

Recovering management information from source code

L.M. Kwiatkowski & C. Verhoef

Department of Computer Science, Vrije Universiteit Amsterdam

De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

{lukasz,x}@cs.vu.nl

Abstract

IT has become a production means for many organizations and an important element of business strategy. Even though its effective management is a must, reality shows that this area still remains in its infancy. IT management relies profoundly on relevant information which enables risk mitigation or cost control. However, the needed information is either missing or its gathering boils down to daunting tasks. We propose an approach to recovery of management information from the essence of IT; the software's source code. In this paper 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 eventually utilize 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. 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.

Keywords: IT-portfolio management; management information; source code analysis; lexical analysis; Latent Semantic Indexing; LSI; source code comments; compilers; obsolete language constructs; volatility; vendor locks; legacy systems; operational risk; technology risk; risk mitigation; cost control; market value; scenario analysis; IT assets; IT metrics; automated data extraction; information retrieval; Cobol; case study;

1 Introduction

Information technology plays an important role in many organizations. World-wide IT related expenditure has been on the rise. Gartner reports that in 2008 spending on information technology was at \$3.4 trillion [39]. This means a three-fold growth since 1996 when the spending was estimated at \$1.076 trillion [94]. 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 [7]. 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 [7]. Later the Sarbanes-Oxley (SOX) Act of 2002 aligned IT governance with corporate governance hence requiring from IT investment decisions transparency [43]. 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 [68]. One aspect of mature decision making is access to management information [15, 51]. 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 [56]. 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 [22]. According to Jones' data 75% of IT organizations worldwide 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 [72]. 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 [75]. 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 [73]. One reason for the lack of IT measurement programs is the difficulty with the collection of relevant data [10]. The sheer size of the organizations results in data being widely spread among different people and databases, and hence difficult to obtain [83]. 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 [8, 78, 100]. 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 [107].

Source code The term IT-portfolio embraces various entities relating to IT such as projects, software, hardware, licenses, people (knowledge), networks, and more. In this paper we focus on the IT systems and treat them as organizations' assets which are an inseparable part of business operations. Source code constitutes the nuts and bolts of the IT systems. 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 [34]. 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 [47, 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 paper 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 [5]. 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 [27]. Cobol constantly evolves and adapts itself to the latest technology trends [77, 42]. 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 paper In this paper 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). We exploit the potential of the concealed data which resides in the source code statements, source comments, and also compiler listings. 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 analyzes 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 [109, 108, 32]. To handle the large amounts of data extracted from the source code we used MySQL database engine [35]. 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 [40, 104]. On the basis of a number of examples we demonstrate how deployment of our approach helps in embracing various IT risks and costs. To assure 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. Our technical framework was used also in achieving as much as 16.8% MIPS cost reduction in another large industrial case study. We reported on that effort elsewhere [61].

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 manage-

ment 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 analyzes 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.

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 paper 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 [4, 59, 70, 91, 92, 93]. Much work is devoted to automated software modifications [58, 102, 96, 97, 82, 103, 55]. The related work also focuses on the cost aspect which in the industrial setting plays a paramount role [25, 80, 105]. For instance, in [57] the authors emphasize on the cost reduction factor that code analysis adds to a software transformation project. We already presented elsewhere [61] 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 paper 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 [81, 98, 62]. 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 [79]. Analysis involving grammar based parsing is especially difficult. We propose in this paper 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 [17]. 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 analyzes 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 [17] the authors analyzed two systems of about 200k lines of code (LOC) from a portfolio of banking systems, or in [30] 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 [61, 85, 101] 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 paper This paper is organized as follows. In Section 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 analyzes. Section 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 4. In Section 5 we present how to recover information from the code extracted data bits. In Section 6 we illustrate how to use the bit-level data to obtain board-level insights. Finally, in Section 7 we conclude our work and summarize findings.

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 source. We are also indebted to the employees of their IT department for providing us with answers to our questions. We thank Patrick Bruinsma, from IBM Netherlands, for providing us with comments and expertise with regards to mainframe environments. Last but not least, we are very grateful to our colleague Rob Peters for meticulously

reviewing this paper several times.

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 paper. Finally, we discuss which source code parts we use to fuel the process of management information recovery.

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 attaining it. To reach for the information one must collect the relevant data, analyze them and properly interpret.

To support IT decisions it is often sufficient to find answers to relatively simple questions relating to the characteristics of the IT-portfolio. However, most of the time the solid answers are missing. In [112] 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?
- How good is maintenance?

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 [112].

In our approach we strive for obtaining reliable data. To support IT executives in challenging questions like those posed above and other, we propose

in this paper to analyze source code of the IT-portfolio. We argue that source code is the prime unbiased source of data that enables gaining lucidity in the condition of IT. It constitutes the nuts and bolts of the information systems. Therefore, by reaching for it we are able to derive characteristics of the information systems 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.

In its nature source code is a repository of various textual data. The questions posed by Zvegintzov provide us with the top level guidelines which determine what we will look at when analyzing source code. For instance, the demand to establish the amount of software in the portfolio immediately tells us that we need to decide upon some size measure. Lines of code or module counts are the most natural candidates which already serve the purpose and can be collected during code analysis. For other questions there will be other elements in the code worth extracting. We will elaborate on this later on.

