# SOFTWARE METRICS:

# TOWARD BUILDING PROXY MODELS

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Izzat Mahmoud Alsmadi

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

April 2006

Fargo, North Dakota

**NORTH DAKOTA STATE UNIVERSITY**
**Graduate School**

Title
Software Metrics: Toward building proxy models.

By

Izzat M Alsmadi

The supervisory Committee certifies that this disquisition complies with North Dakota State University's regulations and meets accepted standards for the degree of

**Master of Software Engineering**

SUPERVISORY COMMITTEE

Chair

Approved by Department chair :

---------- ----------
Date Signature

## ABSTRACT

Alsmadi, Izzat Mahmoud, M.S., Department of Computer Science, College of Science and Mathematics, North Dakota State University, April 2006. Software Metrics: Toward Building Proxy Models. Major Professor: Dr. Kenneth Magel.

The purpose of software metrics is to obtain better measurements in terms of risk management, reliability prediction, cost containment, project scheduling, and improving the overall software quality. Metric tools achieve this by gathering and analyzing the software or the application through a metric tool. This paper describes the process to develop a software metric. It summarizes the different documents of the software development stages. It also describes the product developed. This application is developed specifically for Honeywell's aviation division. Honeywell uses Activity Based Management (ABM) to estimate, track, and manage software projects. ABM requires estimating code by size, in other words, to use lines of code (LOC) or statement lines of code (SLOC) as the basic metrics for predicting software development cost. Metrics assist the process of reverse engineering. The information gathered by those metrics can be used to build a classification model or some formulas that can be used for future predictions. The tool will help us define the requirements for such models. Those models can be used for similar projects in the same industry field

## TABLE OF CONTENTS

## LIST OF TABLES

Table Page

## LIST OF FIGURES

Figure Page

# CHAPTER 1. INTRODUCTION

## 1.1. Problem Definition

This is a project for Honeywell's aviation division. To succeed in the software industry, managers need to cultivate a reliable development process. By measuring what teams have achieved on previous projects, managers can more accurately set goals, make bids, and ensure the successful completion of new projects [1].

The knowledge gathered by software metrics plays an important role in software management. This knowledge can be used to build classification or proxy models that can be used toward future projects. A software metric tool may help us know the required information to build such models. In general, the metrics that are gathered need to be compiled to make some hypothesis about the model.

Honeywell uses Activity Based Management (ABM) to estimate, track, and manage software projects. ABM abstracts an activity into a set of predefined tasks and defines one output as an output of a benchmark size.

In software, the benchmark for one output is one Normalized Module Unit (NMU). For software, a simple ABM model would look like Table 1.1. The activity abbreviation would be the time charge code in the time tracking system filled out by the developer. The standard measurement unit is called NMU. Each development stage will have a specific amount of hours per NMU. This specifies how many labor hours one NMU will cost. For example, planning rate is 2 hours/NMU, testing rate is 10 hours/NMU, and so on.

Table 1.1. Simple ABM model.

| Activity Abbreviation | Hours per Output | Activity Name |
|---|---|---|
| A0 | 1.0 | Planning |
| A1 | 2.0 | High Level Requirements |
| A3 | 3.0 | Low Level Requirements |
| A4 | 4.0 | Coding |
| A5 | 5.0 | Unit Testing |
| A6 | 4.5 | High Level Requirements Based Testing |
| A7 | 0.5 | Build support |
| A8 | 0.75 | Management Support |
| Total Hrs/NMU | 20.75 | |

We hope we will be able to define this model at the end of the project and define what else we may need to build it. These metrics may not fully define the model attributes. Model attributes should be in terms that are more usable to project managers or stakeholders. The research of building such proxy models can utilize the rich concept usage in data mining classification models. This is expected to be the second step in this research.

Honeywell is looking for a metric tool that will work in different operating systems and versions and handle issues that were not handled by the existing earlier metric tool like "//" comments, more than one function in a file, and dealing with C and C++ syntax. They

had an earlier version of a metric tool that run on a specific Linux operating system version. They expect the new application to be flexible, to run manually and automatically; by calling it from another application or script.

They are also looking at comparing earlier and new projects in terms of project resources, complexities, and size. The information will also be used to create a non-Activity Based Management, ABM, estimation model.

This proxy or estimation model will be built similarly to the same way classification models in data mining are built. To make the time or resource estimates, the model will calculate these parameters from earlier projects. The more projects to which the model will be applied, the more confident we will have in the developed model. As the model is built on information from the same company that used nearly the same resources, i.e. ,nearly the same developers, and the same equipment, this will make it more accurate and suitable for their case. It will be useful to compare it to some existing models, within the same field of industry (aviation control systems).

There are also other parameters that are defined by Honeywell that will be measured by our application. For example, they have a definition for the test rank to equal :

Test rank = nestingLevel/2 + Countmcdc/15 + mathCount/40.

And the design rank= Math.Pow(locLines,quad) ,that is the fourth root of the lines of code. The Rank of the function will be calculated the maximum of the above two ranks.

This is a customized formula that is calculated and optimized internally. This makes it harder for any application they purchase externally to be able to calculate such a formula automatically.

By having a graph like Figure 1.1 from earlier projects, we will be able to set standards to tell how many hours certain amount of NMU's should take in a present or future project. There is a certain formula to measure NMU's.

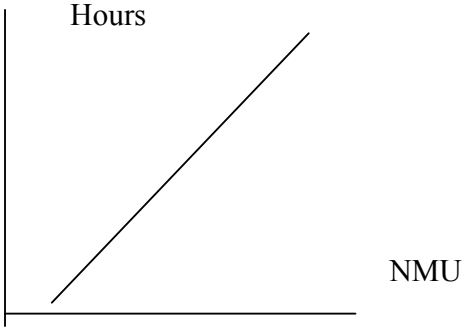



Figure 1.1. NMU/Hour graph.

NMU is calculated using the following equation

NMU = (Rank+1)/2 + Impact, where Impact is a fixed number that is measured and given by a system expert. For example, a new unit is given impact as 2 so that NMU will be Rank+1. Changing documents only, will be given 0.5 and so on.

Rank, and NMU are expected to be measured through our developed application.

**1.2. Approach**

The first step to solve the above problem is to design an application that is capable of parsing a specific list of metrics. This list is written and defined by Honeywell.

The metrics are

1.2.1. Lines of Code (LOC)

Total number of lines of codes in file or method.

1.2.2. Statement Lines of Code (SLOC)

LOC with excluding declarations, global variables, and/or any line that is produced by the application that is used for development.

1.2.3. Maximum nesting level

This reflects the depth of the function or the file. It will be described in details later in the document.

1.2.4. Number of Maximum Coverage/Decision Coverage (MC/DC) conditions

MC/DC is directly related to the amount of conditions in the file or method. It will be described in details later in the document.

1.2.5. Number of mathematical operators

The number of mathematical operators in the file or method. Those operators will be listed later in the document.

1.2.6. McCabe cyclic complexity (optional)

The application will also calculate other formulas like rank, design and test rank. The application is expected to solve the limitations in Honeywell older versions of metric tools that are defined in the problem.

The application is also expected to be used for building a proxy or classification model. Research should be done to determine whether the above metrics are adequate to define and build such model.

**1.3. Related Work**

1.3.1. Code (Implementation) Metrics

Building software metrics is not a new concept. There are many available software metrics in the market. Some of the popular and easy ones to obtain are [2]

1.3.1.1. SDMetrics

Designed in 2003. Available at http://www.sdmetrics.com.

1.3.1.2. Jmetric

Designed in 2000. Available at www.it.swin.edu.au/projects/jmetric/products/jmetric.

1.3.1.3. Together Control Center

Designed in 2002. Available at www.togethersoft.com.

1.3.1.4. Eclipse metric framework plug-in

Designed in 2002. Available at metrics.sourceforge.net.

Eclipse plug-in provides code metrics plug-ins for the IBM Eclipse project. This is may be the most comprehensive and powerful available software metric tool. The following are some of the metrics that can be collected using eclipse:

1.3.1.4.1 Number of Classes

Total number of classes in the selected scope

1.3.1.4.2. Number of Children

This is the total number of direct subclasses of a class. A class implementing an interface counts as a direct child of that interface.

1.3.1.4.3. Number of Interfaces

This is the total number of interfaces in the selected scope.

1.3.1.4.4. Depth of Inheritance Tree (DIT)

Distance from class Object in the inheritance hierarchy.

1.3.1.4.5. Number of Methods (NOM)

Total number of methods defined in the selected scope.

1.3.1.4.6. Number of Fields

Total number of fields defined in the selected scope.

1.3.1.4.7. Lines of Code (LOC)

Since version 1.3.6, LOC has been changed and separated into

1.3.1.4.7.1. Total Lines of Code (TLOC) that counts non-blank and non-comment lines in a compilation unit. Useful for those interested in computed KLOC.

1.3.1.4.7.2. Method Lines of Code (MLOC) which counts and sum non-blank and non comment lines inside method bodies.

This is a summarized evaluation for four other Java metric tools [3]

1.3.1.5. JCSC

It works properly, but too simple, only two checks, and only file-per-file checking.

1.3.1.6. CheckStyle

Reports only errors, does not check result, and gives "wrong" results.

1.3.1.7. JavaNCSS

Gathers small number of metrics, easy to use and has solid user interface.

1.3.1.8. JMT

This is the most serious of the four. It has the highest number of gathered metrics.

In general, software metric tools have issues of accuracy and robustness. Another issue with metrics is that the metric measurements have no standards. There is no universally agreed upon definition for each metric. This makes most of the software metrics suitable only for a particular situation. It is almost the same with our software metric tool. The metrics gathered are defined according to Honeywell's definition of such metrics, which may not be a standard.

### 1.3.2. Design Metrics

Object Constraint Language (OCL) is a good example of a language that is used for design metrics. Some of the software design metrics are [4, 5] given below.

#### 1.3.2.1. Number of parameters

It tries to capture coupling between modules.

#### 1.3.2.2. Number of modules and number of modules called

It is useful for estimating the complexity of maintenance.

#### 1.3.2.3. Fan-in

It refers to the number of modules that call a particular module.

#### 1.3.2.4. Fan-out

Fan out for a module is how many other modules it calls. High fan-in means many modules depend on this module. High fan-out means module depends on many other modules.

#### 1.3.2.5. Data bindings

Data bindings Reflects possibility that two objects may communicate through the shared variable.

#### 1.3.2.6. Cohesion metric

Construct flow graph for module. Each vertex is an executable statement. For each node, record variables referenced in statement. If a module has high cohesion, most of variables will be used.

In Chapter 6, Table 6.1. Lists some of the design metrics that can be gathered from the design or the UML diagrams.

### 1.3.3. Requirement Metrics

This is a subject in its early progress. These are some of the requirement metrics that can assess in software requirement evaluation

1.3.3.1. Function Points

Count the number of inputs and output, user interactions, external interfaces, and files used. It is used to predict size or cost and to assess project productivity.

1.3.3.2. Number of requirements errors found

Count the number of errors that is found in the requirement specifications.

1.3.3.3. Change request frequency

It is used to assess the stability of requirements. Frequency should decrease over time. If not, requirements analysis may not have been done properly [5].

**1.4. Innovations**

This application has its own parser that is designed and customized for our specific purpose. Yet, the application is capable of parsing other types of codes, and not only C or C++ code. Most available metric tools target one specific metric, and very few of those tools are able to gather a comprehensive list like the one we designed.

This application has evolved into two versions: A Windows version that can run manually under Windows, and a DOS or Console version that can run manually or automatically as part of a script or a make-file. Another point that makes this application fits for automation process is that there is one entrance operation (Count) for the whole application. This operation will trigger the parser and all the metrics.

The other advantage for this application that may make it unique is that it collects the metrics on both the class and the file level along with the function level. According to some of the available metric tools, those tools gather metrics on the class level only. (Some

tools that do on both, levels like JavaNCSS [3] do it for very limited metrics, only LOC and NOC; Number of classes).

Some tools like JCSC [3] cannot scan more than one file at a time, and cannot scan the whole folder recursively. Our tool has no limit on the amount of files or folders it can scan.

The gathered data is saved as a (.csv) comma delimited file type. This enables the data to be imported to any database.

The focus of our tool is on static metrics. There are some tools that gather the dynamic or the run-time metrics of the software application.

### 1.5. Contributions

I was the only person involved in the design and development of this application. Supervision and feedback were provided by Daniel Henrich from Honeywell, Dean Knudson, and Dr Ken Magel from NDSU.

# CHAPTER 2. BRIEF USER MANUAL FOR THE APPLICATION

## 2.1. Introduction

The purpose of this chapter is to introduce to the potential users of SWMetrics tool the basic functionalities and usage of this tool. This will include both versions: the Windows and the Console.

SWMetrics is software that is used to run on software code to gather some metrics display the results, and save them in a suitable format. The metrics are LOC, SLOC, Math counts, Maximum nesting, MC/DC, and Cyclic complexity.

The output file is in a comma delimited format (.csv) that can be imported to any type of database. The application does not require any type of training.

## 2.2. Functionalities

The two versions of the application have only one main operation from which to start the entire application. This operation is the “Count” method or operation.

## 2.3. Hardware and Software Requirements

The Windows version will run on a Windows 2000 or above operating system. It will not require extra hardware requirements other than the operating system requirements. If the .NET environment is not installed on the system, the application requires .NET Framework Version 2.0 Redistributable Package that is available to download from the Microsoft website at http://msdn.microsoft.com/netframework/downloads/updates/default.aspx.

The Console version is expected to be portable and platform independent. This may have some limitation or requirements from the .NET package. It may require the same compact framework above, or what may work instead for other platforms.

### 2.4. Setup and Installation

The present application form is an exe file that can be run by simply clicking on it. Once the application is completed, a setup application maybe created that requires a standard setup procedure.

The Console version of the application can run manually or can be called from a script.

### 2.5. Performing a Standard Run

#### 2.5.1. Windows Version

Starting the SWMetrics.exe Window version will first display the following Window (Figure 2.1).

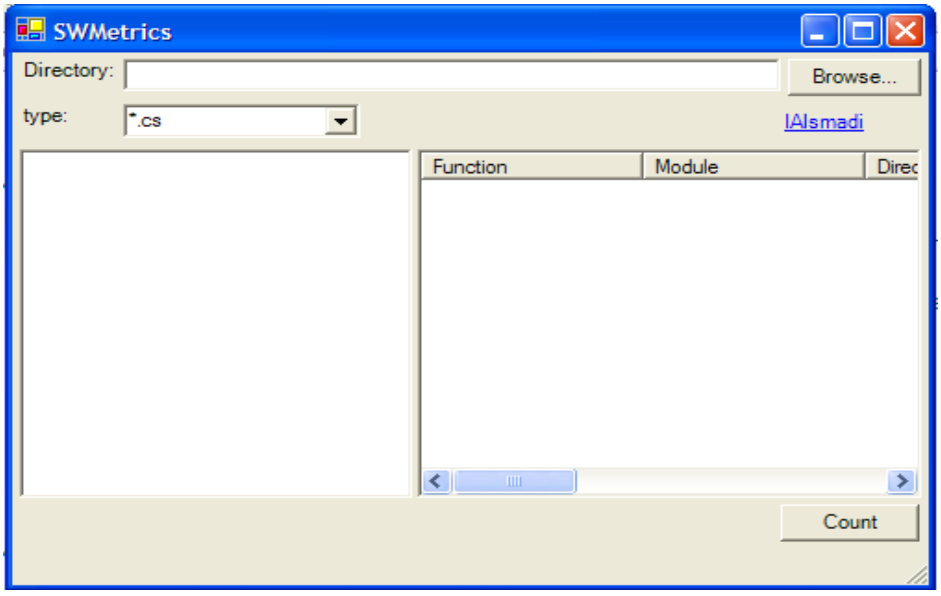



Figure 2.1. SWMetrics main window GUI.

Before starting the counting process, by pressing the Count Button, we need to "Browse" to select the folder(s) that has the folders and/or files from which to parse metrics. From type, we will then choose the type of the code. This implies that we can

parse different types of codes; not only C or C++ types (.h, .c and .cpp). But, this also means that we can parse one type at a time (except for C and C++, which are combined as one type).

Then a saved file dialog will be prompted for the user to select where to save the parsed metrics (Figure 2.2).

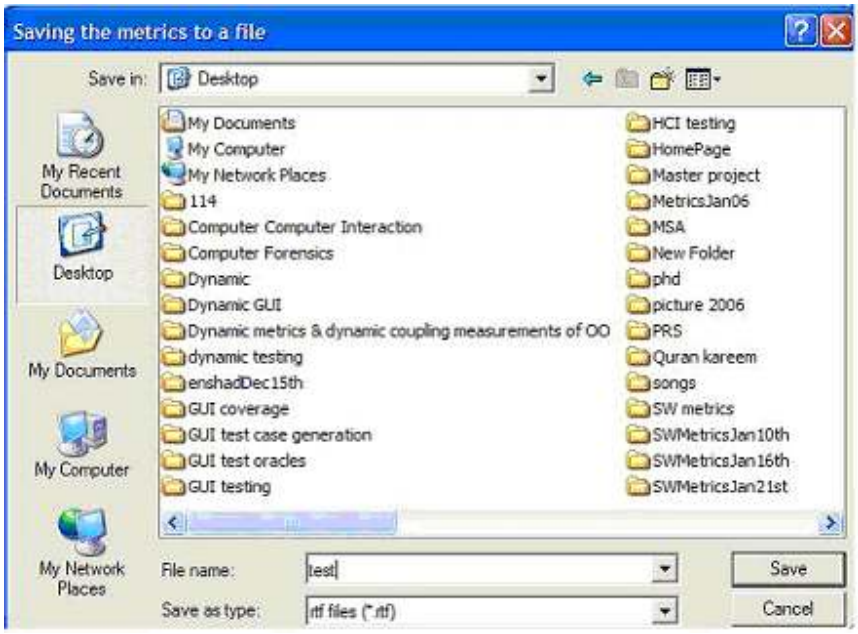



Figure 2.2. Saving metric to a file.

