Legacy Software Evaluation Model
for Outsourced Maintainer

Cristiane S. Ramos
Politec Ltda
cristianer@bsb.politec.com.br

Kátia M. Oliveira, Nicolas Anquetil
Catholic University of Brasilia
{kathia,anquetil}@ucb.br

Abstract

Outsourcing has become common practice in the software industry. Organizations routinely subcontract the maintenance of their software assets to specialized companies. A great challenge for these companies, is to rapidly evaluate the quality of the systems they will have to maintain so as to accurately estimate the amount of work they will require.

To answer these concerns, we developed a framework of metrics to evaluate the complexity of a legacy software system and help an outsourcing maintainer define its contracts. This framework was defined using a well known approach in Software Quality, called “Goal-Question-Metric”.

In this article we will present the Goal-Question-Metric approach, its results, and the initial experimentation of the metrics on five real life systems in Cobol.

1. Introduction

As software maintenance outsourcing becomes more common, outsourced maintainers are faced with new challenges. One of them is the ability to rapidly evaluate a system to get an idea of the cost to maintain it. Depending on the type of maintenance contract offered, this cost may be linked to a number of variables: number of maintenance requests, urgency of these requests, their intrinsic difficulty, or the ability of the software system to endure modifications. Rapidly evaluating such variables without previous knowledge of a system is a difficult task that all outsourced maintainer must perform. Polo [16] highlights that this problem received yet little attention.

In this paper, we are more concerned with evaluating the ability of a software system to be understood and modified cost effectively, a characteristic called maintainability in software quality. We could notice, however, that the metrics defined to evaluate the maintainability of a software system (e.g. in the international standard on software quality, ISO/IEC 9126 [7, 8, 9, 10]) are intended to be collected during maintenance as a mean to monitor the evolution of the system. Such a model is not adapted to the necessity of an outsourced maintainer trying to evaluate a system before it starts maintaining it, for example to price its services.

Using a methodology called GQM (Goal-Question-Metric) to establish a software quality plan, we designed a model to assess the complexity of a software system from the point of view of the maintainer. Although, we are well aware that this model may not fully represent the maintainability of the systems (e.g. as defined in software quality standards), we hope it can give good insights on this quality at a lower cost and mainly without previous knowledge of the system. In this paper, we present our model, detailing how we came to define it. We also present empirical results on five industrial projects in Cobol.

The organization of the paper is the following: First we review the related work in section 2. In section 3, we give some basic notions of software measurement, and present the GQM methodology used to define a software measurement plan. Our model is detailed in section 4 and we present and discuss a preliminary validation experiment in section 5. Finally we conclude and discuss future work in section 6.

2. Related Work

There is a plethora of published metrics for software. However, they usually are not directly related to our goal. First many of them are defined for object-oriented systems whereas our outsourced maintainer mostly deals with programs in Cobol. Second, even if they are for procedural languages, they may not help assessing the ease with which a software will be maintained.

Software quality is a well developed field, with numerous publications and international standards such as the ISO-9126, ISO-12219, etc. The main standard in this field is the ISO/IEC 9126 [7] which defines various quality characteristics for software products. There are three technical reports associated to the ISO/IEC 9126 [8, 9, 10] proposing...
metrics for each one of these characteristics. Together these three technical reports define close to 200 metrics.

One of the characteristics is Maintainability: “The capability of the software product to be modified” [7, p.10]. The ISO/IEC 9126 standard and the accompanying technical reports are organized as follow:

ISO/IEC 9126: Presenting the model, the quality characteristics and sub-characteristics.

ISO/IEC 9126-2: Dealing with external metrics, measuring for example “such attributes as the behavior of the maintainer, user, or system [... ] when the software is maintained” ([8]: Maintainability).

ISO/IEC 9126-3: Dealing with internal metrics, used for example “for predicting the level of effort required for modifying the software” ([9]: Maintainability).

ISO/IEC 9126-4: Dealing with quality in use. It does not include Maintainability as a quality characteristic.