To illustrate the mechanics behind our approach we obtained numerous insights into the portfolio. We formulated our insights through histograms, density plots, tables, etc., and analyzed them in the context of code quality, risks for maintainability, cost drivers, and other areas which are important for the decision makers. We want to emphasize that the insights we present in the remainder of the paper do not exhaust all possible views that IT decision makers may opt for. For instance, in [61] we presented on the basis of a large-scale portfolio how to obtain other insights and reduce MIPS usage. The goal of the work presented here is to show how source code analysis can be incorporated into IT-management practice.

2.2 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 1 we present some of its characteristics.

Let us explain the content of Table 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

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 1: Characteristics of the IT-portfolio.

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 paper we will elaborate on how we processed this huge amount of source code and recovered more than the basics listed above.

2.3 Source code facts

To structurally present our approach towards analysis of the portfolio we introduce two concepts we will use throughout this paper: 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 [64, 42, 89]. 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 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 1: Fragment of a Cobol program containing the *documentation statements*.

The first line in Figure 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, AUTHOR, INSTALLATION, DATE-COMPILED, and REMARKS. Each of which is

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 3: Fragment of a compiler listing containing compiler appended meta-data.

To illustrate how the data is expressed in the studied portfolio we present in Figure 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 4: Examples of signatures of Cobol code generators.

In Figure 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 paper. 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 analyzes 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 [89] 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 [89, 42]. 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 [89, 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 [66, 67]. 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 [26, 84], 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 paper. 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.

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.

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 publically 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. It is well observed that when industrial projects are in place, such as ours, 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.

In the context of our study all of these requirements must have been fulfilled. We dealt with more than 18 millions of lines of code. To tackle such an amount of source code we opted for 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 (R and Matlab). Here, the feature of pretty printing was vital.

Given the stated requirements we opted for *Perl* as our main implementation vehicle [108, 109]. In order to instantiate the codebase for the studied portfolio we used the MySQL database engine [35]. We found this open-source product sufficient to accommodate our needs. Furthermore, it provided us with a powerful database engine.

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 5 we visualize the analysis process.

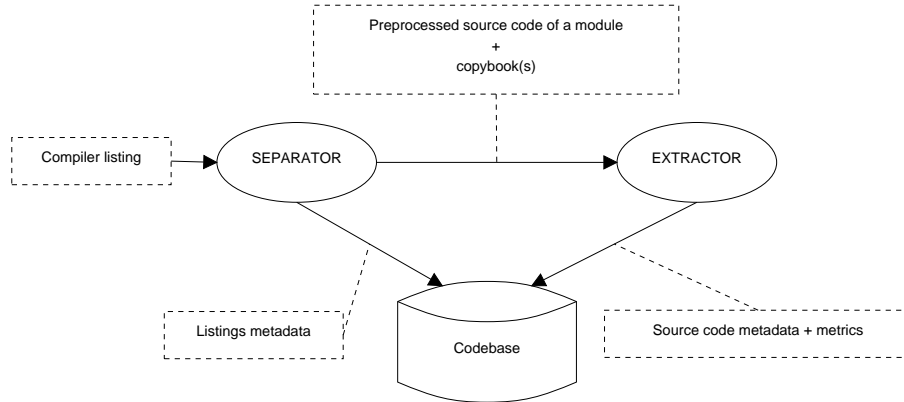


Figure 5: Compiler listing analysis process.

Initially each compiler listing is processed by the *Separator*, indicated in Figure 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 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 literature [108, 109].

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 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 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 6 we present a *Perl* code fragment from the separator’s script containing definitions of the regular expressions.

Figure 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 there are no unidentified compiler listings left. In case when such 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

```

$1st_type1 = '1PP 5740-CB1 RELEASE 2\4\s+(IBM OS\VS COBOL\s+JULY 1\, 1982)\s+\d{1,2}\.\d{1,2}\.\d{1,2}\s+(\d{1,2})\s+(\d{4})$';
$1st_type2 = '1PP 5648-A25 (IBM COBOL for OS\390 \& VM 2\.\d\.\d\.\d)\s+Date (\d{2})\/(\d{2})\/(\d{4}) Time [\d\.]{8} Page[\s\d]+$';
$1st_type3 = '1PP 5655-G53 (IBM Enterprise COBOL for z/OS 3\.\d\.\d\.\d)\s+Date (\d{2})\/(\d{2})\/(\d{4}) Time [\d\.]{8} Page[\s\d]+$';
$1st_type4 = '1PP 5688-197 (IBM COBOL FOR MVS & VM \d\.\d\.\d\.\d)\s+Date (\d{1,2})\/(\d{1,2})\/(\d{4}) TIME [\d\.]{8}\s+PAGE[\s\d]+$';

