–A Prototype*. Carnegie Mellon University, 1997. Handbook CMU/SEI-97-HB-001.

[76] ISO/IEC. Information technology Programming languages COBOL. Technical report, ISO/IEC 1989:2002(E), 2002.

[77] Jaap Bloem and Menno Van Doorn and Piyush Mittal. *Making IT Governance Work in a Sarbanes-Oxley World*. John Wiley and Sons, Inc., 2006.

[78] Capers Jones. Software metrics. *Computer*, pages 98–101, September 1994.

[79] Capers Jones. *Patterns of Software Systems Failure and Success*. International Thomsom Computer Press, Boston, MA, 1996.

[80] Capers Jones. Programming Languages Table, Release 8.2, March 1996. `http://www.spr.com`.

[81] Capers Jones. *The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences*. Addison-Wesley, ACM Press, 1998.

[82] Capers Jones. *Software Assessments, Benchmarks, and Best Practices*. Addison Wesley Information Technology Series, 2000.

[83] Larry Kahm. Building an Early Warning System to Enable COBOL Compiler Migration. Available via `http://www.heliotropicsystems.com/pubs/HSTSa112007.pdf`.

[84] P. Kampstra and C. Verhoef. Reliability of function point counts. Available via `http://www.cs.vu.nl/~x/rofpc/rofpc.pdf`.

[85] R.S. Kaplan and D.P. Norton. *The Balanced Scorecard Translating Strategy into Action*. Harvard Business School Press, 1996.

[86] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, and Katsuro Inoue. MUDABlue: an automatic categorization system for open source repositories. *J. Syst. Softw.*, 79(7):939–953, 2006.

[87] Rick Kazman, Haruka Nakao, and Masa Katahira. Practicing What is Preached: 80-20 Rules for Strategic IV&V Assessment. In C. Verhoef, R. Kazman, and E. Chikofsky, editors, *IEEE EQUITY 2007: Postproceedings of the first IEEE Computer Society Conference on Exploring Quantifiable Information Technology Yields*. IEEE Computer Society, 2007.

[88] Chris F. Kemerer. Reliability of function points measurement: a field experiment. *Commun. ACM*, 36(2):85–97, 1993.

[89] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[90] T.A. Kirkpatrick. Research: CIOs speak on ROI. *CIO Insight*, 1(11), 2000. Available via `http://www.cioinsight.com`.

[91] Hans-Bernd Kittlaus and Peter N. Clough. *Software Product Management and Pricing: Key Success Factors for Software Organizations*. Springer, January 2009.

[92] P. Klint and C. Verhoef. Evolutionary software engineering: a component-based approach. In *Proceedings of the IFIP TC2 WG2.4 Working Conference on Systems Implementation 2000 : languages, methods and tools*, pages 1–18, London, UK, UK, 1998. Chapman & Hall, Ltd.

[93] Paul Klint and Chris Verhoef. Enabling the creation of knowledge about software assets. *Data Knowl. Eng.*, 41(2-3):141–158, 2002.

[94] A. S. Klusener and C. Verhoef. 9210: The Zip Code of Another IT-Soap. *Software Quality Control*, 12(4):297–309, 2004.

[95] Steven Klusener, Ralf Lammel, and Chris Verhoef. Architectural Modifications to Deployed Software. *Science of Computer Programming*, 54:143–211, 2005.

[96] R. L. Krikhaar. *Software Architecture Reconstruction*. PhD thesis, University of Amsterdam, The Netherlands, 1999.

[97] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[98] Erald Kulk. *IT Risks in Measure and Number*. PhD thesis, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands, November 2009.

[99] G. P. Kulk and C. Verhoef. Quantifying requirements volatility effects. *Science of Computer Programming*, 72(3):136–175, 2008.

[100] Łukasz Kwiatkowski and Chris Verhoef. Reducing operational costs through MIPS management. Available via `http://www.cs.vu.nl/~x/mips/mips.pdf`.

[101] Łukasz M. Kwiatkowski. Reconciling Unger's parser as a top-down parser for CF grammars for experimental purposes. Master's thesis, Vrije Universiteit Amsterdam, The Netherlands, August 2005.

[102] Ralf Lämmel and Chris Verhoef. Cracking the 500-language problem. *IEEE Software*, 18(6):78–88, 2001.

[103] Ralf Lämmel and Chris Verhoef. VS COBOL II grammar Version 1.0.4. Technical report, Vrije Universiteit Amsterdam, 2002.