From this model, the first and last parts do not apply to our case. ISO/IEC 9126-2 does deal with maintainability, but would imply monitoring the maintainers of a system. Apart from the difficulty of having to wait for some time (months) to get a clear idea of how the system is behaving under maintenance. Getting past information on the system is not always possible, either because it was not recorded, or because the data from another maintenance team would not necessarily apply to the outsourced maintainer.

ISO/IEC 9126-3 would be the most interesting for us, however, looking at the metrics it proposes we perceived two problems. First, they suppose a level of documentation rarely attained in real world legacy software. An outsourced maintainer cannot rely on such an ideal system. Second, they are mostly designed to measure maintainability during the system’s life cycle, and assume a model where one follows the evolution of the software as maintenance occurs. Example of these metrics are:

- “Number of changes in functions/modules having change comment confirmed in review” DIVIDED-BY “Total number of function/modules changed from original code”
- “Number of affected variable data by modification, confirmed in review” DIVIDED-BY “Total number of variables”.
- “Number of implemented built-in test function as specified, confirmed in review” DIVIDED-BY “Number of built-in test functions required”.
- “Number of detected adverse impacts after modifications” DIVIDED-BY “Number of modifications made”.

In summary, although the ISO/IEC TR 9126 proposes some maintainability metrics, it does so from the point of view of an organization maintaining a software on an extended time frame and with an extensive documentation on both the development and maintenance activities. This model does not fit the case of an outsourced maintainer which must evaluate a legacy software with no previous knowledge of it.

The Datrix team (see for example [11]) considered a problem similar to ours. Their problem was to assess the quality of a software system before investing in it, and verify that it had the potential to gracefully evolve to be adapted to future (yet unknown) requirements. Our position is a little different since an outsourced maintainer does not require the software it will receive be easy to maintain, but merely need to get an idea whether it will be difficult (costly) to maintain or not. The Datrix team based its model on common sense observation such as: number of indirections (gotos) in the code, number of line of code, number of cloned functions, etc. We are basing our model on a better grounded quality approach, where we applied the GQM methodology to define the metrics we wish to apply.

Misra and Bhavsar [13] “investigate the usefulness of a suite of widely accepted design/code level metrics as indicators of difficulty”. They analyzed 20 metrics over 30 “varying widely in size and application domains”. The research uses Halstead’s Difficulty metric as a basis against which to evaluate the value of other metrics that may be computed early in the development of a software system (design and implementation phases).

Frappier, Matwin, and Mili [5] also studied metrics to predict maintainability of a system, but they are considering metrics to predict maintainability during development and as such consider documentation artifacts (e.g. preliminary design document, detailed design document) which may not all be available for legacy software. They also deal with quality criteria such as documents readability which we will not consider.

A very closely related work is that of Polo et al. [16]. In this paper, the authors are adopting the same point of view as ours, that of an outsourced maintainer trying to assess the quality of a software system, with whatever little information it has on it. However, their approach is different from our in two senses: First they use the metrics that are already available which are the size of the systems in LoC (Lines of Code) and in number of programs. We, on the other hand, want to come up with a set of metrics that will certainly include, but will not be limited, to these two. The second difference between our approaches is that Polo et al. are considering the fault proneness of a system. In our case and because of the difference in maintenance contracts, we are more interested in knowing how difficult it will be to perform a maintenance.
Pearse and Oman present a model for measuring software maintainability in [14]. After studying several large software systems at Hewlett-Packard, they came up with a four metrics polynomial Maintainability Index\(^1\). One may make some objections to this work. First, it seems somehow improbable that a single formula (however carefully crafted) may adequately measure such an abstract notion as maintainability. Second, the formula was fine tuned for software systems at Hewlett-Packard, and would probably need to be adapted to new environments. The difficulty for an outsourced maintainer is that each new system potentially comes from a very different source than the preceding ones, which would require to fine-tune the formula for the various potential clients. Third, the formula is a black box evaluation of a system, only indicating whether it seems easy to maintain or not. The formula does little to help identifying specific reasons for the poor or good maintainability of the system, a feature which would be needed to negotiate prices with a client. Finally, the formula only considers the source code of the systems, whereas one could try to take advantage of any existing documentation for example.

