Using Software Metrics Tools for Maintenance Decisions: A Classroom Exercise

Andrew D. Marshall Michael J. Katchabaw Michael A. Bauer
Computer Science Department
The University of Western Ontario
London, Ontario, Canada N6A 5B7
E-mail: {flash,katchab,bauer}@csd.uwo.ca

Abstract

In this paper we explore the use of software metrics tools to guide software maintenance decisions. A senior undergraduate class was given a copy of QUIPU, an implementation of the X.500 directory standard, and asked to determine which component of the system would be the best to rewrite in order to improve the overall quality of the entire software system. We describe the course structure and the information and tools provided to the students. The primary tool used by the students was Rigi, a program understanding tool. We present the methodology and results of one of the successful groups of students. Based on the experiences of the students, we conclude with suggestions for tool designers.

1. Introduction

One of the points of agreement in the software engineering community is recognition of the disproportionate allocation of resources to software maintenance. Of the activities encompassed by the term software maintenance, in this paper we are particularly interested in renovation [4]. We define renovation as a preventive maintenance activity: rewriting parts of a system to improve their quality and to ease future maintenance tasks. Our focus in this paper is on the use of program understanding and software metrics tools to guide system renovation, to facilitate targeting of modules whose renovation would best improve the overall quality of the system. That is, the tools would identify modules with low software quality; renovating those modules would be a step toward improving the quality of the entire system. In theory, this is a simple idea, and one which might help reduce the resources expended on software maintenance; in practise, we need to evaluate various tools to identify which are most appropriate, to identify limitations ans strengths of the tools and, finally, to gain "hands-on" experience in applying tools to renovation decisions on real programs.

Recently, one of us (Bauer) taught a senior undergraduate course entitled "Software Processes, Measurements and Quality". The course concentrated on the use of software metrics to evaluate software quality. We decided to use the course to address the use of tools in software renovation. To that end, the major assignment in the course involved putting software metrics into practice by examining a large software system with a view toward deciding which part of the system would be best to renovate.

Our purpose in writing this paper is twofold. First, we intend to provide insights into the use of tools for this task, draw conclusions on their efficacy for assessing software quality, and make suggestions for future tool developers. Second, we feel that our approach to teaching software metrics was useful, and we wish to provide information for others to replicate our approach.

In the remainder of this paper, we first present a description of the course in which the research reported here was performed. Section 3 describes Rigi [10], the program understanding tool that was the primary tool used by the students. Section 4 presents the methodology and results of one of the teams of students. Finally, section 5 contains our concluding remarks.

2. Overview of the course

The course was organized to simulate an industrial environment. The students were employees of a mythical company whose job was to evaluate the quality of another company's software product to identify its

Appeared in the Proceedings of the International Symposium on the Assessment of Software Tools, Toronto, Canada, May 1996

key weaknesses for later improvement. The students were divided into 6 teams of 4-5 students each. The initial teams were assigned, at random, by the instructors. The team members had a set of prescribed roles to fill (shown in Figure 1); each team had to decide among themselves who would assume each role. The people playing the same role in different teams were instructed to interact to help each other in their duties, the goal being to implement a matrix management organization. Finally, the two course instructors took on the roles of company executives to whom the team leaders reported.

The software system examined by the students was version 8.0 of QUIPU [7, 12], an implementation of the X.500 directory standard [2, 3]. QUIPU is but one component of a larger system, ISODE, the ISO Development Environment. QUIPU's source code consumes about 3.5 Megabytes of disk in approximately 550 files spread over about 20 directories.

The problem the students were assigned was to determine which component of the system would be the best to rewrite in order to improve the overall quality of the entire software system. The students were required to produce their results in two phases. In the first phase, they were to use a program understanding tool called Rigi [10] to inform their decision. In the second phase, they were to use Rigi, along with any other tools, metrics, or techniques they deemed appropriate.

2.1. An overview of X.500

To understand QUIPU and the rest of this paper, it is first necessary to understand the basics of the X.500 directory service. The X.500 Standard specifies a Directory Service that provides and manages information about entities. The information held by the Directory is the Directory Information Base (DIB). The DIB consists of entries (or objects) that contain information about entities. Each entry consists of a set of *attributes*, each with a type and one or more values. The types of attributes that are present at each entry are dependent on the *class* of entities that the entry describes. Classes can be both predefined and user defined. Entities in the DIB are represented by entries in a hierarchical name space called the Directory Information Tree (DIT). Entries are placed in the DIT according to the organizational relationships between the entities that they represent. The X.500 Abstract Service Definition defines abstract ports and operations that provide the user functionality for retrieving, searching, and modifying directory information. These operations form the Directory Access Protocol (DAP). The Directory Service (including the DIT) is distributed over physically separated entities (such as, computer hosts) that are Directory System Agents (DSAs). This distribution is transparent to the user. DSA's communicate with each other using the Directory Service Protocol (DSP). Each user or user process is represented by a Directory User Agent (DUA) that interacts with the Directory. The functional model of the Directory Service is illustrated in Figure 2.