[104] Johan Laurenz Eveleens and Chris Verhoef. The rise and fall of the chaos report figures. *IEEE Software*, 27(1):30–36, 2010.

[105] Ted MacNeil. Don't Be Misled By MIPS. November 2004. Available via `http://www.ibmsystemsmag.com/mainframe/tipstechniques/systemsmanagement/Don-t-Be-Misled-By-MIPS/`.

[106] Jonathan I. Maletic and Andrian Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *ICTAI '00: Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence*, page 46, Washington, DC, USA, 2000. IEEE Computer Society.

[107] Thomas J. McCabe and Charles W. Butler. Design complexity measurement and testing. *Commun. ACM*, 32(12):1415–1425, 1989.

[108] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-12(3):308–320, 1976.

[109] META Group. The Business of IT Portfolio Management: Balancing Risk, Innovation, and ROI. Technical report, META Group, Stamford, CT, USA, January 2002.

[110] Ian Moor. An Introduction to SQL and PostgreSQL. Available via `http://www.doc.ic.ac.uk/lab/labman/postgresql/x101.html`.

[111] F. Mosteller and J.W. Tukey. *Data Reduction and Regression*. Addison-Wesley, 1977.

[112] Müller H. and Wong K. and Tilley S. Understanding software systems using reverse engineering technology. *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)*, 26(4):41–48, 1994.

[113] NIST/SEMATECH. e-Handbook of Statistical Methods, April 2012. Available via `http://www.itl.nist.gov/div898/handbook/`.

[114] NIST/SEMATECH. e-Handbook of Statistical Methods: How Does Exploratory Data Analysis differ from Classical Data Analysis?, April 2012. Available via `http://www.itl.nist.gov/div898/handbook/eda/section1/eda12.htm`.

[115] L. O'Brien, C. Stoermer, and C. Verhoef. Software architecture reconstruction: Practice needs and current approaches. Technical Report CMU/SEI-2002-TR-024, Software Engineering Institute, 2002.

[116] Basel Committee on Banking Supervision. *International Convergence of Capital Measurement and Capital Standards, A revised framework*. Bank for International Settlements, Basel, Switzerland, June 2004.

[117] Andy Oppel and Robert Sheldon. *SQL: a beginner's guide*. McGraw-Hill Osborne, 3rd Edition, 2008.

[118] Hewlett Packard. HP 3000 Manuals: HP COBOL II/XL Reference Manual.

[119] Ryan Paul. Exclusive: a behind-the-scenes look at Facebook release engineering. Published online: arstechnica.com, April 2012. Available via `http://arstechnica.com/business/2012/04/exclusive-a-behind-the-scenes-look-at/-facebook-release-engineering/`.

[120] M.C. Paulk, C.V. Weber, B. Curtis, and M.B. Chrissis. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley Publishing Company, Reading, MA, 1995.

[121] R. J. Peters and C. Verhoef. Quantifying the yield of risk-bearing IT-portfolios. *Science of Computer Programming*, 71(1):17–56, 2008.

[122] D.R. Pitts. Metrics: Problem Solved? *Crosstalk: The Journal of Defense Software Engineering*, 1997. Available via `www.stsc.hill.af.mil/crosstalk/1997/dec/metrics.asp`.

[123] P.H. Porter. Revising R & D program budgets when considering funding curtailment with a Weibull Model. Master's thesis, Air University, Air Force Institute of Technology, Wright-Patterson Air Force, Ohio, USA, March 2001.

[124] H. Rubin and M. Johnson. What's Going On in IT? – Summary of Results from the Worldwide IT Trends and Benchmark Report, 2002. Technical report, Metagroup, 2002. Available via: `metricnet.com/pdf/whatIT.pdf`.

[125] Gerard Salton and Chris Buckley. Term Weighting Approaches in Automatic Text Retrieval. Technical report, Ithaca, NY, USA, 1987.

[126] Don Schricker. Cobol for the next millennium. *IEEE Software*, 17(2):48–52, 2000.

[127] Alex Sellink, Harry Sneed, and Chris Verhoef. Restructuring of COBOL/CICS legacy systems. *Science of Computer Programming*, 45:193–243, 2002.

[128] Alex Sellink and Chris Verhoef. Reflections on the Evolution of COBOL. Technical report, 1997. Available via `http://www.cs.vu.nl/~x/lib/lib.html`.

[129] Alex Sellink and Chris Verhoef. An architecture for automated software maintenance. In *Proceedings of the Seventh International Workshop on Program Comprehension*, pages 38–48. IEEE Computer Society Press, 1999.