3. Software Measurement

Software measurement is formally defined as “the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules” [4]. In summary, it is the process to obtain a measure for an entity of the real world. Software measurement became common practice in many organizations since it is a necessary mechanism for feedback and continual evaluation. It allows project managers to define and implement more realistic plans, and to control and improve software process and products. Symons and McGarry \textit{et al.} [18] highlight that software measurement provides the information required to make key project decisions and to take appropriate actions in a given context.

To secure all these advantages, one should make sure that the measurement efforts are in accordance with the project, process, and product measured, otherwise, one risks making wrong decisions [15]. A measurement only makes sense when associated with some clearly defined goals and models [1]. The measurement model should describe the entities measured and their attributes and the relationships among the measures [15].

A commonly used methodology to establish a measurement model, is the Goal-Question-Metric (GQM), proposed by Basili [1] and that has been largely used (see for example the case studies described in [19]). The GQM methodology helps identifying the metrics required and the reasons why the data are being collected.

The main idea of GQM is that measurement should be goal-oriented. It is a top-down methodology [19] starting with the definition of an explicit measurement goal that is refined in several questions that break down the goal into its major components. Then, each question is refined into metrics that, when measured, will provide information to answer the questions. By answering the questions we will be able to analyze if the goal has been attained. As a result of this process we have a model in three levels [1] (see Figure 1): the conceptual level, where a goal is defined for an object (product, process or resource) from various points of view; an operational level, where questions are defined for the goal and characterize the object of measurement with respect to a selected quality issue from the selected point of view; and, the quantitative level, representing a set of data associated with each question in order to answer it in a quantitative way. The data can be of two kinds: objective, depending only on the object and not on the point of view (e.g. size of a program, number of errors, etc); or, subjective, if they depend on both the object and the viewpoint (e.g. level of user satisfaction, clearness of a specification, etc.)

A commonly used methodology to establish a measurement model, is the Goal-Question-Metric (GQM), proposed by Basili [1] and that has been largely used (see for example the case studies described in [19]). The GQM methodology helps identifying the metrics required and the reasons why the data are being collected.

The main idea of GQM is that measurement should be goal-oriented. It is a top-down methodology [19] starting with the definition of an explicit measurement goal that is refined in several questions that break down the goal into its major components. Then, each question is refined into metrics that, when measured, will provide information to answer the questions. By answering the questions we will be able to analyze if the goal has been attained. As a result of this process we have a model in three levels [1] (see Figure 1): the conceptual level, where a goal is defined for an object (product, process or resource) from various points of view; an operational level, where questions are defined for the goal and characterize the object of measurement with respect to a selected quality issue from the selected point of view; and, the quantitative level, representing a set of data associated with each question in order to answer it in a quantitative way. The data can be of two kinds: objective, depending only on the object and not on the point of view (e.g. size of a program, number of errors, etc); or, subjective, if they depend on both the object and the viewpoint (e.g. level of user satisfaction, clearness of a specification, etc.)

\begin{equation}
M I = 171 - 5.44 \times \ln(V) - 0.23 \times CCC - 16.2 \times \ln(LoC) + 50 \times \sin(\sqrt{2.46 \times LoCmt}) \text{ where } V \text{ is the average Halstead volume metric, }\CCC \text{ is the average extended cyclomatic complexity, } \LoC \text{ is the average number of line of code, and } \LoCmt \text{ is the average number of lines of comment.}
\end{equation}
[19] recommend that the project team itself do this interpretation and draw the conclusions, rather than the GQM team.

The result of the application of these phases will be presented in the next two sections.