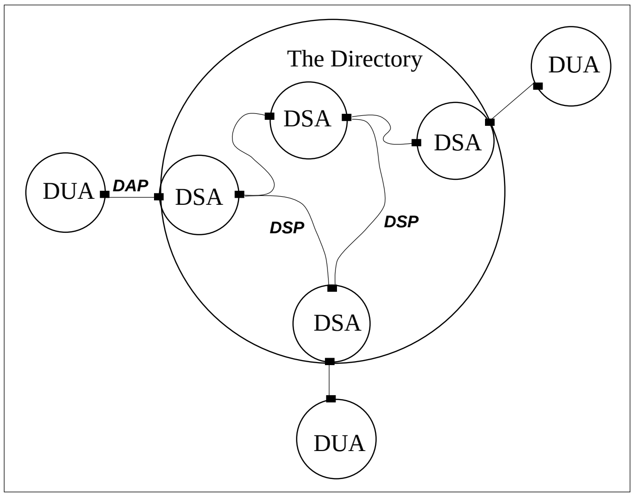


Figure 2. Functional Model of the Directory Service

3. Rigi: the primary tool

Rigi is a system for reverse engineering developed at the University of Victoria [10]. The process of reverse engineering involves the identification of components and their dependencies, and the extraction of design information and system abstractions.

Rigi consists of three major software components:

- a parsing program called *rigireverse* which parses the subject software to identify the objects of the system and the relationships among them

- a database server, called *rigiserver*, which manages the resource-flow graph, created by rigireverse

- an interactive graph editor called *rigiedit* which allows browsing and analysis of the graph, both visually and quantitatively, allowing the traversal and modification of the graph structure in the database.

Appeared in the Proceedings of the International Symposium on the Assessment of Software Tools, Toronto, Canada, May 1996

Team Leader Responsible for organizing the team, team meetings, agenda, minutes, overall coordination of team activities; Interacts with other Team Leaders in overall project management and meetings; Interacts with Company Executives (instructors).

Rigi specialist Responsible for becoming the team's expert on Rigi, helping other team members to understand the tool and how to use it; interacts with other Rigi specialists.

Architecture specialist Responsible for delving into the target system, understanding the structure, components, providing support for Rigi analysis of the software.

Process person Responsible for team processes and measurement, tracks team's activities, documents team processes; In Phase 2 is responsible for identifying additional measurements and tools for analysing the target software.

Documentation specialist Responsible for team documentation.

Others Additional team members can assist in all activities or support specific tasks; e.g., another Rigi specialist, a process person to help find or develop analysis software.

Figure 1. Team Roles [1]

3.1. Capabilities of Rigi

After parsing the input source code, and creating the necessary database information, Rigi can then provide analysis and results in two main forms. The first form is a graphical overview of the various systems, subsystems, files, and other components of the software being analysed. The second form is a collection of text-based report mechanisms that will provide data based on several metrics.

Based on the graphs produced by Rigi, one can visualize the modularity of a component and the interaction between its subsystems. Rigi shows different arc types connecting nodes together in its graphs. The two most commonly used are: *call arcs* and *data arcs.* The *call arc* type is from a source node that makes a function call to the destination node. Both nodes represent functions. The *data arc* type represents one of the following:

- access to the internal structure of a data type from a function
- containment of one data type from within another
- reference of one data type from within another

Looking at the relative number of incoming and outgoing arcs to each node in the graphs, it is also possible to estimate a system's complexity.

Using its report mechanisms, Rigi can provide low-level details of a subsystem, called an *exact interface.* The report generated consists of three kinds of information for the selected subsystem: *provisions, requirements,* and *internalizations.* A *provision* is a dependency from a node inside the subsystem to a node outside the subsystem. A *requirement* is a dependency from a node outside the subsystem to a node inside the subsystem. An *internalization* is a dependency between two nodes inside the subsystem.