After those selections, the metric process is ready (Figure 2.3).

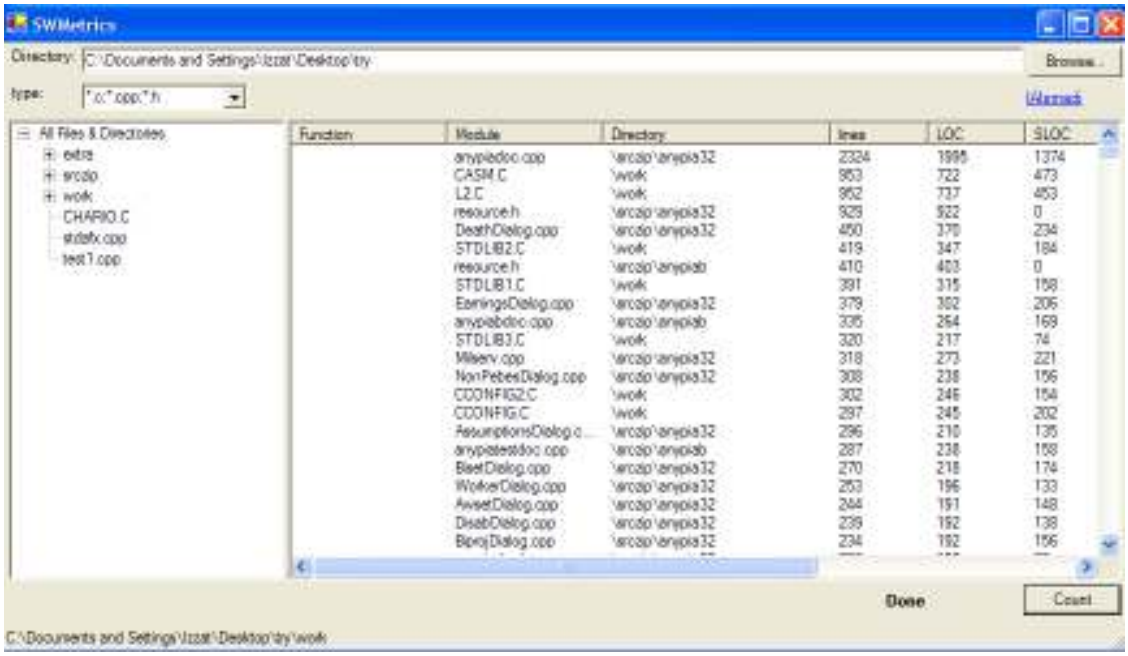



Figure 2.3. Running SWMetrics.exe screenshot.

The data will be shown on the screen and will also be saved to a (.csv) file. There is a progress bar at the bottom left corner that shows the present directory that is being parsed. A label next to the count will show “Counting” during the process, and “Done” at the end of a successful process.

2.5.2. Console Version

The Console version can be run manually by typing:

SWCMetrics.exe Directory-Name Destination-File-Name (optional).

For example, to run the metrics on a folder in D drive called “testDirectory” and to save the parsed metrics to a file “D:\test.csv”, the user should type

**SWCMetric.exe D:\testDirectory D:\test.csv**

This is assuming we have SWCMetric.exe in the current directory. As the second argument is optional, we can leave it blank or ignore it. For example

**SWCMetric.exe D:\testDirectory**

This will cause the application to save the parsed metrics in a default file with the name of today's date, i.e. "3-30-2006.csv", in the same current application directory. If we want to run this application automatically, we simply need to feed the above line through our script or make-file in order for the application to start.

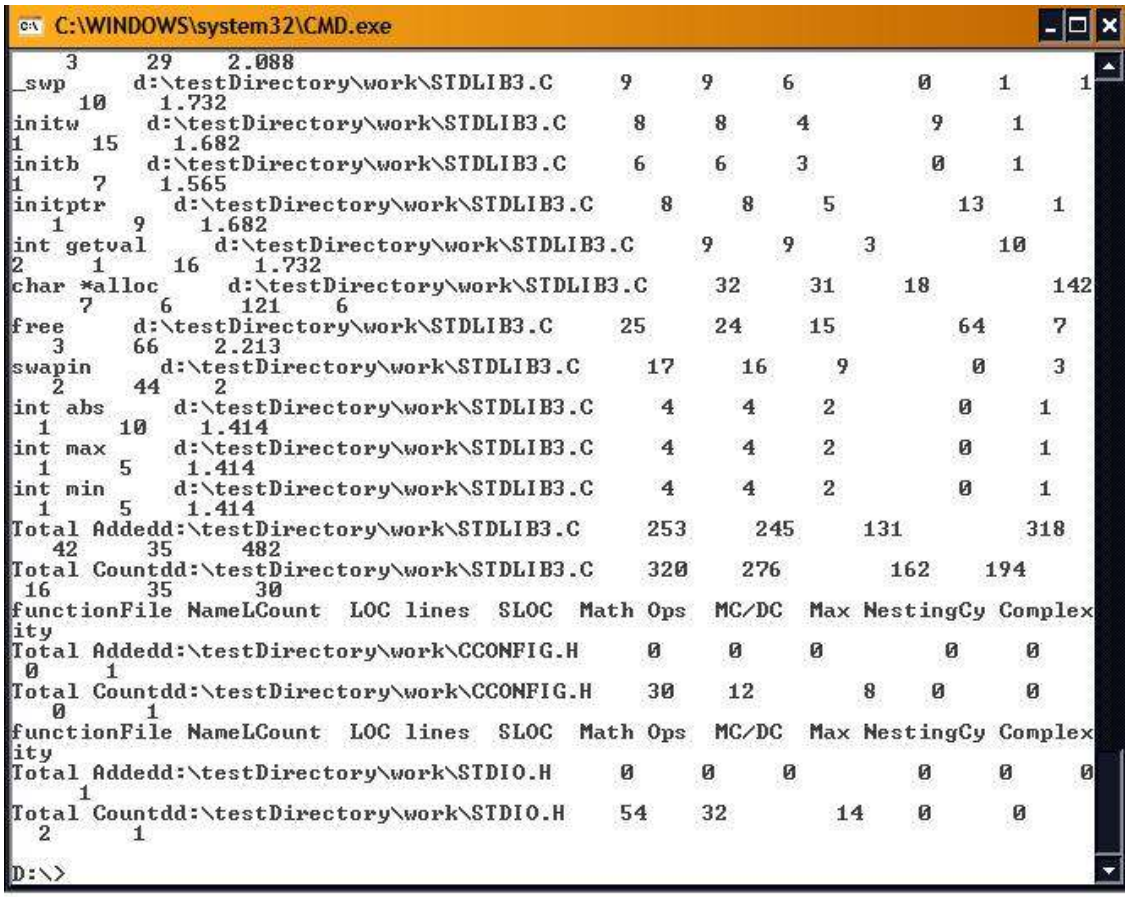
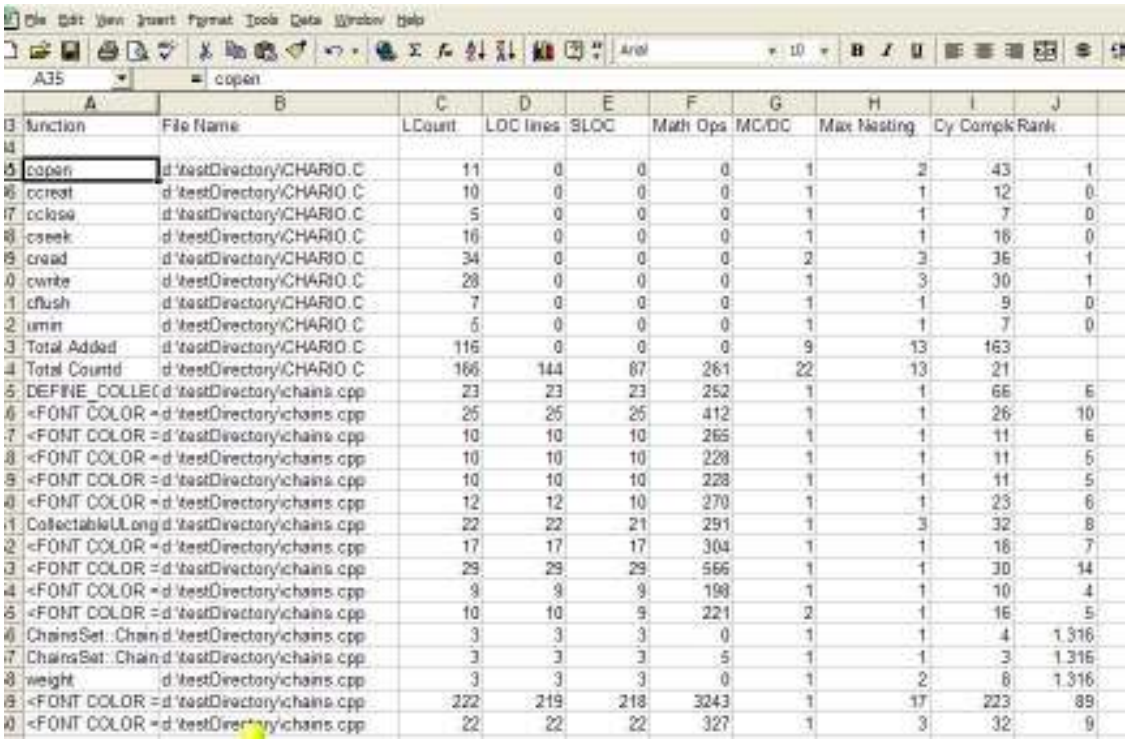



Figure 2.4. Running SWCMetrics.exe screenshot.

The output will be on the Console and to the destination file. Below is a copy of the output file opened in MS Excel (Figure 2.5).



| | A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | function | File Name | LCount | LOC lines | SLOC | Math Ops | MC/DC | Max Nesting | Cy Compl | Rank |
| 4 | | | | | | | | | | |
| 5 | copen | d:\testDirectory\CHARIO.C | 11 | 0 | 0 | 0 | 1 | 2 | 43 | 1 |
| 6 | ccreat | d:\testDirectory\CHARIO.C | 10 | 0 | 0 | 0 | 1 | 1 | 12 | 0 |
| 7 | cclose | d:\testDirectory\CHARIO.C | 5 | 0 | 0 | 0 | 1 | 1 | 7 | 0 |
| 8 | cseek | d:\testDirectory\CHARIO.C | 16 | 0 | 0 | 0 | 1 | 1 | 16 | 0 |
| 9 | cread | d:\testDirectory\CHARIO.C | 34 | 0 | 0 | 0 | 2 | 3 | 36 | 1 |
| 0 | cwrite | d:\testDirectory\CHARIO.C | 28 | 0 | 0 | 0 | 1 | 3 | 30 | 1 |
| 1 | cflush | d:\testDirectory\CHARIO.C | 7 | 0 | 0 | 0 | 1 | 1 | 9 | 0 |
| 2 | umin | d:\testDirectory\CHARIO.C | 5 | 0 | 0 | 0 | 1 | 1 | 7 | 0 |
| 3 | Total Added | d:\testDirectory\CHARIO.C | 116 | 0 | 0 | 0 | 9 | 13 | 163 | |
| 4 | Total Countd | d:\testDirectory\CHARIO.C | 166 | 144 | 87 | 261 | 22 | 13 | 21 | |
| 5 | DEFINE_COLLEC | d:\testDirectory\chains.cpp | 23 | 23 | 23 | 252 | 1 | 1 | 66 | 6 |
| 6 | <FONT COLOR = | d:\testDirectory\chains.cpp | 25 | 25 | 25 | 412 | 1 | 1 | 26 | 10 |
| 7 | <FONT COLOR = | d:\testDirectory\chains.cpp | 10 | 10 | 10 | 265 | 1 | 1 | 11 | 6 |
| 8 | <FONT COLOR = | d:\testDirectory\chains.cpp | 10 | 10 | 10 | 228 | 1 | 1 | 11 | 5 |
| 9 | <FONT COLOR = | d:\testDirectory\chains.cpp | 10 | 10 | 10 | 228 | 1 | 1 | 11 | 5 |
| 0 | <FONT COLOR = | d:\testDirectory\chains.cpp | 12 | 12 | 10 | 270 | 1 | 1 | 23 | 6 |
| 1 | CollectableULong | d:\testDirectory\chains.cpp | 22 | 22 | 21 | 291 | 1 | 3 | 32 | 8 |
| 2 | <FONT COLOR = | d:\testDirectory\chains.cpp | 17 | 17 | 17 | 304 | 1 | 1 | 18 | 7 |
| 3 | <FONT COLOR = | d:\testDirectory\chains.cpp | 29 | 29 | 29 | 566 | 1 | 1 | 30 | 14 |
| 4 | <FONT COLOR = | d:\testDirectory\chains.cpp | 9 | 9 | 9 | 198 | 1 | 1 | 10 | 4 |
| 5 | <FONT COLOR = | d:\testDirectory\chains.cpp | 10 | 10 | 9 | 221 | 2 | 1 | 16 | 5 |
| 6 | ChainsSet::Chain | d:\testDirectory\chains.cpp | 3 | 3 | 3 | 0 | 1 | 1 | 4 | 1.316 |
| 7 | ChainsSet::Chain | d:\testDirectory\chains.cpp | 3 | 3 | 3 | 5 | 1 | 1 | 3 | 1.316 |
| 8 | weight | d:\testDirectory\chains.cpp | 3 | 3 | 3 | 0 | 1 | 2 | 8 | 1.316 |
| 9 | <FONT COLOR = | d:\testDirectory\chains.cpp | 222 | 219 | 218 | 3243 | 1 | 17 | 223 | 89 |
| 0 | <FONT COLOR = | d:\testDirectory\chains.cpp | 22 | 22 | 22 | 327 | 1 | 3 | 32 | 9 |

Figure 2.5. The Excel parsed file screenshot.

# CHAPTER 3. THE DEVELOPMENT PROCESS

This chapter will describe the development process that is used to build the SWMetrics tool. Although there was no previous decision to follow any specific known software development process, the knowledge and experience of those processes are used to develop this application.

## 3.1. Challenges

There were some challenges and difficulties while developing this project that may play an important role on the way it is developed.

Other than the typical limited availability of time and resources that I have, I worked on this project by myself as a developer. Working alone on a project may have a risk of ignoring the same error again that needed to be noticed by another person.

Another challenge is the nature of the project. This project is related to an aviation company that considers all its information as “private” and limited to its employees. This was a challenge because there was only one person to contact. It was also impossible to access some of the company documents that could have discussed the problem, the needs for it, and/ or any other useful information that could have helped in identifying the real problem, and gathering the requirements.

In testing, the tool has to run on some of the available free C or C++ code and not on the actual company code. The testing process on the company code is supposed to be run by the company employee. The results of such tests are to be received unofficially and informally through emails. This was another challenge to the testing stage.

The definition for the problem is defined according to the information received from the client. We planned some meetings to discuss the requirements, but none of the planned meetings actually occurred until later in the development stage. The limitations of distance and weather played a major role in postponing such meetings.

Because communication was a real barrier, and the understanding of the problem has been developed with time, rather than having the whole picture before starting the project, the decision was to follow some type of Software evolving or iterating process. Starting earlier in time, prototypes were developed almost weekly. The intention was to get the client feedback to clear any misunderstanding of the problem while in earlier stages.

Another important challenge was to determine the correct definitions and algorithms for the required metrics. Software metrics in general, suffer from a problem of not having standards for their definitions. Finding the right algorithm to implement a certain definition was one of the innovative approaches I have to define. Tuning such algorithms was a coordinated process with the client trying to reach the expected goal.

**3.2. Initial document**

The first step was writing the project initiation document. The intention was to get all contributing members of the project to agree on the main scope and features required.

The following is a summarized version of the project initiation document.

3.2.1. Scope/Vision

By producing an application that can estimate the parameters mentioned by the client, Dan from Honeywell, we should be able to study many aspects about coding or programming.

This is a project to design a software metric application that will give the project sponsors the ability to do better cost/time predictions for future projects by studying or reverse engineering earlier developed codes.

The proposed metrics are listed in Table 3.1. The first five, which are required for, Personal Software Process (PSP), and Activity Based Management (ABM), will be accomplished as the main committed tasks. The information gathered should be saved to a log file. The other optional metrics are open for modifications and completion upon available resources.

### 3.2.2. Goals and Objectives

The application should be able to measure: LOC, SLOC, number of mathematical operators, max nesting level, and number of MC/DC as described as the committed requirements (Table 3.1).

In general, the process of metrics gathering is an earlier step for code analysis and mining. The ultimate goal is to develop a predictive model or formulas that can be used to support decision making. Designing a good software metric tool is a very important step towards software test automation.

### 3.2.3. Project Setup

The application will be written in C# as a Windows application. A Console application will also be developed. The client will test the application on the existing company codes to evaluate the application. The tested code is written in C and C++. The application will able to deal with other kinds of codes (This is an optional suggested feature that may make the project useful for reverse engineering different type of applications).

Table 3.1. The Commit matrix.

| Metric Name | Status | |
|---|---|---|
| | Commit | Target |
| Lines of Code (LOC)<br>Non-blanc, non comment line | Yes | |
| Statement Lines of Code (SLOC)<br>Only statement lines, LOC without declarations, macro-definitions, begin-end | Yes | |
| Number of mathematical operators | Yes | |
| Number of mathematical operators per statement with a math operator | Yes | |
| Max nesting level | Yes | |
| Number of MC/DC conditions | Yes | |
| Number of function calls total | | Yes |
| McCabe- cyclic complexity | | Yes |

3.2.4. Project Risks

The time availability, to finish all the required metrics was not planned very well. Some of the metrics may require a complicated algorithm that will require additional time to be optimized.