4. Legacy Software Evaluation Model

We used the GQM methodology to define a model to rapidly assess the complexity to maintain a legacy software. For an outsourced maintainer, the cost of maintaining a software system may come from several sources:

- High number of maintenance requests per month. As described in [16], maintenance contracts may involve attending an unspecified (or partially specified) number of maintenance requests per month: the more maintenance requests, the higher the maintenance cost.
- Urgency of maintenance requests. Urgent maintenance requests may cost more to fulfill because they are not planned.
- Intrinsic difficulty of the maintenance requests. Some maintenance requests may be simple to implement, requiring only a simple change in a well known part of the system, others may require to restructure the entire system.
- Difficulty to understand and/or modify the system. Depending on the documentation quality, system modularity, programming languages used, data model, etc., the same maintenance request can be easy to implement or, on the contrary, quite difficult.

In this paper, we focus on the last of these issues. Maintenance is usually classified as corrective, adaptive, perfective, or preventive (e.g. see [15]). We will not use this classification as an outsourced maintainer may be requested to do any of these on a system.

As a result, our first measurement goal is:

To analyze the system documentation
with respect to completeness and consistency
from the viewpoints of analysts and programmers
in the context of the outsourced maintainer.

The second measurement goal defines what one should know about the source code of the system. Given that we know nothing about the future maintenance requests, the most important aspect to be considered is the complexity of the system. This complexity not only encompasses the source code itself, but also the user interface, the tables manipulated by the system, and the possible integration with other systems.

The second measurement goal is:

To analyze the system source code

\[ \text{The goals, their measurable questions and associated metrics will be described below according to the frame proposed in [19].} \]
for the purpose of assessing with respect to the complexity to understand and modify it from the viewpoints of analysts and programmers in the context of the outsourced maintainer.

Next, we derived measurable questions from each of these two goals. The questions should cover the measurement goals as completely as possible, they were derived following the guides proposed in [1]: How can one characterize the object measured and its attributes with respect to the goal? How does one evaluate the characteristics of the object with respect to the goal? . . .

The questions and metrics for each goal will be presented in the following sub-sections.

4.1. First Goal: Analyzing the Documentation

As discussed above we consider three aspects of the documentation: To what extent is the documentation complete? Does it really deal with the application domain? Are the different parts of the documentation consistent with themselves? Each one of these aspects resulted in a measurable question. Ideally these questions would be measured automatically. However, this is very difficult for free form text in natural language. Therefore, we will use manual metrics but will try to select some that are easy to evaluate. We will come back to this point in the case study section 5.1.

Question 1: To what extent is the system documented?

To answer this question we will simply count how many elements of the requirements specification, data models and other diagrams are documented. The metrics are:

M1.1.a: Percentage of elements (external entities and data flows) from the context diagram which are documented

M1.1.b: Percentage of elements (external entities, data flows, processes, and data repositories) from the DFD level zero diagram which are documented

M1.1.c: Percentage of elements (tables, columns, and domains) from the physical data model which are documented

M1.1.d: Number of requirements which are documented

The interpretation of the result of these metrics will be based on a four-point scale: insufficient (from 0% to 25% of elements documented), fair (from 25% to 50% of elements documented), good (from 50% to 80% of elements documented) and excellent (above 80% of elements documented).

Next, we need to evaluate whether the documentation is consistent with the real world:

Question 2: Is the documentation consistent with the application domain and the application objectives?

To measure this question, we will verify how many elements of the documentation agree with the application domain. The metrics are:

M1.2.a: Percentage of documented elements (external entities and data flows) from the context diagram which are consistent with the application domain and application objectives

M1.2.b: Percentage of documented elements (external entities, data flows, processes, and data repositories) from the DFD level zero diagram which are consistent with the application domain and application objectives

M1.2.c: Percentage of documented elements (tables, columns, and domains) from the physical data model which are consistent with the application domain and application objectives

M1.2.d: Percentage of documented requirements from the requirement specification which are consistent with the application domain and application objectives

The interpretation of the result of these metrics will follow the same scale as proposed for the first question.

Finally the last question will allow measuring the consistency of the documentation within itself.