Rigi can evaluate the quality of a selected subsystem according to a set of measures embodying established software engineering principles with a *graph quality report.* This report gives the *control encapsulation* and the *data encapsulation.* The *control encapsulation* measure increases with the number of control flow dependencies between nodes inside the subsystem, and decreases with the number of control flow dependencies from nodes inside the subsystem to nodes outside. This favours localized control and small interfaces. The *data encapsulation* measure increases with the number of local references to data types, and decreases with the number of external references to data types. This favours data encapsulation and object-oriented designs.

4. One team's approach

One of the more thorough reports was produced by "Team A". In this section, we describe the approach they took to solve the assigned problem.

As mentioned above, the first phase made use of Rigi, a tool with which the students were not initially familiar. The results of the first phase amounted to an evaluation of QUIPU and of Rigi, conducted in parallel. The second phase made further use of Rigi, as well as a number of public domain metrics programs,

Appeared in the Proceedings of the International Symposium on the Assessment of Software Tools, Toronto, Canada, May 1996

code inspections by volunteers from the Department outside the class, an e-mail survey of users registered with the QUIPU mailing list, and contact with one of the QUIPU developers.

The following section describes how the team applied Rigi to QUIPU and their opinions of its usefulness and applicability.

4.1. Applying Rigi to QUIPU

Both the graphical and textual reports produced by Rigi provide an indication of a system's modularity and complexity. Using this information the team's goal was to estimate values for each of the metrics below, and use them to guide the evaluation of the quality of QUIPU. The definitions for the metrics used were taken from [13].

clarity an overly complex system is more difficult to read, understand and use.

efficiency if a module is utilized frequently and is overly complex, the *efficiency* of the entire system may be adversely affected.

modifiability many errors can be introduced when attempting to modify a complex component, thereby decreasing its *flexibility* and making *maintenance* more difficult.

testability increased complexity reduces a system's *testability* because exhaustive testing becomes more difficult.

understandability generally speaking, a system's *understandability* is inversely proportional to its complexity.

4.2. Analysis of Rigi's reports

In building the initial Rigi database, several problems were encountered due to constructs in the QUIPU source code that the Rigi parser interpreted as fatal syntax error. Once the syntax errors were corrected, Rigi correctly parsed the code and generated its database.

After Rigi had built its initial database, the team initially focused on analysing the graphs produced by Rigi directly. However, due to the size and complexity of QUIPU, it was difficult to recognize problem areas and come to any kind of conclusion. As a result, they had Rigi generate its results in the form of a series of reports. However, again due to the size of QUIPU, these reports were large and difficult to interpret and use. Because of this, an automated customization process was employed to assist in summarizing Rigi's observations on the QUIPU code.

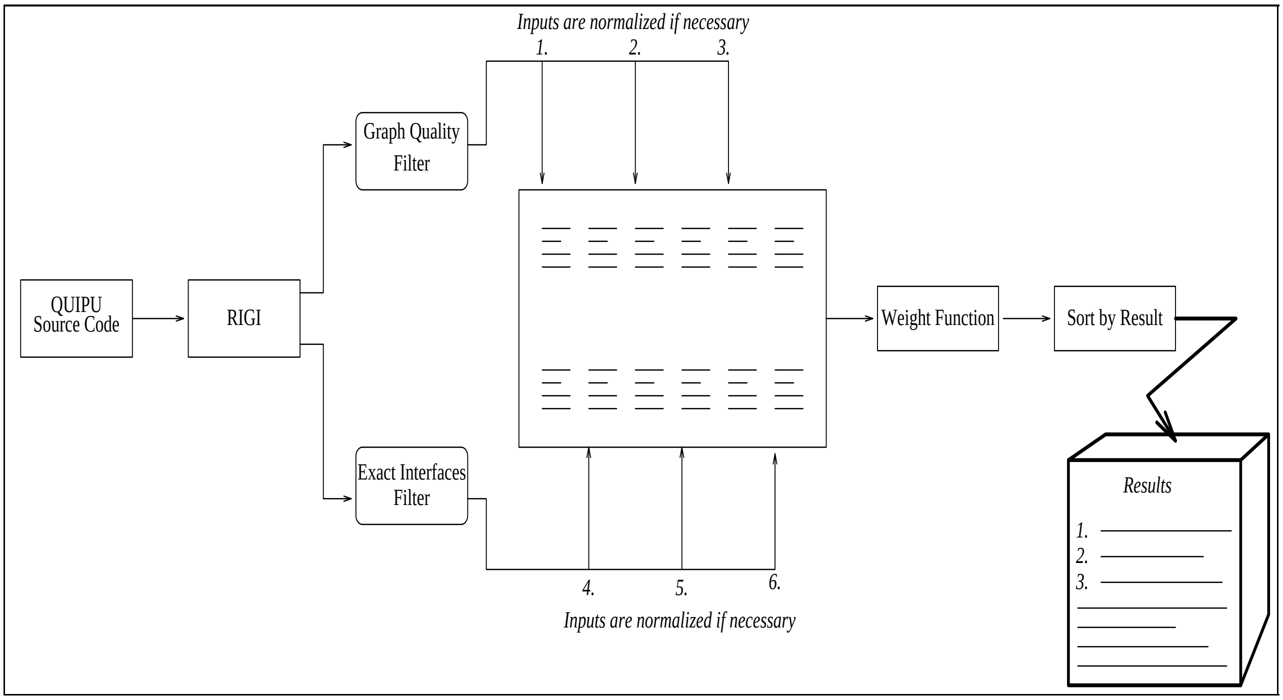
Automated analysis was used to narrow down the potential components for improvement to a list of "candidates", based on software quality statistics generated by Rigi. The automated analysis was performed by a collection of small filter programs and built-in Unix shell commands.