```

Figure 6: Regular expressions used to recognize compiler listing types.

```

1 000318 031600 03 MSGNUM PIC X(6). 03160001 BLW=00008+8E0,0000140 6C
2 000319 031700 03 ERRORMSG PIC X(80). 03170001 BLW=00008+8E6,0000146 80C
3 000320 031800 03180001
4 1PP 5655-G53 IBM Enterprise COBOL for z/OS 3.3.1 NC011 Date 09/10/2005 Time 14:05:54 Page 10
5 LineID PL SL ----+*A-1-B-----2-----+-----3-----+-----4-----+-----5-----+-----6-----+-----7-|-+-----8 Map and Cross Reference
6 000321 031900/** MSG-1 *** 03190001
7 000322 032000 COPY MSG1. 03200002
8 000323C *****
9 000324C *** NAME :MESSAGE 1 00000201
10 000326C *****
11 000327C 01 MSG-1. *****
12 000328C 03 IMS-HEADER. 0000601 BLW=00008+938 OCL32745
13 000329C 05 AA-VALUE USAGE COMP-8 PIC S9(3). 0000701 BLW=00008+938,0000000 OCL13
14 000330C 05 BB-VALUE VALUE +0 USAGE COMP-8 0000801 BLW=00008+938,0000000 2C
15 000331C PIC S9(4). 0000901 BLW=00008+93A,0000002 2C
16 000332C 05 IMS-CODE PIC X(9). 0000101 BLW=00008+93C,0000004 8C
17 000333C 05 FILLER PIC X(1). 00001201 BLW=00008+944,0000000 1C
18 000334C 05 MESSAGE-TEXT PIC X(12400). 00003001 BLW=00008+965,000002D 32700C
19 000335 032100 03210001
20 1PP 5655-G53 IBM Enterprise COBOL for z/OS 3.3.1 NC011 Date 09/10/2005 Time 14:05:54 Page 11
21 LineID PL SL ----+*A-1-B-----2-----+-----3-----+-----4-----+-----5-----+-----6-----+-----7-|-+-----8 Map and Cross Reference
22 000336 032200/** NEXT MESSAGE *** 03220001

```

Figure 7: Fragment of a compiler listing taken from the studied portfolio.

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

Each regular expression in Figure 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 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 `$cobol_line2`, the part of the line embraced by the brackets () is appended to the file where the preprocessed code of a Cobol modules is held. From the listing presented in Figure 7 the lines 1–3,6–7,19, and 22 would be appended to the Cobol's module. If the line matches the regular expression denoted by `$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 [64] that precedes the copybook content. In case of the listing presented in Figure 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 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 analyzes we stored them in the DD-MM-YYYY format.

3.4 Metrics computation

In the process of collecting metrics, listed in Section 2.3, 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 [42, 64, 89, 71, 23, 67]. 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:

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 [89, 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 through 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 [67]. 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 [37]. 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 $MC(m)$.

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 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 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 9.

- Historical changes are typically characterized by means of three data items: date of change, version number, and name or identifier of a pro-

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 analyzes. 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 paper.

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 [65, 52, 12, 6, 21]. 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 .

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 [19, 76] since it has been shown to perform well in combination with LSI [20, 110]. 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 [19, 76, 99]. 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 [99] the authors discuss several approaches to choosing k . As it turned out for us determining the value of k on the basis of formula 1, taken from [60], sufficed to facilitate our needs.

$$k = (m \cdot n)^{0.2}. \tag{1}$$

In formula 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.

$$\text{cos}(i, j) = \frac{e_i^T A_k^T A_k e_j}{\|A_k e_i\|_2 \|A_k e_j\|_2}. \quad (2)$$

Formula 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 [99]. 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 1. By means of formula 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 [13]. 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*) [99]. Apart from the linkage criteria it is also important to determine the tree cut-off threshold used to identify the actual clusters. In [60] 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 [40].

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 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: 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 cluster 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 cluster 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 10: Regular expressions corresponding to the recovered CASE tools signatures.

Figure 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.

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 analyzes 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.

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.

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 analyzes are an optional element of the Cobol sources. So, in case they are absent the related metadata is also missing.

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

Table 3: Cobol modules coverage by the metadata.

In Table 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 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.

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.

In Tables 4 and 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

Metric	<i>LOC</i>	<i>SLOC</i>	<i>CLOC</i>	<i>BLOC</i>	<i>MC</i>
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 4: The five-number summaries for the metrics: *LOC*, *SLOC*, *CLOC*, *BLOC*, and *MC*.

Metric	<i>OBS_x</i>															
	<i>x</i>	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	0
1st Q	0	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	0
3rd Q	0	0	0	0	1	0	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 5: The five-number summaries for the *OBS_x* metrics.

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 [64]. From practice we know that at most one occurrence of any of these keywords happens. From Table 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.

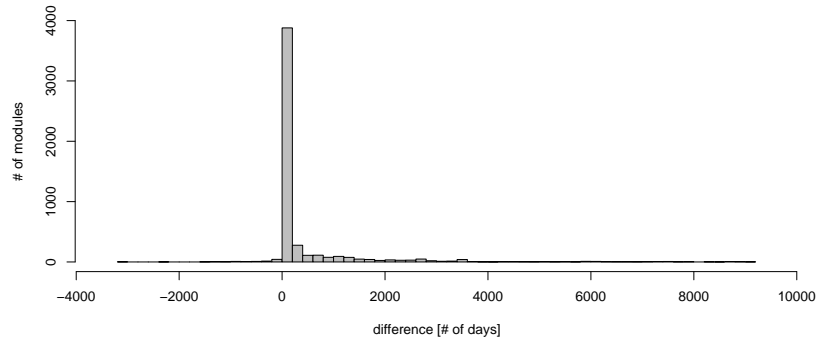


Figure 11: Distribution of the distance (measured in days) between the modules' last compilation and last modification dates.

Figure 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 analyzes.