Question 3: Is the documentation consistent with itself?

For this, we need to consider all the possible combinations between the documentation artifacts (requirements specification, requirements description, data model, context diagram, DFD). The metrics are:

M1.3.a: Percentage of functional requirements from the requirement specification which have a detailed description

M1.3.b: Percentage of integration with other system in the requirement specification which are consistent with the context diagram (identified through external entities)

M1.3.c: Percentage of integration with other system in the requirement specification which are consistent with the production manual

M1.3.d: Percentage of elements (tables, columns, domains, and constraints) from the physical data model which are consistent with the production database

M1.3.e: Percentage of external entities from either the DFD level zero diagram or the context diagram which are in both the DFD level zero diagram and the context diagram
M1.3.f: Level of detail (1 for not satisfactory, 2 for satisfactory, and 3 for very satisfactory) of the requirements’ descriptions

M1.3.g: Level of consistency (1 for not satisfactory, 2 for satisfactory, and 3 for very satisfactory) of the production manual and the context diagram

M1.3.h: Level of consistency (1 for not satisfactory, 2 for satisfactory, and 3 for very satisfactory) of the production manual and the DFD level zero diagram

M1.3.i: Percentage of functional requirements which have a description of business rules

Interpretation of the metrics M1.3.a to M1.3.e is based on the same four-point scale.

All these metrics are used to analyze the documentation. With the data collected for each metric, one may answer the associated question. For example with the four metrics associated to question 1, one may get an idea of how well the system is documented (from insufficient to excellent). Answering all the questions, one has information to analyze the completeness and consistency of the documentation as stated in the first measurement goal.

4.2. Second Goal: Analyzing the Source Code

For the source code analysis, we are mainly interested in evaluating the complexity of the system. This includes: complexity of the source code, complexity of the user interface, complexity of the persistence layer, and complexity of interaction with other systems.

For this second measurement goal, we focused on automatic metrics (see the two restrictions in section 4). This is specially important for source code analysis since a typical legacy system would have millions of lines of code.

A first factor of complexity of the source code is its size:

Question 1: What is the size of the system?

To answer this question we consider different metrics that involves the size of the code (for the entire system and per program), number of programs, tables and languages used. The metrics are:

M2.1.a: Size of the system in LoC (physical lines in the file)

M2.1.b: Average number of LoC per program

M2.1.c: Number of tables used (from the system and from other systems)

M2.1.d: Average number of LoC per module

M2.1.e: Number of programs

M2.1.f: Average number of programs per module

M2.1.g: Number of screens

M2.1.h: Number of programs for each programming language used

There is no ordinal scale of size to evaluate the complexity of a system. The only interpretation is that the bigger a system is, the more difficult it is to understand and therefore maintain.

Sheer size is not enough to assess the complexity of a system, another important aspect is to know the level of interaction within the system, considering: number of calls among programs and number of access to tables.

Question 2: What is the level of internal transactions in the system?

To answer this question we derived the following metrics:

M2.2.a: Number of tables from the system accessed in input mode

M2.2.b: Number of tables from the system accessed in output mode

M2.2.c: Average fan-out per program

Again the interpretation cannot be based on an ordinal scale, but on the fact that the more of these interactions there is, the more complex the system is.

Finally, we need to assess the complexity of the source code in terms of the difficulty to understand it (e.g. complexity of the internal logic).

Question 3: What is the complexity of the code?

M2.3.a: Number of programming languages used

M2.3.b: Average cyclomatic number (defined in [12]) per program

M2.3.c: Average number of indirection per program

M2.3.d: Percentage of LoC with comments

M2.3.e: Average Halstead effort measure (defined in [6]) per program

Although many of these metrics have been used for a long time, there are few ordinal scale of interpretation. The only we could find was for the Cyclomatic Complexity (metric M2.3.b) with the following scale (see for example [5]): from 1 to 10, the program is simple; from 11 to 20, it is slightly complex; from 21 to 50, it is complex; and above 50 it is untestable.