3.2.5. Commit Matrix

Table 3.1 shows the committed matrix of metrics to gather.

3.2.6. Deliverables

The features or functions listed in Table 3.1 will be the deliverables for this application. The client will be notified and will receive a prototype upon finishing each function. The documents delivered will be: Initial document, plan/ schedule, requirement document, design document, test plan, build/configuration info, and a brief user manual.

3.2.7. Assumptions

3.2.7.1. The definitions for some of the metrics are what the client defined. For example, LOC and SLOC have many definitions in the software testing industry. The client has a specific one too.

3.2.7.2. The project time is the semester period. The project actually started in December 2005, a month earlier than the semester.

3.2.8. Dependencies and Constraints

3.2.8.1. Time constraints may limit the amount of features that can be finished.

3.2.8.2. The client will run and test the code prototypes on the company existing code.

3.2.9. Available Resources

I was the only developer involved in the project.

3.2.10. Signatures

- Izzat Alsmadi ----------
- Dean Knudson. ----------
- Daniel Henrich ----------

The project initiation document took several attempts be agreed upon. For example, in an earlier email, the Commit Matrix includes more functions to count. Those functions or metrics were not defined very well. As those functions were not required by PSP or

ABM, and as time will not allow doing or clarifying them, they were discarded as a commit task for this project.

Working in the first Window version prototype was started right immediately after delivering the project initiation document, and while working in the requirement document.

### 3.3. The Iterative Process

"The basic idea behind iterative enhancement is to develop a software system incrementally, allowing the developer to take advantage of what was being learned during the development of earlier, incremental, deliverable versions of the system. Learning comes from both the development and use of the system, where possible. Key steps in the process were to start with a simple implementation of a subset of the software requirements and iteratively enhance the evolving sequence of versions until the full system is implemented. Design modifications are made in every iteration and new functional capabilities are added [6].

This is exactly what the development process of this project adopted. The project started by delivering a simple prototype or implementation of a subset of the software requirements in order to be enhanced iteratively with a sequence of versions until the full system was implemented.

This first prototype was to get a basic Graphical User Interface (GUI) to agree upon, as a shell, and then to choose a basic functionality to apply. Although this may look like an easy part to do, yet it proved to be hard to implement. The major step in this project was actually building the parser. The parser is the part of the application responsible for collecting the files and dividing them by functions. It was not possible to implement any functionality or requirement without actually building the parser.

### 3.4. The Parser

The decision whether to use an available C/C++ parser or to build it, was an important and major decision to make during this project.

There are several available free C/C++ parsers. The choice was whether to spend some time and find the suitable parser, study, implement, and customize it for our own purpose. The alternative was then to build our own parser as part of the project.

Choosing the first alternative, it may take about two weeks; looking for the right parser and studying it before reaching the point to determine whether this specific parser is actually a good choice, or not. On the other hand, building a parser, will take some extra time, yet it will have a specific time that a result is expected at the end. The other advantage to use our own parser is that it will be easier to customize and incorporate this parser within the application.

#### 3.4.1. The Parsing Algorithm

Building a reliable parser requires a well defined algorithm. It also requires an extra time to be tested and verified. I started by parsing a simple file with one or two functions, and displaying the result or the name of the functions in a any type of text editors; i.e. list box, textbox, etc. A procedure was developed to determine where methods or functions start and end. There was also an algorithm to parse the name of the functions. The main symbols that are assigned with functions, especially in C/C++, are the "{","}"," ()". The name of the method first will be followed by the brackets "()". The method will start by the first opening parenthesis "{". The same method will reach the end by the last closing parenthesis "}" that sets the parentheses count back to zero i.e. opening parenthesis will increment the count while closing parenthesis will decrement the count.

The application will first preprocess and excludes the comment and blank lines from the file.

3.4.2. Example

Figure 3.1 is a small code sample that demonstrates the above algorithm

```
/* these things are used to manipulate the input and output files.
*/ single comment line
char hexname(MAXLINE), crlname(MAXLINE);    Declaration.
FILE *hex;  int crl;
/*  These things are used to manipulate
the core buffer.                    */  Multi comment lines.
char *bbase, *bend, *bsize;
int hgetc()              brackets .. look for an opening parenthesis soon, parse the
name from here.
{          the method actually starts, increment count.
int j;
j = hgetn()<< 4;  chks += (j += hgetn());  return j;
}      The count is decrement and reaches zero. End of method.
/*  This is reserved for the use of the hgetn() function.                    */
makename(new,old,ext,flg) brackets .. look for an opening parenthesis soon,
parse the name from here.
char *new, *old, *ext;
int flg;
{ the method actually starts, increment count.
while (*old] {  increment count , count=2.
if (*old == '.') { strcpy(new,flg ? ext : old); increment count , count=3.
return; } decrement count , count=2.
*new++ = *old++;
} decrement count , count=1.
Strcpy(new,ext);
}  decrement count , count=0. End of function.
```

Figure 3.1. Code sample with some metrics demonstration.

### 3.5. Testing

The structural, integration and functional testing are performed while developing, on some free codes. Functional and usability testing on the other hand, is performed by the client.

Testing was iterative just like the development process. For each prototype, testing was first performed while developing. Many free C/C++ code were downloaded from the internet to be used as a test oracle. The second test for each prototype was performed by the client on the company code.

A small database of C/C++ code was built to work as the project test oracle. The verified results of this database are saved and updated. Whenever there was a new prototype, testing was performed on this database to make sure that earlier functionalities or features continued working the way they should.

### 3.6. Process Evolution

#### 3.6.1. Prototype Release

The prototype release was for one of two reasons

##### 3.6.1.1. Implementing of a new metric or feature.

##### 3.6.1.2. Resolving some issues after a client feedback.

Figure 3.2 shows the number of prototypes delivered for this project. The first prototype was finished and delivered on Dec. 9th 2005. The latest delivered prototype, which became the final product, was dated April 28th 2006.

The versions that have the "C" letter after SW in the name are Console versions. The versions that do not have it are Windows version. The last version released from the Windows version was in Jan 27th 2006. Later work and releases were from the Console version. The final deliverable product that all agreed to be was of the Console. Windows version is considered legacy and is not going to be fully tested.

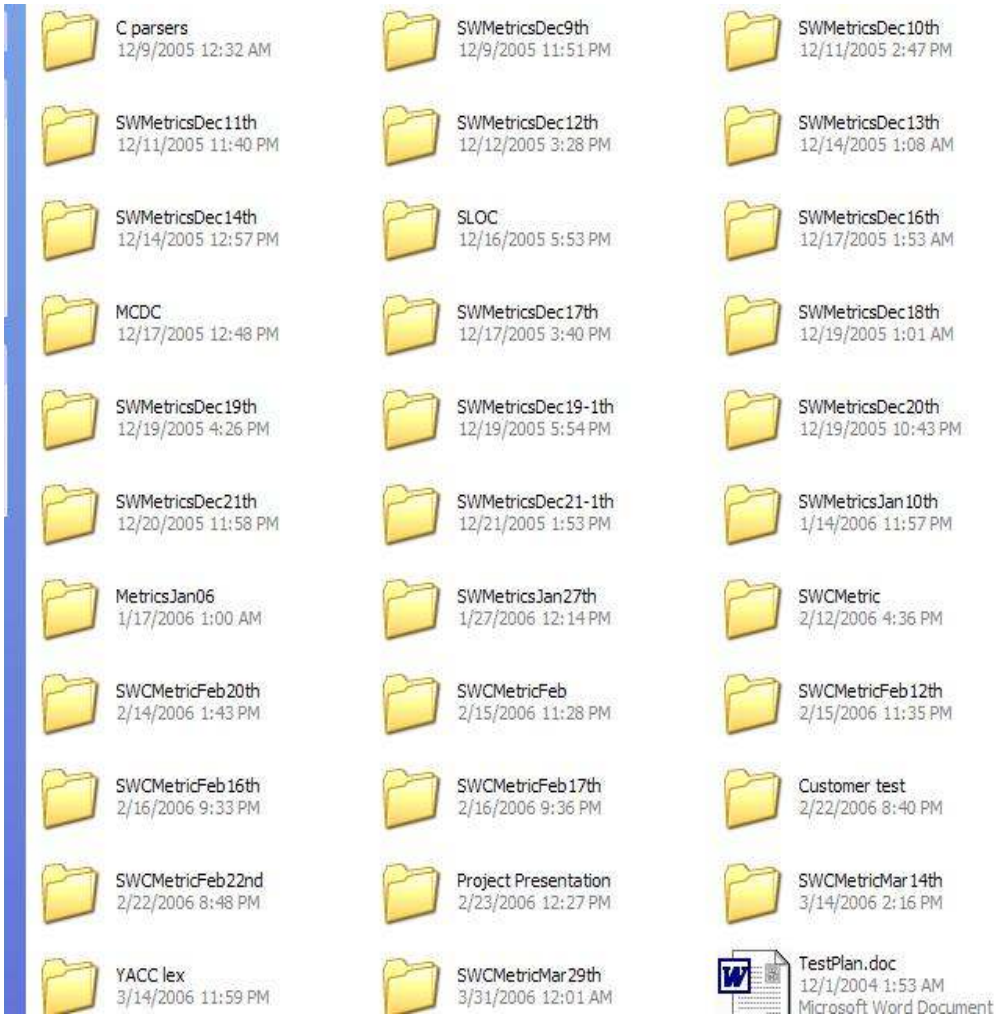



Figure 3.2. Prototype evolution.

### 3.6.2. Two Versions

The evolving of two versions of the application resulted in extra work and was time consuming. Having two versions for the product was a result of some miscommunication.

The requirements stated that the application should run as part of a make-file or a script automatically. It also stated that the application should be platform independent. Those two reasons were supposed to support the decision to build a Console and not a

Window application. Yet, the application started by considering a Window application for many reasons. The following are some of them:

3.6.2.1. In that earlier stage, the project wasn't expected to be of its level of complexity in structure and algorithms. This made the impression that it would be very easy to make a Console version of the project without much extra effort. The Windows version was expected to make it easier dealing with the application.

3.6.2.2. The early prototype main task was to design a parser, an in important feature the application is expected to have, is dealing with many folders and sub folders at the same time. Since Window forms make it easier to build trees of directories and sub directories, and visualize them. Building such trees in a Console version is less feasible and harder to see and deal with. As such, building the Windows version was a more practical choice.

3.6.2.3. As requirements stated that the application should run part of a script, it wasn't clear whether the Console version would make it easier to do so.

Although that was an extra experience in software management, it was a result of ineffective communication.

Communication plays an important role in the development process. Failing to communicate well during any stage, especially in the iterative process, usually leads to a major problem.

Returning to comparing the adopted development process to those standards, most iterative processes suggest that the iteration should start at a later stage and not iterating the whole cycle. For example, in the Unified Software Development Process (USDP) [7], the process has two initial straight forward stages: inception and elaboration, and two iterative stages: construction and transition. In our case, even the elaboration phase happened

iteratively. This approach maybe more of an Agile approach where little requirements are gathered. Then a cycle starts with little design, coding, testing, and evaluation and starts again.

In each week, there are tasks from every stage of the development process. Requirements are gathered from the feedback of an earlier version, or from adding a new feature with the newer releases. The detailed design is then followed. After that, coding for this specific feature is developed and added to the latest prototype.

At the end before the release, test is run on the test oracle, and then the newer prototype is uploaded to the shared folders (Twiki or my NDSU web page). The client would test and evaluate the new release on there own test oracles.

### 3.6.3. Adopting the Scrum Process

In the Scrum development process (Figure 3.3), there are three main stages: High level planning, Sprint cycle and closure. The Sprint cycle is an iterative cycle of about three to four weeks, in which the actual development of the product is accomplished. It begins with a Sprint planning meeting to decide what will be achieved in the current Sprint. A Sprint is closed with a Sprint review meeting where the progress made in the last Sprint is demonstrated, the Sprint is reviewed, and adjustments are made to the project as necessary.

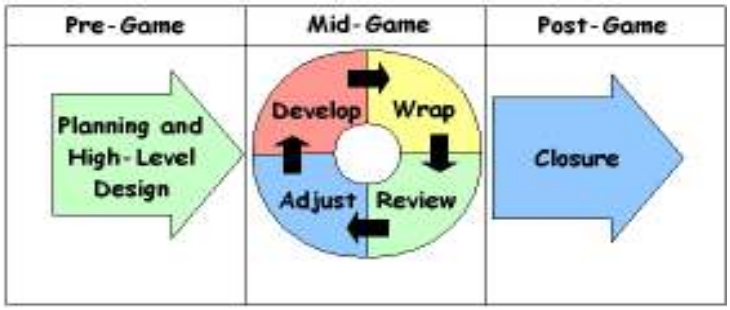



Figure 3.3. The Scrum process [7].

The Sprint cycle is repeated until the product's development is complete. The product is complete when the variables of time, quality, competition, and cost are at a balance.

3.6.3.1. Develop the product further: implement, test, and document.

3.6.3.2. Wrap up the work: get it ready to be evaluated and integrated.

3.6.3.3. Review the work accomplished in this Sprint.

3.6.3.4. Adjust for any changes in requirements or plans [7].

I decided to adopt the Scrum process on the individual level. The Sprint cycle in our project was between one to two weeks instead of the three to four weeks standard in Scrum. We have also three main stages: high level planning, a phase in which all members of the project worked together. We then have the sprint cycle, and finally the disclosure. There was no actual Sprint meeting as explained in the actual process. We had reviews for each cycle at the end of it. The feedback of this cycle will be used as an input for the next cycle.

Although Scrum or Agile development processes in general take a major focus on communication that should be run on a daily basis, yet in our project it was not possible to communicate on that daily basis. A key principle of Scrum is its recognition that fundamentally empirical challenges cannot be addressed successfully in a traditional process control manner. As such, Scrum adopts an empirical approach that acknowledges that the problem cannot be fully understood or defined, focusing instead on maximizing the team's ability to respond in an agile manner to emerging challenges" [8]. In our project case, the recognition that the challenges were hard to estimate at the beginning was a good reason to adopt the Scrum process.

### 3.7. Project Structural Refactoring

The project went through an evolving process in its design, coding, and testing before it reached its final format. It was clear that the structure of the project needed a refactoring process to make it more reusable, understandable, and readable. As described above, the functionalities or the software metrics were developed one per prototype. There was a substantial portion of the code that was repeated in some way or another. The parsing process was also repeated in different places. This was expected to affect the overall performance of the application.

The refactoring process was not an easy task to achieve especially as we were approaching the end of the project implementation. Before the process of refactoring started, we decided to build a better and more accurate test oracle that would act as the project testing backbone. This test oracle suit was very important to make sure that, whenever a refactoring step is implemented, nothing of the actual expected functionalities of the application has been broken. The refactoring process goal is to modify the structure while preserving the project functionalities.

The gathered metrics are collected in an XML or comma delimited (".csv") file format. A procedure is developed to compare the results before and after each time the application is executed.

The refactoring process is started by splitting the code into classes that share the same functionalities and making sure the main file or class has a minimum amount of code. Figure 3.4 shows the final version of the component diagram.

Figure 3.4. High-level component diagram.

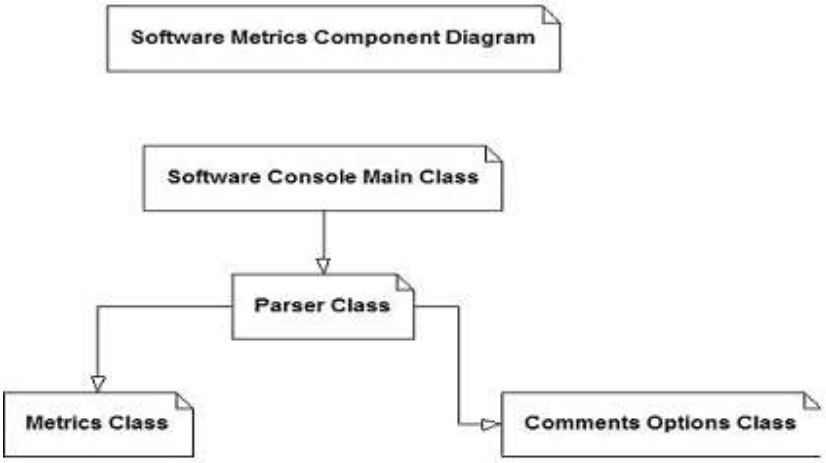



Figure 3.5 is a simplified version of the activity diagram that shows the activities between the different classes or components. The Console Main class is the class that has the public operation “Main” that is called by the user. This class calls the public operation “Count” from the parser class. The count operation is a centralized point where most other methods from other classes are called after parsing the file(s). This ensures the parsing process is called once and hence improves the over all performance of the application.

Although the requirements were collected and processed gradually, the continuous involvement; communication, testing and feedback from the client was a major factor of the success of this project. If a traditional approach of the development process was followed, there would be a high risk of a project failure. It would end up consuming a lot of time and effort developing something the client may actually not expect or look for. This project had some ambiguity in defining the detailed requirements. There was a need to know at a very early stage of the project, if it was on the right track.