[130] Alex Sellink and Chris Verhoef. Generation of software renovation factories from compilers. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, pages 245–255, Washington, DC, USA, 1999. IEEE Computer Society.

[131] Alex Sellink and Chris Verhoef. Scaffolding for software renovation. In *CSMR '00: Proceedings of the Conference on Software Maintenance and Reengineering*, page 161, Washington, DC, USA, 2000. IEEE Computer Society.

[132] Harvard Business Review Analytic Services. Unlocking the Value of the Information Economy. Available via `http://www.save9.com/wp-content/uploads/2010/04/HBR_Symantec_report_Unlocking_Value_of_Info_Economy.pdf`.

[133] Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Softw. Eng. J.*, 3(2):30–36, 1988.

[134] Lee Siegmund. An Intuitive Approach to DB2 for z/OS SQL Query Tuning. *IBM Systems Magazine (www.ibmsystemsmag.com)*, November/December 2006.

[135] Lee Siegmund. DB2 for z/OS SQL Query Tuning, Continued. *IBM Systems Magazine (www.ibmsystemsmag.com)*, January/February 2007.

[136] Harry M. Sneed and Peter Brössler. Critical success factors in software maintenance-a case study. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 190, Washington, DC, USA, 2003. IEEE Computer Society.

[137] SOFTMAINT. FAST for COOL:Gen: eliminate the license costs. Available via `http://www.softmaint.com/solutions/architecture-modernization/fast-for-coolgen/`.

[138] Canam Software. Canam Composer: overview. Available via `http://www.canamsoftware.com/product/report_comp/index.html`.

[139] TSG Software. CA-Telon. Available via `http://www.tsg.co.uk/html/ca-telon.html`.

[140] SRDI. *COBOL 85 Language Reference*. SRDI, 2000.

[141] Sterling Software, Inc. Sterling Software Ships Key:Enterprise 4.1; Company Adds New Features, Including Metamodel Extensibility, Improves Team Development Options and Publishes New Third-Party API. Available via `http://www.thefreelibrary.com/Sterling+Software+Ships+Key:Enterprise+4.1%3B+Company+Adds+New...-a017815694`.

[142] C. Stoermer, F. Bachmann, and C. Verhoef. SACAM: The software architecture comparison analysis method. Technical Report CMU/SEI-2003-TR-006, Software Engineering Institute, 2003.

[143] C. Stoermer, L. O'Brien, and C. Verhoef. Practice patterns for architecture reconstruction. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 151, Washington, DC, USA, 2002. IEEE Computer Society.

[144] Christoph Stoermer, Liam O'Brien, and Chris Verhoef. Moving towards quality attribute driven software architecture reconstruction. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 46, Washington, DC, USA, 2003. IEEE Computer Society.

[145] P.A. Strassmann. *Information Payoff – The Transformation of Work in the Electronic Age*. The Information Economics Press, New Canaan, Connecticut, USA, 1985.

[146] P.A. Strassmann. The Policies and Realities of CIM – Lessons Learned. In *Proceedings of the 4th Armed Forces Communications and Electronics Association Conference*, pages 1–19. AFCEA, Fairfax, VA, USA, 1993.

[147] P.A. Strassmann. *The Squandered Computer – Evaluating the Business Alignment of Information Technologies*. The Information Economics Press, New Canaan, Connecticut, USA, 1997.

[148] P.A. Strassmann. *Information Productivity – Assessing the Information Management Costs of US Industrial Corporations*. The Information Economics Press, New Canaan, Connecticut, USA, 1999.

[149] Paul Strassmann. Will big spending on computers guarantee profitability? Website, February 1997. Datamation `http://www.strassmann.com/pubs/datamation0297/` (Accessed in 09/08/2007).

[150] Paul A. Strassmann. Selected Information Economics Metrics, 2007. IEEE EQUITY 2007 keynote presentation (Available via `http://www.strassmann.com/pubs/gmu/2007-03-20-slides.pdf`).

[151] Andrey A. Terekhov and Chris Verhoef. The realities of language conversions. *IEEE Software*, 17(6):111–124, November 2000. Available via `http://www.cs.vu.nl/~x/cnv/s6.pdf`.

[152] The Economist Intelligence Unit. Business resilience: Ensuring continuity in a volatile environment., 2007. Available via: `http://a330.g.akamai.net/7/330/25828/20070329194828/graphics.eiu.com/files/ad_pdfs/eiu_Bus_Resilience_wp.pdf`.