Another aspect of the complexity is related to the user interface.
Question 4: What is the complexity of the user interface?

This will be measured mainly by the "size" of the user interface and some indication of the internal complexity of the interface. Note that we are considering here graphical interfaces as we targeted programs written in Cobol.

M2.4.a: Number of screens

M2.4.b: Average number of screens per program

M2.4.c: Average number of screen fields per screen

The last aspect of complexity that we consider is the quantity of interaction with other systems.

Question 5: What is the complexity of the interface with other systems?

M2.5.a: Number of tables used from other systems (note: equal to the cardinality of the union of the sets considered in M2.5.b and M2.5.c)

M2.5.b: Number of tables from other systems accessed in input mode

M2.5.c: Number of tables from other systems accessed in output mode

A few other metrics are still under study, as the Card-Glass complexity metric [2] which could be included in question 3; coupling between programs which could be included in questions 2 and 3; and number of library routines used in the programs which could be include in questions 3 and 5. Their inclusion will depend on the results of the current model.

5. Case Study

Although the research is still in an early stage, we started to collect data on some systems from a large software outsourcing company in Brazil, Politec Ltda. The company is one of the biggest in its branch in Brazil with close to six thousand employees distributed in different units located in several cities. The company’s outsourced maintenance division with which we worked has more than 85 systems in maintenance of varying size, programmed in different programming languages, and from different clients. We chose to work initially with Cobol source code, as it is one of the languages most used in legacy software systems. For this preliminary study, we selected five systems and analyzed the parts developed in Cobol.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Sys.1</th>
<th>Sys.2</th>
<th>Sys.3</th>
<th>Sys.4</th>
<th>Sys.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1.1.a</td>
<td>58%</td>
<td>50%</td>
<td>50%</td>
<td>-</td>
<td>64%</td>
</tr>
<tr>
<td>M1.1.b</td>
<td>-</td>
<td>48%</td>
<td>-</td>
<td>-</td>
<td>90%</td>
</tr>
<tr>
<td>M1.1.c</td>
<td>2%</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>M1.1.d</td>
<td>7</td>
<td>43</td>
<td>9</td>
<td>17</td>
<td>68</td>
</tr>
<tr>
<td>M1.2.a</td>
<td>58%</td>
<td>50%</td>
<td>50%</td>
<td>-</td>
<td>27%</td>
</tr>
<tr>
<td>M1.2.b</td>
<td>-</td>
<td>0%</td>
<td>-</td>
<td>-</td>
<td>65%</td>
</tr>
<tr>
<td>M1.2.c</td>
<td>1%</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>M1.2.d</td>
<td>86%</td>
<td>100%</td>
<td>89%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>M1.3.a</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>M1.3.b</td>
<td>75%</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>M1.3.c</td>
<td>100%</td>
<td>85%</td>
<td>0%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>M1.3.d</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>M1.3.e</td>
<td>-</td>
<td>29%</td>
<td>-</td>
<td>-</td>
<td>33%</td>
</tr>
<tr>
<td>M1.3.f</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>M1.3.g</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>M1.3.h</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>M1.3.i</td>
<td>86%</td>
<td>100%</td>
<td>33%</td>
<td>47%</td>
<td>100%</td>
</tr>
</tbody>
</table>

5.1. Documentation Analysis

In Table 1, we give the result of the evaluation of the documentation for the five systems. Given the nature of the metrics, this evaluation was manual. Table 2 indicates the time spent to analyze each system documentation.

The first comment is that many documents are missing, which is already an information about the quality of the system.

Two systems (2 and 5) have an extensive requirement specification with, respectively, 43 and 68 requirements documented. We did not analyze in detail all these requirements to evaluate metrics for the third question (internal consistence of the documentation) but randomly selected a sample of the requirements (for example, only 20 requirements for system 2) and extrapolated the results.

One of the authors had an insider knowledge of the systems analyzed. Using this knowledge (e.g. naming convention of tables) more information could have been deducted (e.g. tables belonging to the system or to other systems). We did not take advantage of this fact here, considering that the “normal” situation is that of software systems completely unknown to the outsourced maintainer.