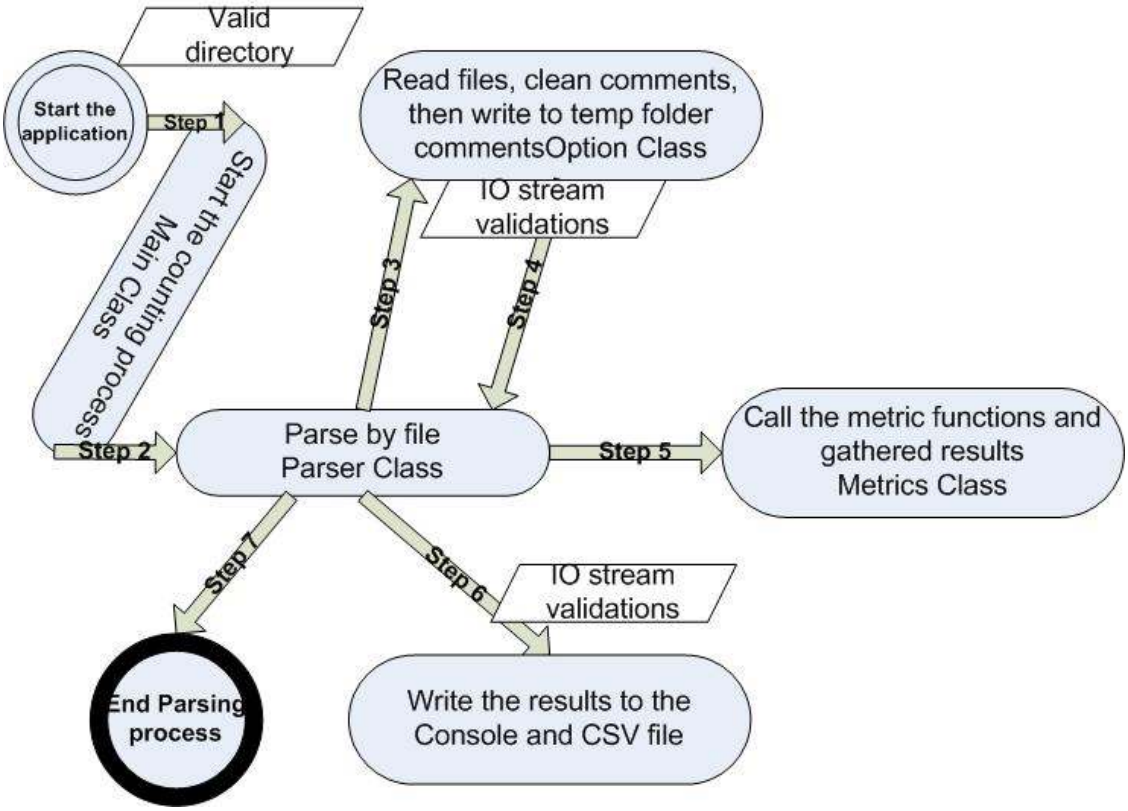



Figure 3.5. Classes' interaction activity diagram.

If there is something that could have been done better, it was in the communication phase. This could have saved the time and effort it took to design two versions of the application.

### 3.8. Innovative Aspects

Some of the innovative aspects of the project are as follows

#### 3.8.1. The Parser

Designing our own parser was a challenging task. The accuracy of the parser was a big concern. The process and algorithm followed in building it, were innovative ones. It is not expected to be a 100 % accurate, yet it does provide a very high accepted accuracy with

an acceptable response time. Considering the time and resources available, this was a major success in the project.

### 3.8.2. The Metric Algorithms

As explained earlier, a problem to this project is the fact there is no standard definition for static software metrics. Honeywell has there own definition for each metric that the tool has to collect. Developing the right mathematical algorithm that implements or represents a certain definition was a major task to achieve.

### 3.8.3. The Overall Development Approach

The over all approach adopted in building this application was important, giving the limitations in resources, or the limitations in the communication and availability of the users.

# CHAPTER 4. THE DESIGN OF THE APPLICATION

## 4.1. Introduction

This chapter describes the process followed for the design and implementation phases of the SWMetrics tool. It provides the programmer with enough information to successfully code all the modules and functions necessary in delivering this application.

This process is intended to take into account the varying levels of experience of people involved in the software development life cycle.

The SWMetrics tool Detailed Design Specification (Code-To) describes the design of the instrument software in sufficient detail to permit code development.

## 4.2. Purpose

This document applies to the detailed design of the SWMetrics tool. Special attention has been given to highlighting critical software design components and overall software system development issues based on object-oriented design techniques.

This document also describes the major design decisions, concepts, architecture, programming language, and development tools used in developing the SWMetrics deliverables.

The purpose of this product is to establish the application “SWMetrics” based on the following constraints and client requirements

4.2.1. A flexible application that can run on different platforms, and has the ability to run manually or part of an automated process.

4.2.2. The ability to gather the listed Software metrics in the requirement document.

4.2.3. Over come all the limitations in the client previous metric tool.

### 4.3. Document References

This document relies on some other documents. Table 4.1 lists these documents.

Table 4.1. Reference Documents

| Part Number | Version | Title |
|---|---|---|
| 1 | 1 | SWMetrics Requirements Document. |

### 4.4. High-Level Design

Figure 4.1 is the high-level component or context diagram.

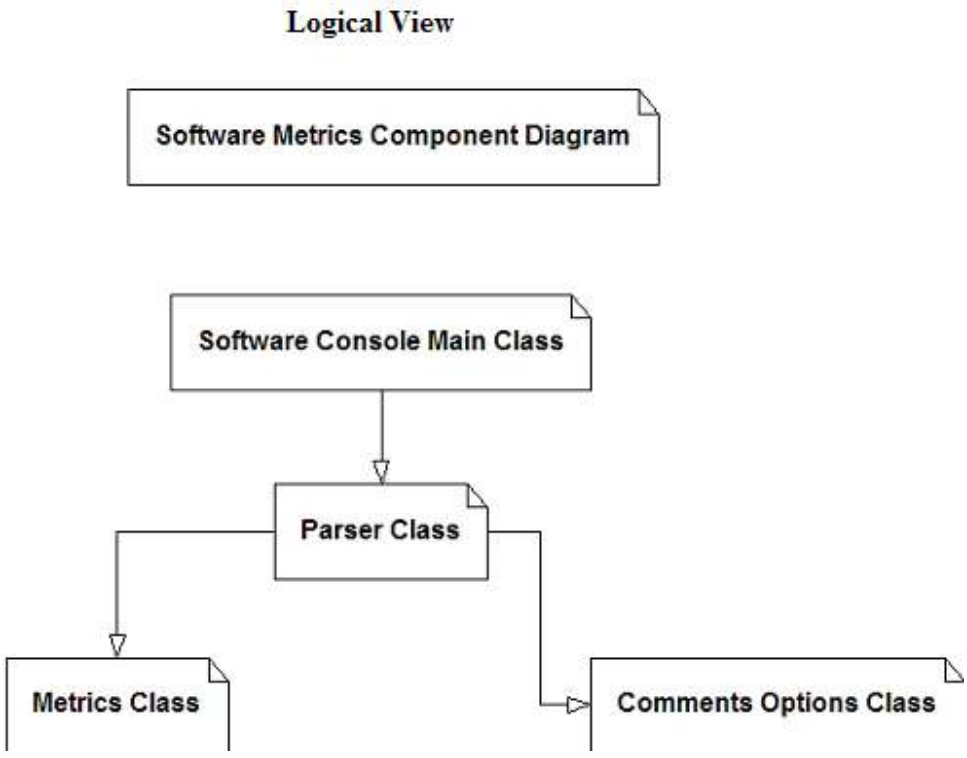



Figure 4.1. SWMetric context diagram.

4.4.1. Module List

Our project modules, as seen in the context diagram, are

4.4.1.1. The Console main application module

4.4.1.2. The parser module

4.4.1.3. The comments option module

4.4.1.4. The metrics module

4.4.2. Use Case Scenarios (Primary with Secondary sub tasks)

4.4.2.1. Running the application (Console version), Primary Use Case

Following are the steps to run the application in Console mode.

4.4.2.1.1. Start the application by typing the Console name.exe on the Console.

4.4.2.1.2. Type the name of the directory which has the files to run the metrics on.

4.4.2.1.3. Type the destination file full name (optional). If no name is listed, the default name, today's date, is assumed.

4.4.3. Use Case Diagrams

4.4.3.1. Primary use case

Figure 4.2 is the primary use case diagram for SWMetric Windows's version. The use case for the Console version is not listed as it is very similar. The difference between both diagrams is that in the Console version, the user does not need to select the extension of the code the application is analyzing. The Console version is able to differentiate between the different source code types.

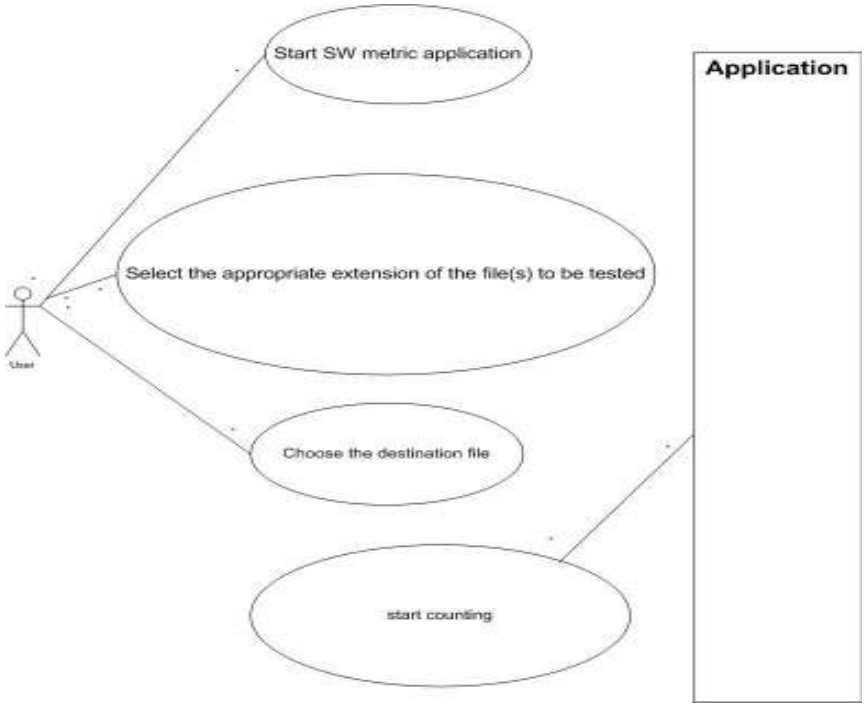



Figure 4.2. SWMetrics primary use case.

## 4.5. Modules, Their Purposes, Dependencies and Interfaces

### 4.5.1. The Main Application Module

This is the main class (SWMetricsForm class) in the Windows version.

#### 4.5.1.1. Purpose

The purpose of the main application module is to contain all the commands, user inputs and outputs that will assist in gathering the software metrics information.

#### 4.5.1.2. External Dependencies

This module depends externally on the platform to assist in bringing the directory and other details about the file(s) under test. It also depends indirectly on the file or files from which the metric information will be gathered.

4.5.1.3. Internal Dependencies

Following are the dependencies on other classes in the application.

4.5.1.3.1. Name: Saving data module

The main module depends on the saving data module to know where to save the gathered data.

4.5.1.3.2. Name: Code entry module

The main module inherits from this module. The code Entry class is an abstract class that is not going to be used directly through the application.

4.5.1.3.3. Name: Comments options module

The main module inherits from this module.

4.5.1.4. Public Interfaces

There are two public interfaces for the Windows version. Those are Browse and Count.

4.5.1.4.1. Name: Browse

4.5.1.4.1.1. Purpose: Called by the tester to select the directory or file from which the application gathers the metrics.

4.5.1.4.1.2. Brief overview: Press the “Browse” button on the main class.

4.5.1.4.1.3. Detailed overview: Select the target directory, and then select the correct file type(s).

4.5.1.4.1.4. Constraints: The file to be tested should be in a directory, or the main root. The directory has files with the right extension or type.

4.5.1.4.2. Name: Count

4.5.1.4.2.1. Purpose: Called by the tester to parse all methods of the tested file(s).

4.5.1.4.2.2. Brief overview: Press the “Count” button on the main class.

4.5.1.4.2.3. Detailed overview: This method calls all the methods of the tested file(s).

4.5.1.4.2.4. Constraints: The tester in the choosing file module should choose a valid file to be tested.

This is the main method in the application that calls several other methods to achieve the count. Each metrics from the listed metrics, lines of code, with comments and empty lines, LOC (without comments or empty lines), SLOC, Math operators, MC/DC, and Max nesting. It has a special function that goes through each file and counts its related metric. The metrics are gathered on the file and the function module. There is another function "functionmetric", that runs all those metrics on the file level. The application also has a parser within the methods to parse the code and define where each function starts and ends.

4.5.1.5. Overview of the operation

The Main module provides the following operations for managing the data.

4.5.1.5.1. Browse button click.

4.5.1.5.2. File type combobox_selectedindexchanged.

4.5.1.5.3. Count Button clicks.

4.5.1.5.4. Save dialogue, to select the file name and then "OK" or "Cancel" to execute or cancel the operation respectively.

All other modules are called through the Main module and have no direct access by the user.

4.5.2. The Main Application Module

This is the main class (SWMetricsForm class) in the Console version.

4.5.2.1. Purpose

The purpose of the main Console module is to contain all the commands, user inputs and outputs that assist in gathering the software metrics information.

4.5.2.2. External Dependencies

This module depends externally on the platform to assist in bringing the directory, and other details about the file(s) under test to the application. It also depends indirectly on the file(s) from which the metric information is gathered.

4.5.2.3. Internal Dependencies

4.5.2.3.1 Name: The parser module

The Main module depends on the parser module and calls the operation “Count” from this module. The Main module depends indirectly on other modules as they are called by the parser module.

4.5.2.4. Public interfaces

Main is the only public interface in the Console module.

4.5.2.4.1. Name: The Main

The Main operation is the public interface to the user or to other applications. It receives the directory name and the destination file name (optional) from the user or other applications and then calls the “Count” operation or interface from the parser module.

4.5.2.5. Overview of the operation

The Main module includes the following tasks for managing the data.

4.5.2.5.1. Running the application from the Console

4.5.2.5.2. Typing the directory name of the files in order to collect the metrics from them.

4.5.2.5.3. Typing the destination file name (optional)

All other modules are called through the Main module and have no direct access by the user. In the automated mode, the user, or the calling application, types one line and has the following options.

4.5.2.5.3.1. SW*.exe directory-name: With this line the tool runs and saves the data to a log file with the current date as name and extension (.csv). The file will in the same directory location.

4.5.2.5.3.2. SW*.exe directory-name filename: This gathers the metrics to the specified file name

4.5.3. The Parser Module (Console Version)

4.5.3.1. Purpose

The purpose of the parser Console module is to parse the directory or the modules of the selected directory, and then call the other classes, metrics, and comments to gather the required metrics.

4.5.3.2. External dependencies

This module has no external dependencies.

4.5.3.3. Internal dependencies

This module depends on the Main, metrics, and comments modules.

4.5.3.3.1. Name: The Main module

The parser module depends on the Main module to trigger it to start. It also receives, from the Main module, the directory and file information.

4.5.3.3.2. Name: The metrics module

The parser module depends on the metrics module. It calls the metric module every time it is calculating a specific metric.

4.5.3.3.3. Name: The comments option module

The parser module depends on the comments module to deal with empty and commented lines of the different types of comment lines.

4.5.3.4. Public interfaces

The parser module has one public interface, the Count operation.

4.5.3.4.1. Name: Count

4.5.3.4.2. Purpose: Count is called by the Main module to invoke all methods of the tested file(s).

4.5.3.4.3. Brief overview: Counted methods are called directly in the Main class or module.

4.5.3.4.4. Detailed overview: This method calls all the methods of the tested file(s).

4.5.3.4.5. Constraints: The counted modules should be of the types supported by C and C++ (.h, .cpp, .c).

This is the main method in the application that calls several other methods to achieve the count. Each metric from the counted metrics has a special function that goes through each file and counts its related metric. The metrics are run on the file level. There is another function, “functionmetric”, that runs all the metrics on the function level. The application also has a parser within the methods to parse the code and define where each function starts and ends.

4.5.3.5. Overview of the operation

The parser module provides the following operations for managing the data.

4.5.3.5.1. Parses the files and directory sequentially

4.5.3.5.2. Calls the metrics module to gather the metrics

4.5.3.5.3. Calls the comments options module whenever required

All other modules are called through this module and have no direct access by the user or the Main module.

4.5.4. The Metrics Module

4.5.4.1. Purpose

The purpose of the metrics module is to gather the required metrics: Lines of codes, statement lines of codes, maximum nesting, math count, MC/DC and cyclic complexity.

4.5.4.2. External dependencies

This module depends externally on the platform to assist in bringing the directory and other details about the file(s) under test to the application. It also depends indirectly on the file(s) from which the metric information is gathered.

4.5.4.3. Internal dependencies

4.5.4.3.1. Name: The parser module

The metrics module waits for the instruction from the parser module. The parser module decides the certain module or directory to be parsed.

4.5.4.4. Public interfaces

The following metrics operations are public interfaces used or called by the parser module.

4.5.4.4.1. Public bool **loopWords**, that specifies the specific loop words

4.5.4.4.2. Public bool **methodstart**, that is triggered when a method starts

4.5.4.4.3. Public bool **mcdcOperators,** that specifies MCDC operators on the function level

4.5.4.4.4. Public int **mcdc,** which returns the MCDC count on the function level

4.5.4.4.5. Public int **mathOperation:** returns the math count

4.5.4.4.6. Public int **CyComplexity**: returns the cyclic complexity count

4.5.4.4.7. Public bool **CcomplexityOperators**, that specifies the cyclic complexity operators

4.5.4.4.8. Public int **CountMaxNesting**: to count maximum nesting

4.5.4.4.9. Public bool **mcdcenabled**, that specifies MCDC operators on the file level

4.5.4.4.10. Public int **CountMCDC**: to return the MCDC count on the file level

4.5.4.4.11. Public int **CountLoc**, which returns LOC count

4.5.4.4.12. Public int **CountSLoc**: returns the SLOC count

4.5.4.4.13. Public int **CountSLocMath**: returns the math count on the file level

4.5.4.4.14. Public int **Countlines**: returns the lines count on the file level.

4.5.4.5. Constraints: Not Available (NA)

4.5.4.6. Overview of the operation

Same as in section 4.5.4.4.