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 [11]. 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 [24]. CANAM Report Composer is a code generator specific to writing programs that generate reports [87]. It supports data retrieval and its presentation. KEY:CONSTRUCT is part of the KEY:Enterprise suite [90]. Since all the detected CASE tools are valid Cobol code generators we assumed the values of the GEN metadata as valid.

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.

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) [3, 28, 18, 53]. 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 [54, 50]. For executives it is important to know how reliable these metrics are. In [29, 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 [38]. 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 [2, 16]. 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 [46]. It is claimed that the method yields results with the accuracy of approximately $\pm 20\%$ [48, 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 [14]. 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.

$$f(S) = \frac{1}{107} \sum_{m \in S} SLOC'(m). \quad (3)$$

In formula 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 [46]. 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.

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 analyzes 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 analyzes.

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 [44, p. 202].

$$d(f) = f^{0.39}. \quad (4)$$

In Formula 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 [44, p. 202-203]. These are overall benchmarks, not specific for the MIS industry.

$$n_d(f) = \frac{f}{150}. \quad (5)$$

$$n_m(f) = \frac{f}{750}. \quad (6)$$

Formula 5 is used to estimate the size of the software development team. Formula 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 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 4. The total development cost can then be calculated as follows.

$$tcd(f, r, w) = \frac{f}{150} \cdot r \cdot w \cdot \frac{f^{0.39}}{12}. \quad (7)$$

In formula 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 6. We express the yearly cost of operation, denoted by yco , as follows.

$$yco(f, r, w) = \frac{f}{750} \cdot r \cdot w. \quad (8)$$

As we see in formula 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 [106]. Based on the public benchmark data from [48, p. 192] the following formulas for chance of IT-project failure were estimated in [106].

$$cf_i(f) = 0.4805538 \cdot (1 - \exp(-0.007488905 \cdot f^{0.587375})). \quad (9)$$

$$cf_o(f) = 0.3300779 \cdot (1 - \exp(-0.003296665 \cdot f^{0.6784296})). \quad (10)$$

In formula 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 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 9 and 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 11 for a portfolio A is expressed as follows.

$$P_{rc}(A, r, w) = \sum_{S \in A} tcd(f(S), r, w). \quad (11)$$

In order to estimate the risk of failure of rebuilding a group of systems we use formula 12 proposed in [106, p. 49].

$$P_{rf_y}(A) = \frac{1}{|A|} \cdot \sum_{S \in A} cf_y(f(S)). \quad (12)$$

The yearly operational cost of keeping a collection of systems up and running are estimated on the basis of formula 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.

$$P_{yco}(A, r, w) = \sum_{S \in A} yco(f(S), r, w). \quad (13)$$

In formulas 11, 12 and 13 the function f approximates the size of the given information system S , according to formula 3. The function cf_y represents either a formula for estimating the chance of failure in case of in-house developed projects (formula 9) or the one used for outsourced developments (formula 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 14.

$$P_d(A) = \max_{S \in A} \{d(f(S))\}. \quad (14)$$

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 15.

$$P_{n_d}(A) = \sum_{S \in A} n_d(f(S)). \quad (15)$$

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

$$P_{n_m}(A) = \sum_{S \in A} n_m(f(S)). \quad (16)$$

Again, in formulas 14, 15 and 16 A denotes the set of systems for which the estimate is done. The identifiers d , n_d , and n_m denote formulas 4, 5 and 6, respectively.

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 [69, 85]. 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.

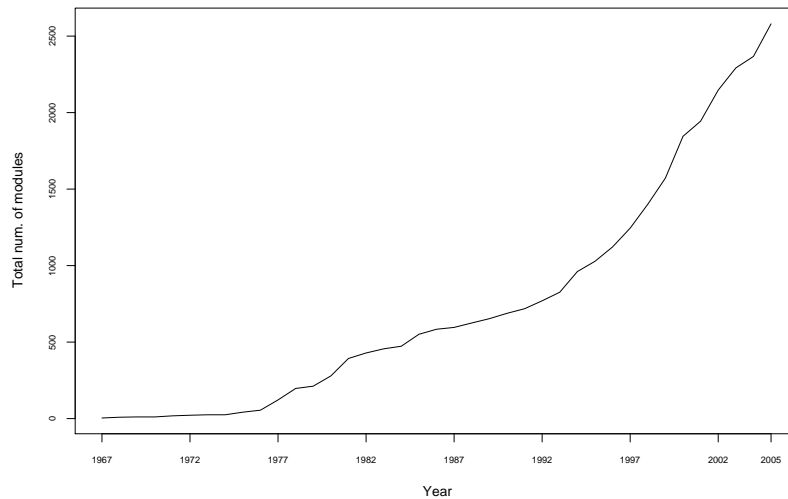


Figure 12: Reconstruction of the process of amassing source code in the portfolio on the basis of modules with the defined *DC* dates.

Figure 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 [31]. Table 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.

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

Let us explain Table 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 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 17.

$$cagr = \left(\prod_{i=1}^n r_i \right)^{\frac{1}{n}} - 1. \quad (17)$$

In formula 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 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 analyzes 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 [69]. In [85] 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 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 6 and computed the *cagr* for the last five years ($i = 6 \dots 10$). 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.

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.

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

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

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 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 13 presents more detailed characteristics.

The plot in Figure 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 [111] 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 paper strongly resembles the one revealed in the other research.