Finally the analysis revealed the need to create a detailed description on how to perform it. Many questions are sub-
jective and need to be better described to get coherent results across various analysis from different persons.

One may highlight the following facts:

- There are many documentation items missing (see for example, results for question 1). This should not come as a surprise when dealing with legacy systems. This is also a strong evidence with regard to the (lack of) quality of the system documentation.

- Surprisingly, the database physical model was a difficult item to encounter. For some systems, there was a mention of some external documents where it could be find, but we have not been able to get access to these documents. For some systems, we found a logical data model and are considering reformulating our questions to consider this specific documentation artifact.

- As an other surprise, the requirement specification was the only item we could find for all five systems. This may be a specificity of the client organization as all systems came from the same client.

- Question 2 (Consistence of the documentation with the application domain) is generally low except for the requirements (M2.1.d). This means that one cannot safely rely on the documentation to describe the application domain.

- Results are mixed for question 3 (internal consistence of the documentation), system 1 has reasonable results, systems 2 and 3 seem to be the worsts, and systems 4 and 5 suffer from a lack of data (missing documentation).

Table 2. Time spent analyzing the system documentations, documentation size (number of pages), and number of pages analyzed per minute

<table>
<thead>
<tr>
<th></th>
<th>Sys.1</th>
<th>Sys.2</th>
<th>Sys.3</th>
<th>Sys.4</th>
<th>Sys.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>1h30</td>
<td>3h38</td>
<td>0h33</td>
<td>0h32</td>
<td>1h19</td>
</tr>
<tr>
<td>#page</td>
<td>44</td>
<td>100</td>
<td>36</td>
<td>53</td>
<td>N/A</td>
</tr>
<tr>
<td>min./p.</td>
<td>2.1</td>
<td>2.2</td>
<td>0.9</td>
<td>0.6</td>
<td>N/A</td>
</tr>
</tbody>
</table>

One may see in Table 2 that the documentation analysis is not too heavy a task. One may hope to spend about two minutes per page of documentation which in our worst case summed up to less than four hours of analysis. As already mentioned, for large documentation, for example with many requirements described, one may randomly select a significant sample.

The part that consumed most time was the third question (documentation internal consistence) that requires a cross analysis of different documents. Due to this cross analysis, the analysis time tends to grow rapidly with the number of pages (in our case, with the number of requirements described). Selecting a sample of the requirements seemed to give reasonable results in a manageable time frame.

Finally, the fact that all five systems were from the same client, facilitated the analysis as the documents followed the same pattern. After some time, it was easier to detect where a given information would be found. This may be an explanation for the difference in productivity for the systems.

5.2. Source Code Analysis

The metrics for the source code were computed automatically using Cobol and SQL parsers written in JavaCC. The results of the metrics are given in Table 3.

Table 3. Assessment of the source code of five systems (see text for a description of the metrics.)