4.5.5. The Comments option module (Console version)

4.5.5.1. Purpose

The purpose of the Comments option module is to specify the empty lines, the comment lines, and the lines that are not counted in the statement lines of code count.

4.5.5.2. External dependencies

There are no external dependencies.

4.5.5.3. Internal dependencies

This module depends on the parser and metrics modules.

4.5.5.3.1. Name: The parser module

The comments module is called by the parser module.

4.5.5.3.2. Name: The metrics module

The comments module is called by the metrics module.

4.5.5.4. Public interfaces

The following comments operations are public interfaces used or called by the other modules.

4.5.5.4.1. Public bool **locLinesMethod,** to specify LOC lines

4.5.5.4.2. Public bool **sLocLinesMethod**, that specifies SLOC lines

4.5.5.4.3. Public int **insideComments,** for inside comment type options

4.5.5.5. Constraints: NA

4.5.5.6. Overview of the operation

Same as in section 4.5.5.4.

**4.6. Algorithms**

4.6.1. The Parser General Description

The application has a built-in parser. The parser is responsible for defining where methods begin and end. The parsing algorithm is to look for the bracket "{", in the code to that triggers a start of a function. The process first excludes any bracket that exists within a comment line. The following are the main functional steps for the parser.

4.6.1.1. The tool parses the file(s) in the selected directory one by one. If there is a sub directory another loop runs to parse the file(s) in this sub folder(s) or directories. The application is designed to get down to four levels of sub folders. Increasing this number is just a matter of repeating the loop related to the inner directory. The decision to go down to four sub folders was heuristic by looking at many folders and how deeply they can usually go.

4.6.1.2. Each file is parsed line by line. The first step is to exclude the empty or comment lines. This includes the two types of standard comment lines: single "//" or multiple "/*","*/".

4.6.1.3. The tool then searches for the bracket and flag a start of a function upon finding the "{"symbol.

4.6.1.4. A counter is triggered to count the brackets. An opening bracket, "{"increments this counter and a closing one, "}" decrements it. The end of the function will be where the counter returns to zero.

The area between the starting "{", and the zero counter is the area from which the application will be gathering its all metrics. The name of the function is parsed from the string before the parenthesis, "(", that precedes the first bracket.

4.6.2. Function Metrics

The tool gathers metrics on the file and function levels. The function method is included within the parser class and not in the metric class, like the other metrics. The function metrics are nearly following the same algorithms of those in the file level. Those algorithms are explained below.

Before explaining the metric algorithms, it is important to mention that the definitions of the metrics are according to the client definitions that may not be all the time the standard definition for that metric.

4.6.3. MC/DC metric algorithm

Multiple Condition/Decision Coverage (MC/DC) is test coverage metric from the FAA DO-178B. This metric has to be satisfied for Honeywell most safety critical software.

One can calculate the number, by taking the number of conditions in a Boolean expression and adding one to it.

The MC/DC metric is triggered to count once the parser finds a word, or an operator of MC/DC. Those MC/DC operators are: If, else, switch and case.

The actual counting is triggered by the operators: and, &&, or, ||. The addition of one is per equation that satisfies the MC/DC condition.

There is an approach followed in parsing the two letters "||" or "&&" from a line. If we run a one round parsing letter by letter and check if two letters = "||" or "&&" there is a chance to miss those operators as we are counting one letter before or after. For example if we have a line that has "if (A or B)". Assume we start parsing from if, two letters each time, then we may get: if-A or B- strings, and then we will miss the "or" because it is divided between the two strings.

The approach followed is to run the line parse two times, one that starts from the first letter in the line and finishes one letter before the end, to avoid getting an empty buffer error, and another one starts from the second letter in the line and finishes in the last one. This way we will guarantee that the string will be caught by one of the two loops.

4.6.3.1. Example 1

If we have a function that has if (A or B) this should count three MC/DC conditions. The algorithm is 2+ 1 for "or" = 3.

4.6.3.2. Example 2

"A and B and C and D", the MC/DC metric == 5. We have 3 "ands" +2 =5.

4.6.4. Number of Math Operators Metric Algorithm

The mathematical operators that are going to trigger the math count are: “+,-,/,*,%,+=, -=,/=,*=,%=,--,>>,<<, and ++”. The assignment operator “=” is under two arguments to use it or not. It is includes in the math count in this tool.

A previous approach of parsing two characters is followed here, for those operators that are of two characters, i.e. +=, -=, etc... There was another issue that may cause an error, that is whether the operator will be counted once or twice, especially in the case of ++ for example, will it be counted as one ++ or two +. This is distinguished through an “if” statement that will not trigger the single + unless the line has no double ++ operators, and so on.

4.6.5. Maximum Nesting Count Algorithm

In the nesting level, the goal is to study the depth of the function or the file. The algorithm is formalized as a basic count, choosing the opening or the closing brackets, as they should be equal any time. The minimum nesting level for the function is expected to be one. If there were any brackets within the function in a comment line, they will not be counted.

4.6.6. Statement LOC Count Metric Algorithm

SLOC or Statement Lines Of Codes, is supposed to be a modified version of LOC where we do not only exclude the empty and comment lines, but also exclude the declaration lines, global variables and things of this manner that can usually be initiated or coded by the application and not the developer.

Here, we listed several words that if a line starts with, this line will be excluded. These words or symbols are : #, using, int, double, struct, signed, unsigned, char, const,

class, public, private, static, begin, end, bool, protected, void, break, return, try, boolean, else, typedef, inline, virtual, if, switch, for, while.

This is not an exhaustive list. It is difficult to obtain all the words that implement a global or declaration statement. Tuning the application for a better parsing should take into consideration to look for a better alternative for SLOC count. Those are the most popular words that will be repeated many times in the code.

Here a problem arises in that this is a Honeywell definition that could not be verified from another source. Nobody has a list of what SLOC words can be or has any other alternative algorithm.

4.6.7. Comments Algorithm

In C and C++, and maybe many other languages, the two types of comments are: "//" for a single line comment, "/*" for a start of multi line comment, and "*/" for the end of this multi line comments. Since the comments signs are always of two letters ( //, /*, */), I followed the same technique described in 4.6.3.

There are some challenges dealing with comments.

4.6.7.1. How to count the middle lines in the multi line comments where there is no sign for a start or an end of a comment line.

Here, a flag variable is raised whenever a start comment is true (“/*”). This flag, if it is true, will keep excluding lines even if there are no comment signs. A closing comment “*/” will disable this flag.

4.6.7.2. Dealing with comments that come within the line. For example you may have a line that starts with code, and then has comments at the end or the opposite.

This was very hard to deal with and is not solved completely in the tool. The main reason this is a serious problem to the code, is that it parses files and directories per line. The application can tell at the end of a line, whether this line is a comment line or not. To determine whether a line is partially commented requires the code to go down to the letter or character parsing level. Even if this is feasible to do by itself, yet it will require most other function to take it into consideration. I thought it is not with a high value that justify going that far. This should also be considered for any improvement of the application.

4.6.8. Cyclic complexity count metric algorithm

The operators that trigger cyclic complexity count are:

If, while, repeat, for, and, or, ||, &&, and case. Finding any of the above operators triggers the cyclic complexity count.

**4.7. Open Issues**

4.7.1. Database

The application is using txt, XML, or csv file types, to save the gathered metrics or data of the file(s) under test. Upon expanding the use of the application to include many tested classes, a database should be designed to hold all those gathered metrics.

#### 4.7.2. Performance

The tool parses several different files to get the different metrics. This may have an impact or overhead on performance.

#### 4.7.3. Accuracy

The accuracy of the parser and the gathered metrics is elaborated in testing. Formal methods can be used to proof the correctness of their algorithms.

### 4.8. Alternatives

Having many if statements, loops or conditions in an application may make it fragile and harder to test and understand. A more stable structure will be a better alternative. Many of the above algorithms were tuned more than once to be optimized and be more accurate. One thing I was hoping to accomplish is to take the code to the level of parse once and write once. Many preprocessing tasks were added to ensure the accuracy of the parsing process.

# CHAPTER 5. EVALUATION OF THE DEVELOPMENT PROCESS

## 5.1. Overview

In evaluating the product, we focus our attention on the quality of the delivered software as it is perceived by the customer. There are two main steps in software evaluation [10].

5.1.1. To collect information about the project's life cycle, measuring the satisfaction of the stakeholders involved, in order to determinate high and low points of the process.

5.1.2. To evaluate the course of the project as a whole. In relation to the first objective, forms should be submitted for the client, evaluating several aspects of the project.

The quantification of the result of this evaluation is used as a reference to trace high and low points presented during the project's life cycle. The collection of information that will be used for project analysis is accomplished through emails received from the client as feedback about the project prototypes. This was the only documented way of communication.

In relation to the second objective, we explain how the project was developed, personal relationships, adopted process, participation level, and problems found. As a result, the information collected will be the basis for the formation of a knowledge base that will potentially act as a continuous learning vector, leading to better results in the next projects. The use of brainstorming seems to be adequate in this aspect.

## 5.2. Introduction

The software development process has five phases: planning phase, specification phase, design phase, production phase, and post- production phase. Each phase has specified deliverables. Institutionally, these phases are coordinated and initiated through an

academic workflow process that ensures effective strategic goal setting, design, resource allocation, and implementation of all academic projects.

Nowadays there are a number of different approaches on how to develop high-quality software that would satisfy needs and expectations of its users. Some of these approaches differ significantly; however they all have a common feature: a quality assurance mechanism that enables to reach the quality of the developing software.

This evaluation is developed as part of the final document by the developer. Effective project evaluations in general should be performed by a supervisor, mentor, manager, or anyone who is in contact with all the different stake holders of the project.

### 5.3. Software Quality Evaluation Standards

There are some process assessments to evaluate the software quality: Some examples of these assessments are: ISO 9001:2000, Capability Maturity Model CMM, or TickIT.

The product assessment is evaluated through testing as well as other characteristics evaluations like: security, safety, usability, and other software external characteristics. Some standards for this evaluation are: ISO/IEC 9126: Software product quality, and ISO/IEC 14598 software product evaluation.

The (CMM) of the Software Engineering Institute (SEI) will be followed to evaluate the software process for this project.

### 5.4. Process Evaluation, CMM Model

Mature processes help ensure consistent quality of products, but they are not sufficient. Coupling process evaluation, with an assessment of the quality of the products that the process produces, can provide a more accurate analysis of an organization's capability than only evaluating the processes. Ignoring the quality of the products

developed by a process, can lead to an incomplete understanding of the risks that a development effort presents. Ignoring product quality can also result in the misallocation of resources in process improvement planning.

In the CMM evaluation mode, the development process is described through key process areas. These key process areas within the development function are organized into five levels on a one to five scale, where level one is considered as the worst organized level and level five the best organized one. This model provides a conceptual framework to classify the key process areas of the development function and it suggests an improvement path to reach a higher degree of development quality and productivity, while decreasing risks.

In this model, the numerous development activities are regrouped into key process areas belonging to key practice areas (there are 18) to all maturity levels. Successful implementation of all key process areas of a given level will indicate that the development function fully to qualify for a specific level. To qualify for the next level of process quality, we must then implement the key process areas required to fully implement the key process areas typical of the next level [11].

The CMM levels will be listed and applied to our project. We are using CMM model to evaluate the development process adopted and there was no adaptation of the CMM during the project. This may make many features of the model inapplicable. Those issues will be specified during the analysis.

### 5.4.1. The Initial Level

The initial level is a default level that any development process will follow. It has no formality, planning, and no standards [12].

5.4.2. Repeatable

In this level, we are supposed to follow a defined software development process and be committed to it. Following a known development process guarantees following a defined experienced approach toward software development. Due to the limited size of those who are working in the project, along with other resource limitations, we were to use a smaller version of the evolutionary or Scrum process. We will give a number between 0 and 10 showing how much our project conforms to the specific recommended task. 0 indicates no conformance at all, and 10 to show total conformance. These values represent only the developer viewpoint.

The following is a list of tasks that ought to be adopted in this level.

5.4.2.1. Requirements Management (7/10)

Establishing a process with the customer to ensure software meets expectations, and that requirements are controlled and managed as they change. This was the main goal of the project's initial document. Requirements and features were explained in a clear way that can be understandable by both sides to remove any ambiguity or misunderstanding. Upon parties' signatures and approval of this document, the project began. The continuous delivering of prototypes was to insure that the user can see the output in the format they will see toward the end and then verify it.

5.4.2.2. Software Project Planning (4/10): This means establishing attainable timelines and goals in the software development process. Although the main goals and features were established at the beginning of the project, no specific timeline or detail requirements were documented. The project is expected to have a fixed timeline which is the semester period. In reality, the time was flexible in that we started a month before the semester. We also had

the option to extend the time as it wasn't a critical factor to finish it within the specific semester time.

5.4.2.3. Software Project Tracking and Oversight (does not apply)

Here we establish management oversight so the project's progress can be monitored and adjusted when deviation is identified. This point is more of a team manager responsibility to keep tracking the different attributes of the project. In our case, there was no team involved in the management or development. Different parts of the projects were informed informally through emails about the project progress and status.

5.4.2.4. Software subcontract management (does not apply)

Means to establish effective criteria to select and manage software sub-contractors.

5.4.2.5. Software Quality Assurance (2/10)

Establish a validation and auditing processes to ensure that the delivered software is of high quality. There was no validation or auditing process available except through testing and testing oracles. The process of evaluating the quality of each deliverable prototype was informal and undocumented.

5.4.2.6. Software configuration management (6/10)

Establish a process that ensures the effective management and maintenance of a software product throughout its life.

Following the Scrum development process, we delivered periodic prototypes that implemented the project functionalities. Despite the fact that the project didn't use certain software for version management and control, yet an informal procedure was used to keep tracking the different deliverable prototypes. The naming convention used with those prototypes was to include the specific date of this version. All shared resources were

updated at the same time upon delivering a newer version. The new version was not officially delivered unless it passed all previous test cases that earlier versions passed.

5.4.3. Defined

Here the planning of the maintenance provides effort and cost trends based on the analysis of previous software version implementation. As indicated, most of level three issues do not apply to our project. These level issues apply for a well organized company with many individuals and teams that have different roles in this company. Here is the only one that may partially apply.

5.4.3.1. Peer Reviews (7/10)

Establish a process of peer review with the goals being to remove defects as early as possible in the development cycle and to foster organization wide communication and understanding. Peer reviews play an important role in this project. The three project members communicate for the verifications of any feature, removing, or modifying any of the requirements. They are also informed and updated with the delivery of any version. Each member performed a different type of review.

5.4.4. Managed

5.4.4.1. Quantitative Process Management (2/10)

A means to measure and control the results of the software process quantitatively to identify areas of weak performance. There was an informal process of identifying weak performance issues.

5.4.4.2. Software Quality Management (2/10)

A means of quantitatively measuring and achieving specified quality goals in the software products. Here also we followed an informal approach for the quality management.

5.4.5. Optimized

5.4.5.1. Defect Prevention (3/10)

It means to identify the causes of defects, and prevent future occurrences. This was a part of the debugging process. Errors were fixed without trying to make hypothesis that may guarantee getting rid of all similar errors.

Making hypotheses about errors to guarantee fixing all similar errors were only on the level that such errors may cause the application to crash or stop. This application is intended to run automatically and called by another application. Causing the application to crash may halt or stop the entire scripting procedure. It is very important to guarantee such errors do not occur.

5.4.5.2. Technology Change Management (TCM), (Does not apply)

TCM means Introducing new technologies into the organization in an orderly fashion.

5.4.5.3. Process Change Management (Does not apply)

This is a process to continually improve those key processes that affect quality, productivity and efficiency

Upon reviewing this brief process evaluation, it may not be easy to assess the overall quality of the system without studying the product quality itself. This evaluation will be more effective if the client contributes mostly to it.

## 5.5. Product Evaluation

This is defined as a systematic examination of the software capability to fulfill specified quality requirements [13]. Software quality evaluation, ISO/IEC 9126 quality model will be followed to evaluate our product quality. The attributes' evaluations, are approximates and informal that represent the experience of the author. Most of these attributes require formal tedious procedures to evaluate. There are also many dependencies and interaction between those characteristics. For example, Correctness may help reliability and efficiency, while hurt robustness, Efficiency may hurt correctness, reliability and robustness, and so on [14].

The above model has the following attributes to consider in the evaluation.

### 5.5.1. Internal Quality

This is the characteristics of software product during development. Some of these characteristics overlap with the external qualities, but all have different shades of meaning that are desired more in some cases and less in others. The attempt to maximize certain characteristics invariably conflicts with the attempt to maximize others.

The "ities" [14] that affect software's internal quality, which is the quality visible to the software's developers, include maintainability, flexibility, portability, reusability, readability, scalability, testability, and understandability. The same ranking used above will be used here (from zero to ten) to those attributes. It reflects the developer's informal opinion only.

5.5.1.1. Maintainability (6/10)

Defined as "the ease with which changes can be made to satisfy new requirements or to correct deficiencies" [15]. Well designed software should be flexible enough to accommodate future changes that will be needed as new requirements come to light.

Before going into the refactoring process, maintainability was very low. The project was complicated and fragile. The internal structure of the application now is easier to modify and change. The modules are divided by their functionalities. The code in the main module is minimized. The communication between modules is minimized. Despite refactoring, the number is relatively low due to the nature of the parsing process; there will always be huge traffic, dependability, and communication between the modules.