In Table 7 we provide the distribution of the systems over 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 [45]. A large 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

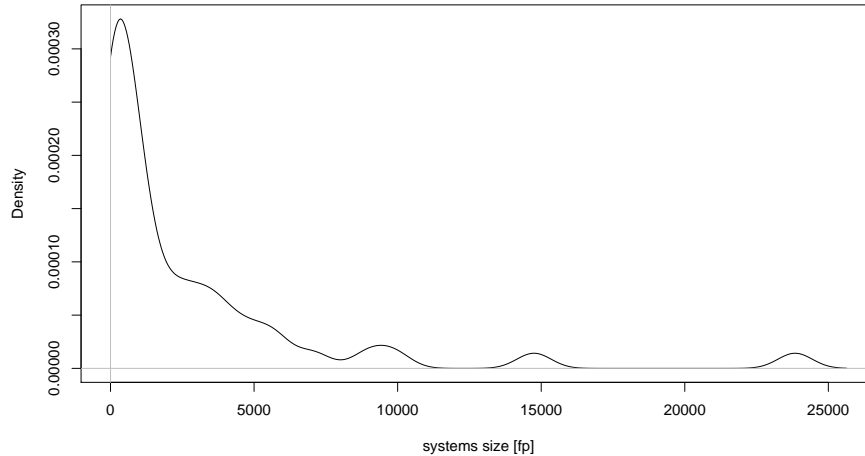


Figure 13: Distribution of the estimated function point counts.

Class property	Size class (FPs)				All classes
	[0..100]	(100..1,000]	(1,000..10,000]	(10,000..100,000]	
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 7: Characteristics of the size classes of the portfolio systems.

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 [45, p. 51]. These reference figures suggest that we deal with a portfolio comprising a significant amount of software.

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 analyzes 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 8 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 8: Levels of importance of the portfolio systems in business operations.

Importance level	Size class (FPs)				All classes
	[0..100]	(100..1,000]	(1,000..10,000]	(10,000..100,000]	
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 9: Distribution of the portfolio systems over size and their importance levels.

Table 9 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 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_{nm}	119.2	18.2	28.1	1.8	167.3
P_{yco}	23.8	3.6	5.6	0.4	33.4

Table 10: The portfolio’s benchmarked spendings and staff assignment.

In Table 10 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 10 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 11 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_{r_b}	356.2	34.9	64.2	2.3	457.6
P_{r_f} (in-house)	24.11%	14.09%	15.37%	7.07%	18.08%
P_{r_f} (outsourced)	15.69%	8.41%	9.48%	3.86%	11.39%

Table 11: Market valuation of the portfolio.

In Table 11 we present selected managerial indicators computed for the same sub-portfolios as those listed in Table 10. 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_{r_b} 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_{r_f}). 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 analyzes. 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 analyzes we used the data available from the codebase.

i	Critical		Sensitive		Non-sensitive	
	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
<i>cagr</i>		12.24%		14.22%		4.82%
All mods	5634		950		1527	
<i>DC</i> mods	25.51%		14.32%		56.12%	

Table 12: Estimates of the compound annual growth rates for the source code implementing the systems from the three sub-portfolios: critical, sensitive, and non-sensitive.

Table 12 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 12 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 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 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_{nm} , 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 14 presents four plots each of which shows the calculated estimates

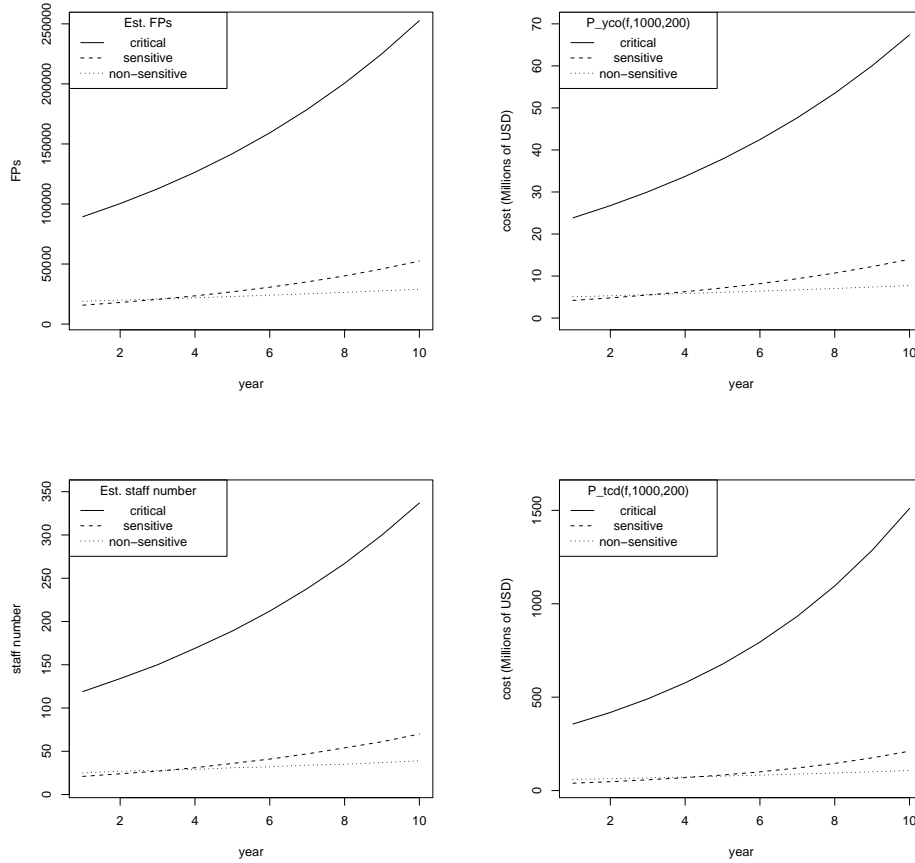


Figure 14: Growth scenarios of three different sub-portfolios.

for the expected number of function points, P_{yco} , P_{nm} , 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_{nm} , and P_{rb} formulas which all dependent on the function points and are all increasing. 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 analyzes 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.