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Sys.1</th>
<th>Sys.2</th>
<th>Sys.3</th>
<th>Sys.4</th>
<th>Sys.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question 1: System size?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M2.1.b</td>
<td>486.6</td>
<td>589.5</td>
<td>676.2</td>
<td>469.6</td>
<td>816.8</td>
</tr>
<tr>
<td>M2.1.c</td>
<td>21</td>
<td>49</td>
<td>8</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>M2.1.d</td>
<td>14,110</td>
<td>43,623</td>
<td>33,135</td>
<td>3,757</td>
<td>16,336</td>
</tr>
<tr>
<td>M2.1.e</td>
<td>29</td>
<td>74</td>
<td>49</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>M2.1.g</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Question 2: level of internal transaction?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M2.2.a</td>
<td>16</td>
<td>41</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>M2.2.b</td>
<td>18</td>
<td>34</td>
<td>8</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>M2.2.c</td>
<td>1.3</td>
<td>1.1</td>
<td>9.4</td>
<td>0.5</td>
<td>2.3</td>
</tr>
<tr>
<td>Question 3: Code complexity?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M2.3.a</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M2.3.b</td>
<td>11.7</td>
<td>26.4</td>
<td>16.4</td>
<td>14.0</td>
<td>50.6</td>
</tr>
<tr>
<td>M2.3.c</td>
<td>0.4</td>
<td>1.1</td>
<td>0.0</td>
<td>1.7</td>
<td>0.1</td>
</tr>
<tr>
<td>M2.3.d</td>
<td>29.8%</td>
<td>26.0%</td>
<td>28.8%</td>
<td>28.0%</td>
<td>20.3%</td>
</tr>
<tr>
<td>M2.3.e</td>
<td>32.7</td>
<td>1121.3</td>
<td>95.8</td>
<td>321.7</td>
<td>1407.4</td>
</tr>
<tr>
<td>Question 5: System interface complexity?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M2.5.a</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M2.5.b</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M2.5.c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Due to specificities of the examples, some metrics were not computed. For example, we only computed the metrics

4 These tools may be found on the web. JavaCC: https://javacc.dev.java.net/. Cobol grammar (had to be tailored to the Cobol dialect we were dealing with): http://mapage.noos.fr/~bpinon/ACobolParser.htm. SQL grammar, the ones found in the "JavaCC grammar repository": http://www.cobase.cs.ucla.edu/pub/javacc/.
for the Cobol part of the systems, and each system had only one Cobol module. The important points are:

- M2.1.a (total LoC of the system) was not computed.
- M2.1.d (average LoC per module) gives the size of the Cobol part as each system has only one Cobol module.
- M2.1.g (number of screens) has no meaning for these systems which either are batch programs or have screens described in other languages than Cobol. For the same reason, question 4 is irrelevant.
- M2.1.f (average number of program per module) and M2.1.h (number of programs per programming language) are not given as they are equal to M2.1.e (number of program) in this case.
- M2.2.a and M2.2.b (number of tables from the system read or wrote) do not consider flat files but only database tables.
- M2.2.c is based on the number of CALLs found in the Cobol programs.
- M2.3.b (average cyclomatic complexity) was computed from the number of test in the code as described in [17, p.446].

The following facts may be highlighted:

- The systems (actually modules) analyzed range from less than 4 KLoC to more than 40 KLoC. Overall, we automatically analyzed more than 110 KLoC of Cobol.
- The percentage of lines with comments (M2.3.d) is reasonably good (from 20% to almost 30%).

The interpretation of the data collected should be performed by the project team [19] (i.e. the maintainers) and we started to interview managers, analysts and programmers to understand how they informally evaluate the systems. The result of these interviews will then be used to define an interpretation of the result of each metric (e.g. up to 100 KLoC a system should be considered small).

We also plan to collect data from past maintenance of software systems (e.g. average number of hours spent on a modification request) to help in defining the interpretation of the metrics.

6. Conclusion

As outsourcing is becoming common practice in the software industry, maintenance providers (outsourced maintainers) are facing new problems, one of which is to rapidly evaluate the complexity of a software system to adequately negotiate their contracts with potential clients. To answer this need, we defined an evaluation model resulting from the application of the Goal-Question-Metric methodology.

The measurement plan considers two dimensions of a software system: its documentation and its source code. The model was applied on five industrial systems at our outsourced maintainer. We could first validate its applicability, results shows that the source code analysis is not a problem —since we only considered metrics that could be automated— and analysis of the documentation may be performed in reasonable time frame (less than four hours for the larger documentation).

The actual interpretation of the data collected is starting now, with the interview of members of the maintenance teams and collection of data from past maintenance.

7. Acknowledgment

The authors wish to thank the people at Politec’s outsourced maintenance division for their contribution to this research. Many thanks to Josivan Ribeiro and his team for their time, support, and invaluable knowledge.

This research was partly supported by the Hercules project, a cooperation between Politec and the Catholic University of Brasilia.

References