5.5.1.2. Flexibility (5/10)

The extent to which the project can be modified to deliver software other than that for which the project was originally intended or to respond to changes in project goals [14]. There are limited flexible features in this project such as having two versions: Windows and Console. Also, the fact that the project is designed to parse different types of codes others than C/C++ or the basic project types, contribute to its flexibility and the ability to be expanded for parsing other type of codes. But this project was customized and designed specifically for the Honeywell aviation division and complies with their standards and definitions.

5.5.1.3. Portability (6/10)

Defined as "the ease with which software can be used on computer configurations other than its current one" [15]. The final deliverable product is in the Console format. The choice of having it in the Console format was mainly to make it more portable and platform

in-dependable. In reality, .NET Console application still requires a specific environment or certain SDK's to be installed.

5.5.1.4. Reusability (5/10)

Defined as "the ease with which software can be reused in developing other software" [15]. This is very connected to flexibility. The algorithms and the parser were designed in a general format that makes them reusable for other types of application. However, there is no strong modularity that can support the ease of reusability. There are some dependencies among the modules that need to be minimized more in order to make those modules reusable.

5.5.1.5. Readability (4/10)

"Is the code readable, well-written, and well documented?" [16]. Maybe not! One feature we made to have the code readable was to have consistent meaningful classes, operations, and attributes names. The code still needs to have better comments in order to describe the operations and modules.

5.5.1.6. Scalability or Robustness (8/10)

"It is the ability of a system to continue to meet its response time, or throughput objectives, as the demand for the software functions increases" [17]. This is also related to the performance.

The test database that is used during testing is downloaded from different resources. It includes a significant amount of C/C++ code that is existed in several folders and sub folders. The total sizes of this code were above 10 MB. The total number of files in this suit was above 15,000 files or modules. The tests were performed on different computers with an average CPU speed and RAM size (Celeron: 1-2 GHz, 128-256 MB). The average speed

of parsing this entire test database, gathering all the required metrics, displaying them on the Console, and saving them to a file is in less than a minute.

Later in the project, the client requested to save the files temporarily to new files after deleting the comments lines, before performing the parsing process. This extra IO processes that require every single file of the directory to be read, cleaned, saved, then parsed, did not affect the overall good performance of the application. (See the testing chapter for details).

5.5.1.7. Testability (8/10)

This is "the ease of verifying that a software product satisfies its specifications and requirements" [17]. This is a project about some gathered metrics that can be formally and easily verified for correctness.

Once we are able to have formal definitions for those metrics, we can have a process to formalize and automate the results of the tests. This was already developed through the system as part of its test oracle.

The application also has no complex GUI that needs to be tested or validated by the user or by any other external tester.

5.5.2. External Quality

This is the executable software product. The "ities" that affect the software's external quality that is visible to the customer include usability, reliability, adoptability, and integrity, as well as correctness, accuracy, efficiency, and robustness [14]. Because many of these attributes are repeated in the internal qualities, and also as those attributes are expected to be evaluated by the client or user, there will be no addition for details by the developer in this section.

5.5.3. Quality in Use

This is the user's view of the software product when it is used in a specific environment and context. The same rules from section 5.5.2. are applied here from the user view.

5.5.4. Quality Measurements

Those are the measurements of internal, external and quality in use attributes, e.g., reliability. This part should be filled by the users of the application.

**5.6. Documentation**

There were many deliverable documents for this project. This is a list of those documents and their approximate delivery time.

5.6.1. Project Initiation Document

This document was delivered in early January 2006.

5.6.2. Requirement Document

Requirement document was delivered in February 2006.

5.6.3. Design Document

Architectural and detailed design documents were delivered in March 2006

5.6.4. Testing Document

It was delivered in April 2006.

5.6.5. Final Project paper or document

It will be delivered by May 2006.

5.6.6. Emails

Throughout the development process, several emails were sent back and forth between the developer and the client [9]. Those emails include some requirements, requirement description, feedback, or error description, etc.

**5.7. Lessons Learned**

5.7.1. It is necessary to standardize or formalize the sending and reception process of functional and non-functional requirements between the client and the development team.

5.7.2. It is necessary to arrange for greater planning and communication.

5.7.3. Project planning and timeline should be documented. This time should always be verified to make sure we are still within the allocated time.

# CHAPTER 6. EVALUATION OF THE APPLICATION DESIGN

## 6.1. Abstract

Design evaluation is quite simply deciding how well a design meets the total set of requirements. It is also the quality engineering task during which the design work products and the performance are evaluated [18]. This is primarily a matter of understanding the quantifiable design performance to cost impacts in relation to quantified requirements. This evaluation needs to go through several stages, but remains fundamentally the same question: *what does the design contribute to meeting our objectives?*

The design evaluation stages can be listed as: proper definition of performance and cost requirements (entry condition), constraint based elimination, performance based selection, resource based optimization, and risk based weeding out [19]. In design verification, we need to verify that the design does not have any characteristics that will cause it to fail under operational scenarios.

Understanding the design process itself does not necessarily lead to understanding how the design should be evaluated. One reason is that not all design processes make explicit what design values the process emphasize. Another reason is that a design process may claim to lead to some values, when this may not always be the case. However, where the values of a design process are explicit, and where it is clear that the process does lead to those values, this could indeed facilitate design evaluation.

## 6.2. Evaluation Principles

### 6.2.1. Specific Requirements

"A design can only be evaluated with respect to specific requirements" [19].

We cannot easily expect anyone to perform a logical evaluation of the suitability of a given design for a fuzzy requirement. In order to evaluate a design, we should have well defined requirements. In our project, we have part of the requirements that can be defined well and formally. Those are the gathered metrics themselves. We can apply formal methods to make those requirements formally defined. This will make it easy for us to verify them through design. However, this is not the case for all the requirements that we have. We actually have some of those requirements changed toward the end of the project development. There are many prerequisites and preconditions for a very good design evaluation that we may not be able to achieve here. Using formal methods in requirements will also help define those requirements quantitatively.

6.2.2. Constraints

"It doesn't matter how good or how cheap a design is, if constraints forbid it" [19].

This simply states that the design should do what the requirements demand, not more or less. We assume that there is a flow of one or more design ideas to be evaluated. The question of how we determine these ideas is a separate topic. Before we go deeper into the design question, we need to assess if it is disqualified by any requirement. Every design feature has to be linked to a requirement or else removed.

At earlier stages, we may need to set a side those design pieces that violate a constraint or a requirement. Later, we may find out that it is indirectly required.

6.2.3. Alternative

" Designs should not be rejected permanently, the reasons for rejection should be clearly documented, the design specification kept; and the rejection possibly reevaluated later" [19].

In some cases, there can be some design alternatives to be evaluated later. This secondary design may come forward and be becomes the primary one.

6.2.4. Resources

” Designs must also be evaluated with respect to the design costs’ relative to our finite resources” [19].

The design should consider what resources we have and take this into account. The best design is what can be implemented within the time limits we have, and by the developers or other personnel available for the project.

6.2.5. A Continuous Process

” The evaluation of a design is a continuous process over a series of estimation and validation events” [19]. A lot of questions need asking by many people and we need many good answers to evaluate a design.

6.2.6. Incremental

” The best practical evaluation of design risks is by practical small steps” [19].

**6.3. Objectives and Desired Characteristics**

The purpose of Software design is to provide a design for the software that implements the requirements, and can be verified against it.

The typical objectives of the design evaluation task are to,

6.3.1. Determine if the designs are having the following characteristics, using the same theoretical zero to 10 scale, better evaluation numbers will be if it was approved by the customer,

6.3.1.1. Correct (7/10)

Is the proposed solution correct? Does it properly define the goals? Are there any errors?

We have to verify that the algorithms we used are correct. Verification can have formal or informal approach. Testing is a typical informal approach for algorithm verifications. All the algorithms built within the system, including the Parser, were verified by the developer and the customer. Formal verification for correctness of such algorithms was above the scope of the project.

A Successful project is [21] completed and operational, while it is on Schedule, within forecasted cost, and with all originally specified features and functions. A challenged project is: completed and operational, but behind schedule, over cost, and with fewer features and functions than originally specified. A failed project is cancelled before completion or never implemented. According to this classification and since our project is completed within semester time and schedule, it is a successful project.

6.3.1.2. Complete (7/10)

Complete means we have captured all requirements the customer needs. It also means that anything in the design has an original requirement piece and satisfies that requirement. This makes our design exclusively complete.

The design completely specifies the committed matrix of requirements; however, it does not include most of the optional one. The suggested original requirements are subjective, and the client expressed their willingness to accept any constructive modifications or recommendations. The final output was not feasible to both sides and there was an acceptable percent of requirement modifications.

6.3.1.3. Consistent (6/10)

Consistent means the requirements are not contradictory. The system can be implemented to meet all requirements. Consistent may also mean following the same style in defining, designing, coding, and testing of the project. Consistency makes it easy to understand, and reuse. The fact that most of the project has been accomplished by one person makes the consistency issue much easier.

Consistency and traceability are established between software requirements and software design. Consistency is supported by establishing and maintaining traceability between software requirements and the software design when needed.

6.3.1.4. Feasible and adoptable (8/10)

Design feasibility can be verified in advance by simulation or modeling. We followed the Scrum agile development approach because feasibility in the beginning was not high. By developing continuous prototypes and getting the client feedback, we are able to verify and build the correct design. In reality the design was tuned and evolved in order for the prototypes to reach its verified stage.

6.3.1.5. Testable (6/10)

The fact that the application has a very basic user interface, the Console, makes it more testable. Testability goes by different definitions. It can be defined as visibility and control. Visibility is our ability to observe the states, outputs, resource usage and other side effects of the software under test. Control is our ability to apply inputs to the software under test. The fact that the tester is the developer for this project is an advantage that I had in mind testing while developing. Testability feature in general, requires a good cooperation between the testing and developing teams.

6.3.1.6. Maintainable (6/10)

Software maintainability is defined as the ease of finding and correcting errors in the software. It is analogous to the hardware quality of Mean Time To Repair, or MTTR. While there is as yet no way to directly measure or predict software maintainability, there is a significant body of knowledge about software attributes that makes software easier to maintain. These include modularity, self (internal) documentation, code readability, and structured coding techniques [4].

As explained earlier, the project went into a refactoring process to improve its maintainability. The project is partially modular, and requires better code comment and documentation. These two deficiencies occurred as a result of trying to meet the expected goals of time and availability.

6.3.1.7. Partitioning (5/10)

A major purpose of the partitioning is to arrange the elements in the software subsystems so there is a minimum of communications needed among them. Partitioning is something that should be done during the design stage. Delaying it to a later stage may cause an extra extensive work. This was a problem that occurred for our project due to the gradual learning of the detailed requirements, and following the Scrum development process.

6.3.1.8. Incremental (9/10)

This means delivering system increments periodically. This is the approach adopted throughout the entire development process. In many projects, the absence of incremental development proves to be a main reason for failure.

Incremental is a scheduling and staging strategy in which pieces of the system are developed at different rates or times, and integrated as they are developed. It addresses errors and new learning in the overall methodology. After a section of the system is built, the methodology is examined to find where it was wrong or could be improved. The structure, the techniques, or the deliverables might be changed [21].

6.3.1.9. Scalable (7/10)

The application has no scalable problems. It has been tested under large amount of code and this didn't affect the overall performance of the process.

6.3.2. Determine if the deliverable design work products are correct, complete, consistent (internally, externally with other work products, and externally with related conventions), and understandable.

6.3.3. Identify defects in the deliverable design work products so that the defects can be fixed, and defect trend analysis can be performed to improve the process and staff training.

6.2.4. Help ensure that the design tasks are completed, effective, and efficient.

**6.4. Design Evaluation Techniques or Mechanisms**

The design evaluation task typically can be performed using the following techniques,

6.4.1. Demonstration

A demonstration is the technique of modeling the execution of a partially complete application or prototype, in order to obtain feedback from the customer [18]. As explained earlier, this is a technique heavily used in our project. Many partially and gradually completed prototypes were delivered during the life time of the project.

The reason this is a very effective technique is because people can often understand an executable work product easier by seeing it demonstrated than by sitting through a presentation or reading documentation.

The typical objectives of a demonstration are to inform the customer of the current state and capabilities of the application or prototype, and then to obtain feedback from the customer (e.g., new requirements and defects identified).

This is actually part of the Scrum agile development process; that is to develop deliverables to the customer in every cycle. This will lower the risk of the project to fail. Another advantage of using prototypes is that we actually make the customer part of the testing team and continuously receive his or her feedback.

Selecting the person who is interested in the product, the person who is in contact with the original problem, is going to be the best one to evaluate the prototype. In general, the risk of using prototypes is to ignore the fact that this is just a prototype and not a final product.

It is important to make sure that the delivered prototype is executable and free from errors. Another advantage of having prototypes earlier is that those prototypes will be tested in the client's environment. This is the environment on which the final product will be working. It is very important to know early if there are any major obstacles that prevent our application from working properly in the customer environment.

6.4.2. Inspection and Inspection Reports

Inspection is a quality engineering technique during which a work product is formally evaluated against a checklist, in order to identify defects.

There was no direct inspection from the customer on the design products. Customer feedback is inspected to verify those products. This inspection is accomplished manually and informally.

An inspection report is the work product that is produced by an individual inspector during an inspection in order to document the defects and provide any feedback. The customer provided some inspection reports informally through emails.

6.4.3. Design Walkthroughs

This is another design verification task that is done informally. The customer has the chance to look at the design diagrams and verify or comment on them. The design is also verified against the use cases and diagrams.

6.4.4. Modification Tasks

This is where the design must accommodate reasonable changes. Other informal design verifications are as follows: specification checks to see if the design does match analysis and design critics (by developer, supervisor, or customer).

6.4.5. Software Metrics

The goals of software design metrics are to satisfy the following issues [22]:

6.4.5.1. The possibilities of predicting the degree of errors in the software being designed.

6.4.5.2. What is the degree of maintainability that would be needed for particular software?

6.4.5.3. What is the degree of difficulty in testing that would be required; also what is the level of testing needed after which it would be possible to decide if the software module is still useful?

6.4.5.4. The amount of effort needed based on some quantifiable features like number of errors in a module

6.4.5.5. The amount of effort needed for testing a software module based on the design features

6.4.5.6. Does the design predict the final size of the software based on the design?

6.4.5.7. Using design metrics, can we identify problems in the design on the basis of the identification of certain outliers?

Our project is a software code metric tool. Metric applications are used to evaluate the design and the implementation of projects. We can use our own application to evaluate it. However, the metrics themselves are the first step of the evaluation. As explained in the problem definition, the evaluation metrics are usually of higher level concepts than the actual gathered metrics. Drawing the connection between both is the subject of the research paper that is expected to follow. For example, Mean Time Between Failures (MTBF) or maintainability, can be an evaluation metric. We can not gather MTBF directly from the code. There are some formal approaches that define, for example: MTBF, or maintainability, in terms of MC/DC, maximum nesting, lines of codes, cyclic complexity, and/or some more metrics that are not yet gathered by our tool.

There are some attempts to define external and internal design metrics, to identify error-prone modules in software design [29]. There are also some researchers that draw a direct link for some of our collected software code metrics with error prone modules (those modules that require more testing), like cyclic complexity. This is why cyclic complexity was suggested although it was not listed in the committed list.

It is not recommended to have many public operators in classes or modules. The correct object oriented practice is to minimize the amount of class exposed information. The application, however, has a relatively large number of public operations due to the

high amount of information or messages that are transmitted or passed between the classes. This is another issue that could have been designed better in this project. There is an available Software metric design tool available to download [31] that I was hoping to use to evaluate my design. This tool has a large amount of design metrics that can be gathered and evaluated. Some of those design metrics are listed in Table 6.1.

Table 6.1. Some software design metrics [31].

| **Type** | **Metric** | **Category** | **Description** |
|---|---|---|---|
| Class Metrics | NumAttr | Size | The number of attributes in the class. |
| Class Metrics | NumOps | Size | The number of operations in a class. |
| Class Metrics | NumPubOps | Size | The number of public operations in a class. |
| Interface Metrics | NumOps | Size | The number of operations in the interface. |
| Interface Metrics | EC_Attr | Coupling (export) | The number of times the interface is used as attribute type. |
| Interface Metrics | EC_Par | Coupling (export) | The number of times the interface is used as parameter type. |

Due to the time and resource limitations, and the necessity of achieving goals on schedule especially with following an agile development approach, the main actual design evaluation was only through demonstration or prototypes.

**6.5. Design Variability**

For designs which are sufficient and complete, the next phase of an evaluation is the capability of the design to adapt to changes in the requirements and environment.

The design is present for a discussion of "what-if" scenarios. These scenarios cover the dimensions of design variation such as [27],

6.5.1. Changes to the domain model

Those changes indicate how the design would change if the analysis model were extended to include ...). There are some extensions that our project can accommodate. For example, the application can parse different type of code, other than C/C++ original goal. The parser and algorithms can be applied in general to those codes. Other extensions like collecting a new metric are not implemented. This new metric can be included within the metric class, and then it will be called by other modules.

The design of the parser has the limitation of going to four sub directories. This is just an arbitrary number that can be easily modified within the implementation.

6.5.2. Changes to the computing environment

How would the design change if the system were distributed across multiple processors? The application may not be flexible with changing the hardware or the platform.

### 6.5.3. Changes to the infrastructure

How would the design change if the client wanted to choose a different database?. The application receives the module or function data through a tree of files and folders. There is no practical alternative (like a database) for this approach. The project is specifically designed to parse from a tree.

### 6.5.4. Changes to enable reuse

How would the design change if we wanted to make the processing engine available for reuse by other projects? The processing engine we have is the parser. The parser receives a file or directory of files, cleans them from the comments, and then calls the metrics from the metrics class. It has the main functionalities that can be reused for other projects after some modifications.

The time limitation of the project reduces the ability to design for a more flexible design that can accommodate different type of scenarios. The object of this elaboration is not to change the design to accommodate these possibilities, but rather to explicitly expose the variation points (or lack thereof) within the design.