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 [74, 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.

Table 13 presents the distribution of the portfolio Cobol modules over the detected code generators. In the first column we list the names of the code

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

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 situation obviously raises potential problems with maintainability.

Migration impact One important information that we learned from Table 13 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 [9, 86, 88]. 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 [95]. 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 analyzes. 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.

Code generator	Critical		Sensitive		Non-sensitive		Not-rated	
	#	%	#	%	#	%	#	%
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 14: Distribution of the portfolio Cobol modules over the code generators and the systems’ importance levels.

In Table 14 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 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 14 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 13). 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 15: Distribution of the portfolio modules over staff needed for maintenance.

Table 15 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 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 analyzes 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 [103, 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 [63]. 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 [36, 49]. 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 16 provides the names of the compilers along with the distribution of their usage over the portfolio modules.

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 16: Last compilation of the portfolio modules: compilers coverage.

In the first column of Table 16 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 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 article [36]. 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.

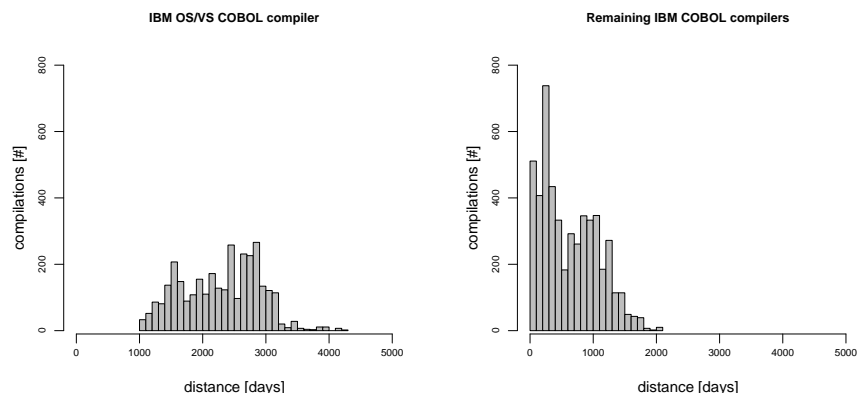


Figure 15: Compilations in the portfolio: the IBM OS/VS COBOL compiler and the remaining compilers.

In Figure 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.

Compiler name	Critical		Sensitive		Non-sensitive		Not-rated	
	#	%	#	%	#	%	#	%
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 17: 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 17. 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.

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 analyzes presented in this paper 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 analyzes 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 analyzes. We treated one construct, the **ALTER** statement, separately for its reputation for complicating program comprehension.

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

Table 18: Code characteristics of the hand-written modules.

Table 18 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 ALTERs. 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 ALTERs 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 [67]. In [33] we find that Hewlett Packard requires modules with a cyclomatic complexity higher than 16 to be re-designed. In [41] 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 process would involve identification of the qual-

ity 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 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

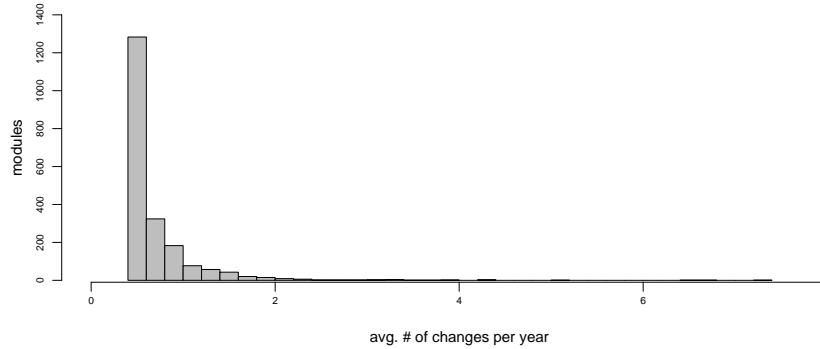


Figure 16: Distributions of the average number of code related activities over the top 25% of most volatile Cobol modules.

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 analyzes 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 19 we provide detailed characteristics of the top 25% of systems.

The first column in Table 19 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

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

and compared them with the remaining source modules.

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

Table 20: Characteristics of the hand-written modules from two groups: *hot spots* and other modules.

Table 20 provides characteristics of the hand-written modules from the two groups: *hot spots* and other modules. The content for Table 20 was generated in a similar manner as the one for Table 18. 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.

7 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 paper 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 [61]. 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 paper 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 ad-

just 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.