[153] J.W. Toigo. *Disaster Recovery Planning – Strategies for Protecting Critical Information Assets*. Prentice Hall, 2000.

[154] J.W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

[155] M. G. J. van den Brand, P. Klint, and C. Verhoef. Reverse engineering and system renovation—an annotated bibliography. *ACM SIGSOFT Software Engineering Notes*, 22(1):57–68, 1997.

[156] Mark G. J. van den Brand, Paul Klint, and Chris Verhoef. Core technologies for system renovation. In *SOFSEM '96: Proceedings of the 23rd Seminar on Current Trends in Theory and Practice of Informatics*, pages 235–254, London, UK, 1996. Springer-Verlag.

[157] M.G.J. van den Brand, P. Klint, and C. Verhoef. Term rewriting for sale. *Electronic Notes in Theoretical Computer Science*, 15:218–241, 1998. Available via `http://www.cs.vu.nl/~x/sale/sale.html`.

[158] Pieter van der Spek, Steven Klusener, and Pierre van de Laar. Complementing software documentation: Testing the effectiveness of parameters for latent semantic indexing. Available via `http://www.cs.vu.nl/~pvdspek/files/complementing.pdf`.

[159] Arie van Deursen, Paul Klint, and Chris Verhoef. Research issues in the renovation of legacy systems. In *FASE '99: Proceedings of the Second International Conference on Fundamental Approaches to Software Engineering*, pages 1–21, London, UK, 1999. Springer-Verlag.

[160] Arie van Deursen and Tobias Kuipers. Rapid system understanding: Two cobol case studies. In *IWPC'98*, pages 90–97, 1998.

[161] N. Veerman. Automated mass maintenance of a software portfolio. *Science of Computer Programming*, 62(3):287–317, 2006.

[162] N. Veerman and E. Verhoeven. Cobol minefield detection. *Software: Practice & Experience*, 36(14):1605–1642, 2006.

[163] Niels Veerman. *Automated mass maintenance of software assets*. PhD thesis, Vrije Universiteit Amsterdam, The Netherlands, 2007.

[164] W. N. Venables, D. M. Smith, and the R Development Core Team. An Introduction to R – Notes on R: A Programming Environment for Data Analysis and Graphics. Technical report, 2010. Available via `http://cran.r-project.org/doc/manuals/R-intro.pdf`.

[165] Chris Verhoef. Software strong as a dyke. Available via `http://www.cs.vu.nl/~x/sil/sil.pdf`.

[166] Chris Verhoef. Towards automated modification of legacy assets. *Annals of Software Engineering*, 9(1–4):315–336, May 2000.

[167] Chris Verhoef. Quantitative IT Portfolio Management. *Science of Computer Programming*, 45(1), 2002.

[168] Chris Verhoef. Managing Multi-Billion Dollar IT Budgets using Source Code Analysis. *Third IEEE International Workshop on Source Code Analysis and Manipulation, Keynote speech*, 2003.

[169] Chris Verhoef. Quantifying the value of IT-investments. *Science of Computer Programming*, 56(3):315–342, 2005.

[170] Chris Verhoef. Quantitative aspects of outsourcing deals. *Science of Computer Programming*, 56(3):275–313, 2005.

[171] Chris Verhoef. Quantifying the effects of IT-governance rules. *Science of Computer Programming*, 67(2–3):247–277, 2007.

[172] L. Wall, T. Christiansen, and R.L. Schwartz. *Programming Perl, 2nd Edition*. O'Reilly & Associates, Inc., 1996.

[173] L. Wall and R.L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 1991.

[174] F. Wild, C. Stahl, G. Stermsek, and G. Neumann. Parameters Driving Effectiveness of Automated Essay Scoring with LSA. In Myles Danson, editor, *Proceedings of the 9th International Computer Assisted Assessment Conference (CAA)*, pages 485–494, Loughborough, UK, July 2005. Professional Development.

[175] Cheung Y., Willis R., and Milne B. Software benchmarks using function point analysis. *Benchmarking: An International Journal*, 6(3):269–276, 1999.

[176] Haiping Zhao. HipHop for PHP: Move Fast. Published online: developers.facebook.com, February 2010. Available via `https://developers.facebook.com/blog/post/2010/02/02/hiphop-for-php--move-fast/`.

[177] Marcin Zukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage*. PhD thesis, Universiteit van Amsterdam, 2009. ISBN 978-90-9024564-5.

[178] Nicholas Zvegintzov. Frequently Begged Questions and How To Answer Them. *IEEE Software*, pages 93–96, March/April 1998.