If the design stage can be done again, the above design variability consideration will be taken into consideration. This makes the tool suitable for other businesses or scenarios. This also makes it more capable of being a commercially used product.

# CHAPTER 7. TESTING

## 7.1. Introduction

Software testing is the process of executing a software system to determine whether it matches its specification. Testing is universally acknowledged as an important part of the development process. How should testing be done?! Are formal proofs necessary? How much testing is enough? How important is regression testing? At software engineering's current stage, most of these questions have no single right answer; there are many answers, each with some value.

We should do testing because [34]

7.1.1. It never works completely the first time it is used.

7.1.2. It is sensitive to minor errors, as there is no meaning to "almost right".

7.1.3. It is difficult to test because interpolation is not valid.

7.1.4. There are some "sleeper bugs".

Software testing is trying to answer two questions,

A. Are we building the right system (validation)? This is testing the requirement specification.

B. Are we building the system correctly (Verification)? This is about testing the implementation against the requirements. This part will be the main focus for this chapter.

## 7.2. Test strategies

The proper testing of software requires a lot of work, and therefore a lot of time. In an environment where you have limited time and resources, perhaps the best strategy to follow is what is called "risk and requirements based testing" [35]. In this strategy, we assume that testing everything is not feasible. This strategy helps us to determine what to

test first, and in which sequence. This is to insure that we will spend the time we have for testing on the important parts. The first step is to analyze requirements to determine the most sensitive ones. There is a list of such risk or important functionalities candidates in [35] that we will apply to our project. Since there are several items in the list, the most relevant only, will be listed in every one.

7.2.1. Functions Often Used by the Users

The most important functionality in our project is the parsing task. The parsing process and its all sub-parts or functionalities is important to the application as well as to the customer. To the customer, everything depends on how accurate the parser is. If the parser parses more or less functions and modules, everything else is affected by this accuracy. It is sequentially the first step in the whole process. The parser then should be given the highest priority in testing.

7.2.2. Complex Functions

This also applied to the parser. It has many attributes and algorithms.

7.2.3. Functions that Have a Lot of Updates or Bug-Fixes

To list the system main functionalities, will be again the parser algorithms and functionalities.

7.2.4. Functions that Require High Availability

The Input Output, IO functionalities within the parser require a lot of processing and availability.

7.2.5. Functions that Require a Consistent Level of Performance

This includes all the metrics and parser algorithms.

7.2.6. Functions that are Developed with New Tools

Our whole tool is new, so this has no specific candidate.

7.2.7. Functions that Require Interfacing with External Systems

The operation “Count” is the only operation that is interfacing the main function.

The other listed tasks will give similar results. This strategy helps us focus a major part of the testing on the parser itself. Most of the unit or the white box testing is done during the development cycles. This testing part focuses on black box testing.

As the Console version is the actual final deliverable product, testing will be on this version only.

**7.3. Test Cases**

A test case is a specific set of steps. It has an expected result, along with various additional pieces of information. Each test case should be simple enough to clearly succeed or fail, with little or no gray area in between. Ideally, the steps of a test case are a simple sequence: set up the test situation, exercise the system with specific test inputs, verify the correctness of the system outputs.

An excellent test case satisfies the following criteria: reasonable probability of catching an error, exercises an area of interest, does interesting things, and does not do unnecessary things, neither too simple nor too complex, and not redundant with other tests. Test cases will take these points into consideration. The Test case template below [9] will be adopted.

Figure 7.1 (test case number1) is a basic parsing process testing. The focus on this test is on verifying the parsing of the specific functions with their names, no more or less. A visual check is still needed to verify whether those are the actual functions and names in

the file without adding or subtracting from them. A pass or fail result will be written in the expected result column to express the final result of the test.

| Test Case Number 1: Basic Parsing testing | | Target: Parser Class |
|---|---|---|
| Objectives | Validate the main objectives of the parser. | |
| Preconditions | Input | Expected Results |
| A simple C/C++ file in a directory. Only one file in the directory | SWCMetric.exe c:\testDirectory | Parse list of the name of the files within the directory.<br><br>**Pass** |

| | function | File Name |
|---|---|---|
| 2 | | |
| 3 | static int load_fontDB | 4-12-2006\bestfont.c |
| 4 | static int match_aux | 4-12-2006\bestfont.c |
| 5 | static int match | 4-12-2006\bestfont.c |
| 6 | GrTextOption *GrFindBestFo | 4-12-2006\bestfont.c |

Figure 7.1. Test case #1.

There is a serializing process to remove the comments. This process removes the comments from the files and saves them in a temporary folder before running the metrics. As a result, the full name of the file is not correct. Regenerating the whole tree structure may affect the overall performance if we have a big directory with many sub directories.

Figure 7.2 is the file that is parsed. Other files will not be listed in this document due to the large size of those files. The parsed file is,

```
#include "grx.h"
#include "libgrx.h"
#include "grxfile.h"
#include "gmalloc.h"
#include <string.h>
#include <stdio.h>
static GrFont *fontDB;
static int DBsize;
static int loaded = FALSE;
static int load_fontDB(void)
{
FntDirHdr hdr;
char fullname(200);
int  file,size;
_GrGetFname("fonts.dir",_GrFontPath,FNTENV,"",fullname);
if((file = _GrFileOpen(fullname)) == EOF) return(FALSE);
if((read(file,&hdr,sizeof(FntDirHdr)) != sizeof(FntDirHdr)) ||
(hdr.magic != FONTDIR_MAGIC) ||
((size = sizeof(GrFont) * (int)hdr.numentries) <= 0) ||
((fontDB = _GrMalloc(size)) == NULL) ||
(read(file,fontDB,size) != size)) {
if(fontDB != NULL) _GrFree(fontDB);
_GrFileClose(file);
return(FALSE);
}
_GrFileClose(file);
DBsize = (int)hdr.numentries;
return(loaded = TRUE);
}
static int match_aux(char *pattern,char *string)
{
for( ; ; ) {
switch(*pattern) {
case '\0':
case ':':
return((*string == '\0') ? TRUE : FALSE);
case '?':
if(*string == '\0') return(FALSE);
pattern++;
string++;
```

Figure 7.2. Sample parsed file.

```
break;
case '*':
for( ; ; ) {
switch(*++pattern) {
case '\0':
case ':':
return(TRUE);
case '?':
if(*string == '\0') return(FALSE);
string++;
continue;
case '*':
continue;
default:
break;
}
break;
}
for( ; ; ) {
string = strchr(string,*pattern);
if(string == NULL) return(FALSE);
if(match_aux(pattern,string)) return(TRUE);
string++;
}
default:
if(*pattern != *string) return(FALSE);
string++;
pattern++;
}}}
static int match(char *pattern,char *string)
{
while(*pattern != '\0') {
if(match_aux(pattern,string)) return(TRUE);
pattern = strchr(pattern,':');
if(pattern == NULL) return(FALSE);
pattern++;
}
return(FALSE);
}
GrTextOption *GrFindBestFont(int width, int height, int magnify, char *family,
GrTextOption *where){
GrFont *f;
int error = TRUE;
int found = (-1);
```

Figure 7.2. Sample parsed file.

```
int ii,minerr;
if(!loaded && !load_fontDB()) return(NULL);
for(f = fontDB,ii = 0; ii < DBsize; f++,ii++) {
if((f->fnt_height <= height) &&
(!f->fnt_isfixed || (f->fnt_width <= width)) &&
(match(family,f->fnt_family))) {
f->FNT_USABLE = TRUE;
error = FALSE;
if((f->fnt_height == height) &&
(!f->fnt_isfixed || (f->fnt_width == width))) {
magnify = FALSE;
found   = ii;
break;
}}
else f->FNT_USABLE = FALSE;
}
if(error) return(NULL);
if(found < 0) {
minerr = 32000;
for(f = fontDB,ii = 0; ii < DBsize; f++,ii++) {
if(!f->FNT_USABLE) continue;
if(!magnify) {
error = height - f->fnt_height;
if(f->fnt_isfixed) error += width - f->fnt_width;
}
else {
int magn = height / f->fnt_height;
int size = magn * f->fnt_height;
error = (height - size) + (2 * --magn);
if(f->fnt_isfixed) {
magn = width / f->fnt_width;
size = magn * f->fnt_width;
error += (width - size) + (2 * --magn);
}}
if(error < minerr) {
found  = ii;
minerr = error;
}}}
f = GrLoadFont(fontDB(found).fnt_name);
if(f == NULL) return(NULL);
if(where == NULL) {
where = _GrMalloc(sizeof(GrTextOption));
if(where == NULL) return(NULL);
}
memset(where,0,sizeof(GrTextOption));
```

Figure 7.2. Sample parsed file.

```
where->txo_font = f;
if(!magnify)
{
where->txo_xmag = 1;
where->txo_ymag = 1;
}
else
 {
where->txo_ymag = height / f->fnt_height;
where->txo_xmag = f->fnt_isfixed ?
width / f->fnt_width :
where->txo_ymag;
}
return(where);
}
```

Figure 7.2. Sample parsed file.

We have four methods; the parser successfully parsed them with the right method names, with no extra or false methods.

Test Case Number two (Figure 7.3) demonstrates parsing to a specific file name. There is a third optional argument. The user can specify the temporary directory name and location. If it is left empty, the application will save them to a temporary directory that will be deleted at the end of the parsing process. This directory has today's date name and is in the same directory where the application is located.

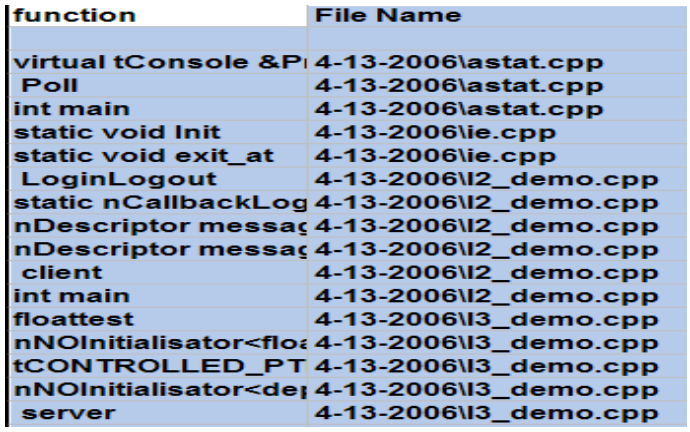
Test Case Number three (Figure 7.4) shows testing the parsing of a directory of files. In some cases, I have to do recent fixes. Prior to adding a temporary step to remove the comments lines, there were chances that the parser will pick something from the comment lines. Most of the comments lines can be eliminated, but in some cases like the inside comment lines, they will not be preprocessed as they have useful information. The idea of getting rid of the comments lines improves the overall accuracy of the parser.

| Test Case Number 2: Parsing to a specific file name | | Target: Parser Class |
|---|---|---|
| Objectives | The user chooses to type a specific destination for the gathered results. | |
| Preconditions | Input | Expected Results |
| C/C++ file/directory existed. | SWCMetric.exe c:\testDirectory c:\testResults.csv | Parse the results to the specified file. **Pass** |

1. The user can also specify the destination file type.
2. The gathered metrics are saved to the specified file.
3. This argument is optional. If it is left empty, the application selects today's date as a file name with extension ".csv".

Figure 7.3. Test case #2.

| Test Case Number 3: Testing the parsing of a directory of files | | Target: Parser Class |
|---|---|---|
| Objectives | Try to validate the main objectives of the parser. | |
| Preconditions | Input | Expected Results |
| C/C++ directory existed. | SWCMetric.exe \\testDirectory<br>Sample Output | Parse list of the name of the files within the directory.<br>**Pass** |



| function | File Name |
|---|---|
| | |
| virtual tConsole &P | 4-13-2006\astat.cpp |
| Poll | 4-13-2006\astat.cpp |
| int main | 4-13-2006\astat.cpp |
| static void Init | 4-13-2006\ie.cpp |
| static void exit_at | 4-13-2006\ie.cpp |
| LoginLogout | 4-13-2006\I2_demo.cpp |
| static nCallbackLog | 4-13-2006\I2_demo.cpp |
| nDescriptor messag | 4-13-2006\I2_demo.cpp |
| nDescriptor messag | 4-13-2006\I2_demo.cpp |
| client | 4-13-2006\I2_demo.cpp |
| int main | 4-13-2006\I2_demo.cpp |
| floattest | 4-13-2006\I3_demo.cpp |
| nNOInitialisator<flo | 4-13-2006\I3_demo.cpp |
| tCONTROLLED_PT | 4-13-2006\I3_demo.cpp |
| nNOInitialisator<de | 4-13-2006\I3_demo.cpp |
| server | 4-13-2006\I3_demo.cpp |

Figure 7.4. Test case #3.

Those are the primary test cases for the parser. Several other test cases were applied to test the performance and the way it deals with high volume of code. Here is a quick list of some test cases for error prone project parts,

7.3.1. IO problems

Many tests were done to verify no input/output errors. As mentioned before, the application has a lot of IO operations and this may make it faulty. Some tests were done using the same file name from earlier test cases. There are temporary folder and directory created that need to be tested. Most of these tests actually triggered some fixes to be done.

7.3.2. No directory

If the user tried to enter an illegal directory that the application will parse from or left the option empty, the application will not run and will prompt entering the directory name again.

Test Case 4 (Figure 7.5) is testing Lines of Code count. Lines of Code count is the first metric. It is an optional metric, but most others depend on it. We can use it to verify the correctness of the later ones. The verification of these results has to be done manually in counting such lines for each function and see if it matches those numbers. In some cases, for very specific code files, I verify them manually and then save their result values to be compared for later results as a test oracle for regression testing as will be explained later.

| Test Case Number 4: Testing Lines of Code count | | Target: Metric Class |
|---|---|---|
| Objectives | Try to validate Lines of code parsing. | |
| Preconditions | Input | Expected Results |
| C/C++ directory existed. | SWCMetric.exe<br>\\testDirectory<br>Sample Output | Parse list of the name of the files within the directory.<br>**Pass** |

| function | File Name | LCount |
|---|---|---|
| | | |
| virtual tConsole &F | 4-13-2006\astat.cpp | 3 |
| Poll | 4-13-2006\astat.cpp | 27 |
| int main | 4-13-2006\astat.cpp | 12 |
| static void Init | 4-13-2006\ie.cpp | 16 |
| static void exit_at | 4-13-2006\ie.cpp | 6 |
| LoginLogout | 4-13-2006\I2_demo.cpp | 27 |
| static nCallbackLo | 4-13-2006\I2_demo.cpp | 16 |
| nDescriptor messa | 4-13-2006\I2_demo.cpp | 9 |
| nDescriptor messa | 4-13-2006\I2_demo.cpp | 20 |
| client | 4-13-2006\I2_demo.cpp | 36 |
| int main | 4-13-2006\I2_demo.cpp | 10 |
| floattest | 4-13-2006\I3_demo.cpp | 60 |
| nNOInitialisator<flc | 4-13-2006\I3_demo.cpp | 5 |
| tCONTROLLED_P | 4-13-2006\I3_demo.cpp | 37 |
| nNOInitialisator<de | 4-13-2006\I3_demo.cpp | 5 |
| server | 4-13-2006\I3_demo.cpp | 18 |
| client | 4-13-2006\I3_demo.cpp | 61 |
| int main | 4-13-2006\I3_demo.cpp | 8 |
| int main | 4-13-2006\master.cpp | 49 |

Figure 7.5. Test case #4.

Test case 5 (Figure 7.6) is to validate LOC and SLOC. LOC and SLOC were defined previously. LOC excludes the comments and empty lines; SLOC excludes some other lines such as macros, global variables, headers, etc.

| Test Case Number 5: Testing LOC and SLOC | | Target: Metric Class |
|---|---|---|
| Objectives | Validate LOC and SLOC. | |
| Preconditions | Input | Expected Results |
| C/C++ directory existed. | SWCMetric.exe<br>\\testDirectory<br>Sample output | Parse list of the names of the files within the directory.<br>**Pass** |

| function | File Name | LCount | LOC line | SLOC |
|---|---|---|---|---|
| | | | | |
| virtual tConsole | 4-13-2006\astat.cpp | 3 | 3 | 2 |
| Poll | 4-13-2006\astat.cpp | 27 | 27 | 24 |
| int main | 4-13-2006\astat.cpp | 12 | 12 | 11 |
| static void Init | 4-13-2006\ie.cpp | 16 | 16 | 12 |
| static void exit_ | 4-13-2006\ie.cpp | 6 | 6 | 4 |
| LoginLogout | 4-13-2006\l2_demo.cp | 27 | 27 | 19 |
| static nCallback | 4-13-2006\l2_demo.cp | 16 | 16 | 12 |
| nDescriptor mes | 4-13-2006\l2_demo.cp | 9 | 9 | 7 |
| nDescriptor mes | 4-13-2006\l2_demo.cp | 20 | 20 | 17 |
| client | 4-13-2006\l2_demo.cp | 36 | 36 | 30 |
| int main | 4-13-2006\l2_demo.cp | 10 | 10 | 7 |
| floattest | 4-13-2006\l3_demo.cp | 60 | 60 | 52 |
| nNOInitialisator | 4-13-2006\l3_demo.cp | 5 | 5 | 4 |
| tCONTROLLED | 4-13-2006\l3_demo.cp | 37 | 37 | 29 |
| nNOInitialisator | 4-13-2006\l3_demo.cp | 5 | 5 | 4 |
| server | 4-13-2006\l3_demo.cp | 18 | 18 | 16 |
| client | 4-13-2006\l3_demo.cp | 61 | 61 | 55 |
| int main | 4-13-2006\l3_demo.cp | 8 | 8 | 5 |
| int main | 4-13-2006\master.cpp | 49 | 49 | 42 |
| main | 4-13-2006\md5.cpp | 32 | 32 | 24 |

Figure 7.6. Test case #5.