References

- [1] Advantage - Mainframe Pricing. Website. Available via http://www.ogs.state.ny.us/purchase/prices/7600021268PL_MainframeProducts.pdf.
- [2] 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.
- [3] A.J. Albrecht. Measuring application development productivity. In *Joint SHARE/GUIDE/IBM Application Development Symposium*, pages 83–92, 1979.
- [4] Nicolas Anquetil and Timothy C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance*, 11(3):201–221, 1999.
- [5] 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.
- [6] 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.
- [7] Bert Kersten and Chris Verhoef. IT Portfolio Management: A banker’s perspective on IT. *Cutter IT Journal*, 16(4), 2003.
- [8] Jesús Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. Legacy information systems: Issues and directions. *IEEE Software*, 16:103–111, 1999.
- [9] Bluephoenix. CA Gen Modernization – Coolgen Migration. Available via <http://www.bphx.com/en/Solutions/ApplicationModernization/Pages/CoolGen.aspx>.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] Carol Dekkers and Ian Gunter. Using ‘backfiring’ to accurately size software: more wishful thinking than science? *IT Metrics Strategies*, November 2000.
- [15] Tom DeMarco. *Controlling software projects*. Yourdon Computing Series, Upper Saddle River, New Jersey, USA, 1982.
- [16] Sheila Dennis and David Herron. FP lite - An alternative approach to sizing. Available from <http://www.davidconsultinggroup.com>.

- [17] Arie Deursen and T. Kuipers. Rapid system understanding: two COBOL case studies. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1998.
- [18] J. B. Dreger. *Function point analysis*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
- [19] S.T. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments and Computers*, 23:229–236, 1991.
- [20] Susan T. Dumais. Enhancing Performance in Latent Semantic Indexing (LSI) Retrieval, 1992.
- [21] 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.
- [22] Soumitra Dutta. Recognizing the True Value of Software Assets - Industry report. *Microfocus*, November 2007. http://www.microfocus.com/000/RecognisingTheTrueValueofSoftwareAssets_tcm21-16042.pdf.
- [23] Mike Ebbers, Wayne O'Brian, and Bill Oden. *Introduction to the New Mainframe: z/OS Basics*. IBM's International Technical Support Organization, July 2006.
- [24] eCube Systems LLC. Cool:Gen History. Available via <http://www.ecubesystems.com/coolgen.htm>.
- [25] D. Faust and C. Verhoef. Software product line migration and deployment. *Software Practice and Experience, John Wiley & Sons, Ltd*, 33:933–955, 2003.
- [26] Norman E. Fenton and Martin Neil. Software metrics: success, failures and new directions. *J. Syst. Softw.*, 47(2-3):149–157, 1999.
- [27] 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>.
- [28] David Garmus and David Herron. *Function Point Analysis: Measurement Practices for Successful Software Projects*. Addison-Wesley, 2001.
- [29] P.R. Garvey. *Probability Methods for Cost Uncertainty Analysis – A Systems Engineering Perspective*. Marcel Dekker Inc., New York, 2000.
- [30] 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>.
- [31] 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.
- [32] GNU. GNU Grep: Print lines matching a pattern. Technical report, 2010. <http://www.gnu.org/software/grep/manual/>.
- [33] Robert B. Grady and Deborah L. Caswell. *Software metrics: establishing a company-wide program*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.
- [34] 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.
- [35] Stefan Hinz, Paul DuBois, Jonathan Stephens, Martin 'MC' Brown, and Anthony Bedford. MySQL Reference Manual, 2009. <http://dev.mysql.com/doc/>.
- [36] 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>.

- [37] 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>.
- [38] IFPUG. Function Point Counting Practices Manual, Release 4.1. Technical report, International Function Point Users Group (IFPUG), Mequon, Wisconsin, USA, January 1999.
- [39] 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>.
- [40] The MathWorks Inc. Matlab. Technical report, 2010. Available via <http://www.mathworks.com/access/helpdesk/help/techdoc/>.
- [41] Software Engineering Institute. *C4 Software Technology Reference Guide—A Prototype*. Carnegie Mellon University, 1997. Handbook CMU/SEI-97-HB-001.
- [42] ISO/IEC. Information technology Programming languages COBOL. Technical report, ISO/IEC 1989:2002(E), 2002.
- [43] Jaap Bloem and Menno Van Doorn and Piyush Mittal. *Making IT Governance Work in a Sarbanes-Oxley World*. John Wiley and Sons, Inc., 2006.
- [44] Capers Jones. Software metrics. *Computer*, pages 98–101, September 1994.
- [45] Capers Jones. *Patterns of Software Systems Failure and Success*. International Thomson Computer Press, Boston, MA, 1996.
- [46] Capers Jones. Programming Languages Table, Release 8.2, March 1996. <http://www.spr.com>.
- [47] Capers Jones. *The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences*. Addison-Wesley, ACM Press, 1998.
- [48] Capers Jones. *Software Assessments, Benchmarks, and Best Practices*. Addison Wesley Information Technology Series, 2000.
- [49] Larry Kahm. Building an Early Warning System to Enable COBOL Compiler Migration. Available via <http://www.heliotropicsystems.com/pubs/HSTSa112007.pdf>.
- [50] P. Kampstra and C. Verhoef. Reliability of function point counts. Available via <http://www.cs.vu.nl/~x/rofpc/rofpc.pdf>.
- [51] R.S. Kaplan and D.P. Norton. *The Balanced Scorecard Translating Strategy into Action*. Harvard Business School Press, 1996.
- [52] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, and Katsuro Inoue. MUD-ABlue: an automatic categorization system for open source repositories. *J. Syst. Softw.*, 79(7):939–953, 2006.
- [53] Chris F. Kemerer. Reliability of function points measurement: a field experiment. *Commun. ACM*, 36(2):85–97, 1993.
- [54] T.A. Kirkpatrick. Research: CIOs speak on ROI. *CIO Insight*, 1(11), 2000. Available via <http://www.cioinsight.com>.
- [55] 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.
- [56] Paul Klint and Chris Verhoef. Enabling the creation of knowledge about software assets. *Data Knowl. Eng.*, 41(2-3):141–158, 2002.
- [57] A. S. Klusener and C. Verhoef. 9210: The Zip Code of Another IT-Soap. *Software Quality Control*, 12(4):297–309, 2004.
- [58] Steven Klusener, Ralf Lammel, and Chris Verhoef. Architectural Modifications to Deployed Software. *Science of Computer Programming*, 54:143–211, 2005.

- [59] R. L. Krikhaar. *Software Architecture Reconstruction*. PhD thesis, University of Amsterdam, The Netherlands, 1999.
- [60] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.
- [61] Lukasz Kwiatkowski and Chris Verhoef. Reducing operational costs through MIPS management. Available via <http://www.cs.vu.nl/~x/mips/mips.pdf>.
- [62] Lukasz 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.
- [63] Ralf Lämmel and Chris Verhoef. Cracking the 500-language problem. *IEEE Software*, 18(6):78–88, 2001.
- [64] Ralf Lämmel and Chris Verhoef. VS COBOL II grammar Version 1.0.4. Technical report, Vrije Universiteit Amsterdam, 2002.
- [65] 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.
- [66] Thomas J. McCabe and Charles W. Butler. Design complexity measurement and testing. *Commun. ACM*, 32(12):1415–1425, 1989.
- [67] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-12(3):308–320, 1976.
- [68] META Group. The Business of IT Portfolio Management: Balancing Risk, Innovation, and ROI. Technical report, META Group, Stamford, CT, USA, January 2002.
- [69] 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.
- [70] 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.
- [71] Hewlett Packard. HP 3000 Manuals: HP COBOL II/XL Reference Manual.
- [72] 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.
- [73] 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.
- [74] 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.
- [75] 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.
- [76] Gerard Salton and Chris Buckley. Term Weighting Approaches in Automatic Text Retrieval. Technical report, Ithaca, NY, USA, 1987.
- [77] Don Schriker. Cobol for the next millennium. *IEEE Software*, 17(2):48–52, 2000.
- [78] Alex Sellink, Harry Sneed, and Chris Verhoef. Restructuring of COBOL/CICS legacy systems. *Science of Computer Programming*, 45:193–243, 2002.
- [79] 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>.

- [80] 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.
- [81] 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.
- [82] 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.
- [83] 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.
- [84] Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Softw. Eng. J.*, 3(2):30–36, 1988.
- [85] 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.
- [86] SOFTMAINT. FAST for COOL:Gen: eliminate the license costs. Available via <http://www.softmaint.com/solutions/architecture-modernization/fast-for-coolgen/>.
- [87] Canam Software. Canam Composer: overview. Available via http://www.canamsoftware.com/product/report_comp/index.html.
- [88] TSG Software. CA-Telon. Available via <http://www.tsg.co.uk/html/ca-telon.html>.
- [89] SRDI. *COBOL 85 Language Reference*. SRDI, 2000.
- [90] 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>.
- [91] 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.
- [92] 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.
- [93] 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.
- [94] Paul Strassmann. Will big spending on computers guarantee profitability? Website, February 1997. Available via <http://www.strassmann.com/pubs/datamation0297/>.
- [95] Andrey A. Terekhov and Chris Verhoef. The realities of language conversions. *IEEE Software*, 17(6):111–124, November 2000.
- [96] 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.
- [97] 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.
- [98] 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>.

- [99] 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>.
- [100] 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.
- [101] N. Veerman. Automated mass maintenance of a software portfolio. *Science of Computer Programming*, 62:287–317, 2006.
- [102] N. Veerman and E. Verhoeven. Cobol minefield detection. *Software: Practice & Experience*, 36(14):1605–1642, 2006.
- [103] Niels Veerman. *Automated mass maintenance of software assets*. PhD thesis, Vrije Universiteit Amsterdam, The Netherlands, 2007.
- [104] 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>.
- [105] Chris Verhoef. Towards automated modification of legacy assets. *Annals of Software Engineering*, 9(1–4):315–336, May 2000.
- [106] Chris Verhoef. Quantitative IT Portfolio Management. *Science of Computer Programming*, 45(1), 2002.
- [107] 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.
- [108] L. Wall, T. Christiansen, and R.L. Schwartz. *Programming Perl, 2nd Edition*. O’Reilly & Associates, Inc., 1996.
- [109] L. Wall and R.L. Schwartz. *Programming Perl*. O’Reilly & Associates, Inc., 1991.
- [110] 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.
- [111] Cheung Y., Willis R., and Milne B. Software benchmarks using function point analysis. *Benchmarking: An International Journal*, 6(3):269–276, 1999.
- [112] Nicholas Zvegintzov. Frequently Begged Questions and How To Answer Them. *IEEE Software*, pages 93–96, March/April 1998.