In an early stage of the project evolution, the comments were excluded without removing them. This would show different values in the first two columns. Late in the implementation, we decided to get rid of all comment lines to increase the accuracy. Having the exact same values in the first two lines for all methods indicates the success of

this intermediate preprocess. The two columns are kept for verification. The third column should always be equal or less to LOC. As the data is saved to Excel, a conditional formatting is made to alert whenever the value in SLOC is larger than LOC. Testing the values beyond this is a subjective manner that needs to be checked manually. Test Case number 6 (Figure 7.7) is to validate the rest of the metrics.

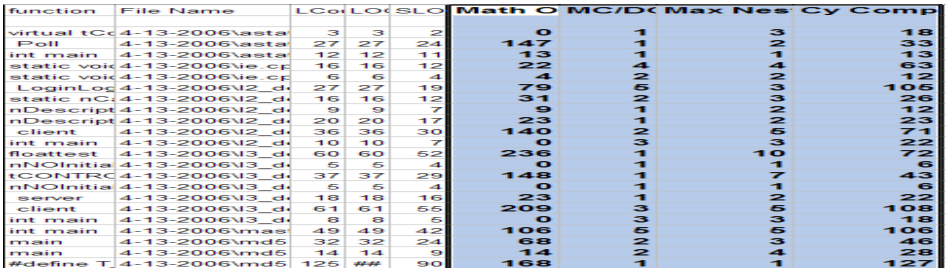
<table>
<tr><td colspan="2">Test Case Number 6: Testing other metrics</td><td>Target: Metric Class</td></tr>
<tr><td>Objectives</td><td colspan="2">Validate other metrics</td></tr>
<tr><td>Preconditions</td><td>Input</td><td>Expected Results</td></tr>
<tr><td>C/C++ directory existed.</td><td>SWCMetric.exe \\testDirectory<br>Sample Output</td><td>Parse list of the name of the files within the directory.<br>Pass</td></tr>
<tr><td colspan="3"> 
</td></tr>
</table>

Figure 7.7. Test case #6.

The tested metrics in Test Case 6 are: Number of mathematical operators per function, MC/DC, maximum nesting, and cyclic complexity. Cyclic complexity is suggested by the developer and so was not verified. Others are defined according to the

customer's definition. The algorithms invented were the best choice in terms of accuracy and performance out of several alternatives. The customer was providing testing and verification for all the above metrics, during the development cycle. Test Case 7 (Figure 7.8) is to validate the test rank formula.

| Test Case Number 7: Testing the test rank | | Target: Metric Class |
|---|---|---|
| Objectives | Validate the test rank metric. | |
| Preconditions | Input | Expected Results |
| C/C++ directory existed. | SWCMetric.exe \\testDirectory<br>Sample Output | Parse list of the name of the files within the directory.<br>**Pass** |

| function | File Name | LCour | LOC line | SLOC | Math Op | MC/C | Max Ne | Cy Com | Rank |
|---|---|---|---|---|---|---|---|---|---|
| virtual tCor | 4-13-2006\astat | 3 | 3 | 2 | 0 | 1 | 3 | 18 | 1.3 |
| Poll | 4-13-2006\astat | 27 | 27 | 24 | 147 | 1 | 2 | 33 | 4 |
| int main | 4-13-2006\astat | 12 | 12 | 11 | 13 | 1 | 1 | 13 | 1.9 |
| static void | 4-13-2006\ie.cp | 16 | 16 | 12 | 22 | 4 | 4 | 63 | 2 |
| static void | 4-13-2006\ie.cp | 6 | 6 | 4 | 4 | 2 | 2 | 12 | 1.6 |
| LoginLogc | 4-13-2006\I2_de | 27 | 27 | 19 | 79 | 5 | 3 | 105 | 2.3 |
| static nCal | 4-13-2006\I2_de | 16 | 16 | 12 | 31 | 2 | 3 | 26 | 2 |
| nDescripto | 4-13-2006\I2_de | 9 | 9 | 7 | 9 | 1 | 2 | 12 | 1.7 |
| nDescripto | 4-13-2006\I2_de | 20 | 20 | 17 | 23 | 1 | 2 | 23 | 2.1 |
| client | 4-13-2006\I2_de | 36 | 36 | 30 | 140 | 2 | 5 | 71 | 5 |
| int main | 4-13-2006\I2_de | 10 | 10 | 7 | 0 | 3 | 3 | 22 | 1.8 |
| floattest | 4-13-2006\I3_de | 60 | 60 | 52 | 236 | 1 | 10 | 72 | 10 |
| nNOInitiali | 4-13-2006\I3_de | 5 | 5 | 4 | 0 | 1 | 1 | 6 | 1.5 |
| tCONTROL | 4-13-2006\I3_de | 37 | 37 | 29 | 148 | 1 | 7 | 43 | 6 |
| nNOInitiali | 4-13-2006\I3_de | 5 | 5 | 4 | 0 | 1 | 1 | 6 | 1.5 |
| server | 4-13-2006\I3_de | 18 | 18 | 16 | 23 | 1 | 2 | 22 | 2.1 |
| client | 4-13-2006\I3_de | 61 | 61 | 55 | 209 | 3 | 5 | 108 | 7 |
| int main | 4-13-2006\I3_de | 8 | 8 | 5 | 0 | 3 | 3 | 18 | 1.7 |

Figure 7.8. Test case #7.

The test rank is one of the main goals of this project (see problem definition). It has a formula that is defined by the client. In general they indicate that most of the function test ranks are between 1 and 25. This is a metric that is also verified by the customer through

user testing. They are also willing to have charts that draw functions and their test rank values.

### 7.4. Regression Testing

This application evolved through many different deliverables or prototypes. Functionalities were incrementally delivered. In such an approach, regression testing was very important to make sure that earlier functionalities were working as expected after adding recent functionalities.

As previously mentioned, a small test oracle database was developed and preserved with all its correct results. An excel file compared columns from the last approved prototype with the recent one before making it available for the user. Each time a new functionality or metric is added to the code, it will be verified with the test oracle. After verification this new feature will be added to the test oracle.

One issue of testing could have been done better is to automate this process. We could have developed a small application that can automatically check the values between two different versions of the code. This is a case where the output format is designed in an easy way to make test automation feasible.

### 7.5. User or Acceptance Tests

The acceptance test is performed by the clients on the company actual code. They downloaded the application from the shared repository, performed the acceptance testing and provided the feedback through emails.

### 7.6. Performance and Robustness Test

Using Toshiba Notebook of 2.8 GHz Celeron CPU and 225 MB of RAM, a total directory of C/C++, code only, of 10.60 MB totals size. The application took about 40

seconds to finish gathering the overall metrics. This includes all intermediate stages of parsing all files from the different folders, cleaning the comments from each file, parsing it again and gathering all metrics, writing the output to a Console and a file, and finally deleting all temporary files and directories. With all the IO operations and loops the application has to go through, this is considered a good performance.

### 7.7. Installation Test

The application should be able to be installed with an install wizard.

Test Case Inputs are:

7.7.1. The computer is powered on and running Windows 2000 and above.

7.7.2. The SWMetrics CD is ready in the CD ROM drive.

7.7.3. Execute the SWMetrics Setup.exe program on the CD.

7.7.4. Choose if desired, a destination folder.

Expected Test Case Output is:

A. The SWMetrics Install Wizard executes.

B. The program by default will be installed to C:\SWMetrics.

### 7.8. Summary

The tests in this document express the tests needed to make sure this tool meets the requirements that were listed in the Requirements document for SWMetrics tool.

Testing in this project consumes a large percent of the development process. If formal methods and test automation were utilized, it is expected to lower the time spent on testing. The Scrum development process that is approximately adopted focuses on deliverables and time rather than on spending extra time earlier with the goal of saving it later. The process of verifying the results of the gathered metrics manually is tedious and

complicated. Automating the verification of the results with the test oracle would have helped the overall time performance in this project.

# CHAPTER 8. CONCLUSIONS

As the application was successfully delivered on the time and requirements expected, this was a successful project. Nevertheless, things can be better and there were some windows for improvements.

## 8.1. What Could Have Been Done Better?

This is a subjective self criticism. It reflects our experience working throughout this project. Here are some of the things that we think could have some improvements.

First, Table 8.1 shows the overall project summary

Table 8.1. Project summary.

| Purpose | Description |
|---|---|
| LOC/hours | This program has a round 1800 LOC. The estimated over all time effort for all its development stages is 500-800 hours. LOC/hour is approximately 3 (1800/600). |
| Defects / 1000 LOC (KLOC) | There were a total of at least 50 different errors that had to be addressed. 50/2 = 25 defects/KLOC. |
| Program size | 1800 LOC |
| Defects/hour | Giving a total testing time of 800, Defect/hour=50/800 = 0.06 error/hour. |

## 8.2. Description

### 8.2.1. Time

Looking at the LOC/hour, an average of 4 LOC/hour is low. Although there were many deliverables and functionalities, yet the algorithms themselves were complicated and error prone. Testing took the largest percentage of the whole development. This is expected for an agile development approach. I think time effectiveness could have been improved. Communication was a major weak point in this project. It affected the over all productivity. In an environment like our case, where there is no direct contact between the client and the developers, well defined requirements may help cover the shortage in communication. Communication will always play the major role in the failure or success of any project.

### 8.2.2. Defects

Theoretically, the defect\hour value wasn't too high. There were many times during this development where a single error could take many hours to fix. There are many factors that affect this value. The nature of the application is error prone. The parsing process requires a lot of algorithms and sequences, there are many streams of input and output data, and there are many occasions that a single algorithm or metric is defined more than one time.

### 8.2.3. Information Access

It was not possible for the developer to get any details about the business problem and requirements. The information any where else is limited especially about the way we

should define the algorithms, or if there is any type of standard definition for such metrics. Most of those metrics are subjective and could be interpreted differently by different people.

The area of metric tools is a very rich one. Different people and companies have different interests in their earlier projects. Software quality attributes in general, are not easy to define or specifically connect them to a single part of the project. For example software complexity is one of the software characteristics that are very important to know or define. In reality it is not untrue to say that almost everything in the project and the product may play a role in the software complexity.

This application can be used as a commercial tool in the reverse engineering field. It can easily work as a part of a bigger application. The output data is in a standard database format. The performance is very high and the data gathered is very reliable.

### 8.3. Software Mining

This tool can be the first block in a bigger project. Through data mining, we are able to efficiently leverage the strategic value of our data. Through data mining, we find some hidden relationships in our data that can never be found without it. Software mining can learn from data mining. The major difference between the two is that the learning in the data mining is about the relations between the data itself, where as the learning in software mining is for another data. We are trying to make hypothesis from earlier projects, and then see if it works for newer one as well. We want to use mining to evaluate the software internal and external qualities.

## REFERENCES CITED


1. Putnam, Lawrence and Ware Myers. Five core metrics. New York: Dorset House Publishing Co., 2003. 18 Apr. 2006. <http://www.dorsethouse.com/books/fcm.html>.

2. Rufai, Raimi. "Software metric tools survey". Course home page. 28 Aug. 2004. 18 Apr. 2006. <http://www.ccse.kfupm.edu.sa/~rrufai/toolsurvey/>.

3. Putnik, Zoran. "Metric tools for Java programs". Course home page. 05 May 2004. 18 Apr. 2006. <www.informatik.hu-berlin.de/ swt/intkoop/jcse/tools/metric_tools.ppt>.

4. Software Assurance Technology Center. "Software quality engineering". 2000. 18 Apr. 2006. <http://satc.gsfc.nasa.gov/assure/agbsec4.txt>.

5. Zage, Dolores. "An introduction to design metric research". Mar. 2001. 18 Apr. 2006. <http://www.cs.bsu.edu/homepages/metrics/introduction>.

6. Wikimedia Foundation, Inc. "Iterative development". 16 Apr. 2006. 18 Apr. 2006. <http://en.wikipedia.org/wiki/Iterative_development>.

7. Clifton, Mark and J. Dunlap. "What is Scrum". Online posting. 19 Aug. 2003. 20 Apr. 2006. <http://www.codeproject.com/gen/design/Scrum.asp>.

8. Wikimedia Foundation, Inc. "Scrum in management". 20 Apr. 2006. 20 Apr. 2006. <http://en.wikipedia.org/wiki/Scrum_(in_management)>.

9. Mosley, D.J. "Test case template for those who are using or would like to implement the use case modeling technique". 2002. 18 Apr. 2006. <http://www.geocities.com/xtremetesting/TCtemplate.html>.

10. SCG Smallwiki. "Detailed evaluation rules". Course home page. 2005. 18 Apr. 2006. <http://smallwiki.unibe.ch/ese2005lecturesmallwiki/laboverview>.

11. Zaitoni, Mohamed and Alain Abran. "A model to evaluate and improve the quality of software maintenance process". Course home page. 1996. 18 Apr. 2006. <www.lrgl.uqam.ca/publications/pdf/67.pdf>.

12. TopMind Systems LLC. "Capability Maturity Model". Jul. 2004. 18 Apr. 2006. <http://www.topmindsystems.com/home/cmm.pdf>.

13. Dukic, Ljerka and Jorgen Boegh. "COTS software quality evaluation outline". International conference on COTS based software systems. 2003. 18 Apr. 2006. <http://www.iccbss.org/2003/presentations/Beus-dukic_121025VN.pdf>.

14. McConnell, Steve. Code Complete. Redmond, Wash: Microsoft Press, 1993. 18 Apr. 2006. <evanjones.ca/books/code-complete.html>.

15. Balci, O and Dwight Barnette. Virginia Tech. "Software Engineering Lessons". Course home page. 2000. 18 Apr. 2006. <http://courses.cs.vt.edu/csonline/SE/Lessons/Qualities/Lesson.html>.

16. Hemenway, Kevin. "Web apps with Tiger". Online posting. O'Reilly. 20 Sep. 2005. 18 Apr. 2006. <www.macdevcenter.com/pub/a/mac/2005/09/20/apache.html>.

17. Connie, Smith and Lloyd Williams. Introduction to software performance engineering. Addison Wesley, 09 Nov. 2001. 18 Apr. 2006. <www.awprofessional.com/articles/article.asp>.

18. Open process framework. "Design evaluation". 16 Dec. 2005. 18 Apr. 2006. <http://www.donald-firesmith.com/Components/WorkUnits/ Tasks/ QualityEngineering/ QualityControl/DesignEvaluation.html>.

19. Gilb, Tom. "Design evaluation". 24 Feb. 2004. 18 Apr. 2006. <http://www.gilb.com/Download/DesignEvaluation.pdf>.

20. Nasa IV & V facility. "Independent verification and validation overview". Fairmont, West Verginia. 1997. 18 Apr. 2006. <www.ivv.nasa.gov/ivvservices/IV&V_Overview.ppt>.

21. Cockburn, Alistair. Home page. "Growth of human factors in application development". 1996. 18 Apr. 2006. <http://members.aol.com/acockburn/papers/adchange.htm>.

22. Chowdhary, Salam. "Role of metrics and outlier analysis in the software design process". Online posting. 1996. 18 Apr. 2006. <http://www.19.5degs.com/element/17747.php>.

23. Alsmadi, Izzat. "SWMetrics requirement and design documents". 2006. 18 Apr. 2006. < http://www.cs.ndsu.nodak.edu/~alsmadi/SWMetrics/Documents>.

24. Semantic Designs, "Incorporated. Software metric tools". 18 Apr. 2006. <http://www.semdesigns.com/Products/Metrics/index.html>.

25. Munro, M. "A measurement-based approach for detecting design problems in object oriented systems". Jul. 2004. 18 Apr. 2006. <www.cis.strath.ac.uk/~efocs/ home/Research-Reports/EFoCS-57-2005.pdf>.

26. Tomer, Amir. "Iterative Software Development –from Theory to Practice". Stickyminds.com. 2001. 18 Apr. 2006. <www.stickyminds.com/se/S3402.asp>.

27. Crow, Kenneth. DRM associates. “Capability Maturity Model”. 2000. 18 Apr. 2006. <http://www.npd-solutions.com/cmm.html>.

28. Martin, Robert and Mary Drozd. “Using Product quality assessment to broaden the evaluation of software engineering capability”. 1996. 18 Apr. 2006. <http://www.mitre.org/work/tech_transfer/pdf/se_capability.pdf>.

29. Wang, Y. “Design for test and software testability”. Course home page. 2003. 18 Apr. 2006. <www.ucalgary.ca/~ageras/wshop/ abstracts/2003/design-for-testability.htm>.

30. Petticord, Brett. “Design for testability”. 2002. 18 Apr. 2006. <http://www.io.com/~wazmo/papers/design_for_testability_PNSQC.pdf>.

31. O’rourke, Tom. “Guerilla Design Evaluation Heuristics”. PaineWebber, Inc. 1997. 18 Apr. 2006. <http://www.iro.umontreal.ca/~keller/Workshops/OOPSLA97/Papers/orourke.tom>.

32. Wüst, Jurgen. “SDMetrics”. 2006. 18 Apr. 2006. <http://www.sdmetrics.com/LoM.html>.

33. MIT open courseware. “Software requirement metrics”. Course home page. 2002. 18 Apr. 2006. <http://ocw.mit.edu/NR/rdonlyres/Aeronautics-and-Astronautics/16-355JAdvanced-Software-EngineeringFall2002/metrics2.pdf>.

34. Wallenberg, Angela. “Software testing”. Course home page. 2005. 18 Apr. 2006. <http://www.cs.chalmers.se /Cs/Grundutb/Kurser/ itProj2/testingLecture2005/testing.pdf>.

35. Baar, Bas de. “Software test strategy for time pressured projects”. Software Projects.org. 2006. 18 Apr. 2006. <http://www.softwareprojects.org/software-test-strategy.htm>.