4.2.1. Data generation and processing. Rigi produced two reports, one on graph quality, and one on exact interfaces. The exact interfaces report includes three different types of information on the modules: provisions, requirements and internalizations. The graph quality report, on the other hand, was comprised of: partition quality, control encapsulation, and data encapsulation. It was agreed, by all teams in the project, that certain types of information were more relevant to assessing the modularity quality of QUIPU than others. Thus, it was necessary to judge the extent to which each statistic was useful and then provide a method of weighting this information appropriately. The modules with the highest ranking based upon the following Rigi statistics and their associated weights then became candidates for modification, based on modularity.

Provisions Provisions are given as an integer value representing the number of times a particular module is called. The magnitude of the provision value is an important statistic since modifying a module with greater scope (called more frequently) will have a greater effect on the functionality of the program. Furthermore, if there is a high level of complexity or a lot of errors in one module, it will be amplified by frequent calls to that module. It is possible for the number of provisions, when considered on its own, to be a misleading statistic. To prevent this prospect, one must also pay close attention to the requirements since a module with many provisions and few requirements is most likely a well coded object (consider a small library, or encapsulated data type). The number of provisions was normalized to produce a value between zero and one representing the number of provisions each module had as a fraction of the largest number of provisions of all modules. For the aforementioned reasons, provisions was given a weighting factor of five.

Requirements The requirements data is output in a manner similar to that of the provisions: an integer representing the number of requirements for a given module is output. Requirements are a significant factor since a high number of require-

Appeared in the Proceedings of the International Symposium on the Assessment of Software Tools, Toronto, Canada, May 1996



Key:

1) Data encapsulation	2) Control Encapsulation	3) Partition Quality
4) Provisions	5) Requirements	6) Internalizations

Figure 3. Rigi Data Information Flow

ments usually implies an over-extended module; the module should be broken down into several simpler modules. Also, improvements in such a module are magnified by the many calls to it. Requirements, like provisions, is normalized and weighted by a factor of five when factored into the ranking equation.

Control Encapsulation The more localized the functionality of a particular module, the larger the control encapsulation score. This metric is obviously quite significant since outside control often leads to increased complexity, dependencies, and side effects. To assure that poor encapsulation implied a higher final score, to be consistent with the previous metrics, the team took the complement $(1-x)$ of Rigi's encapsulation result. (Rigi outputs a floating point number between zero and one. A score of one denotes extremely high control encapsulation.) The weighting it was assigned was three as it was considered important, but not as important as the previous two metrics.

Data Encapsulation This metric gauges the extent of data encapsulation and object oriented design. The metric for this attribute had to be inverted with the same reason used for control encapsulation. Data encapsulation is an important factor but was only weighted by a value of two since there was little variance in this quality from module to module.

Internalizations This factor was not very helpful in choosing a module to modify since a high score, and similarly a low score, could be either good or bad with respect to software quality. A high number of internalizations could imply that a piece of code was very modular (tightly coupled object) or it could imply that the code was very treacherous (like spaghetti code). Only direct code inspection could explain Rigi's observations in this case. This factor was deemed only marginally important and given a weighting of one because of this.

Partition Quality When generating the graph quality report, Rigi calculated only a single partition value—for the entire program—which did not assist the analysis. When the team attempted to determine partition quality for individual modules, this value was invariably one, most likely because

Appeared in the Proceedings of the International Symposium on the Assessment of Software Tools, Toronto, Canada, May 1996

Let $x \in$ component files of QUIPU:

$$
\begin{aligned}
\mathrm{P} &= \frac{\mathrm{Provisions}(x)}{113} \times 5 \\
\mathrm{R} &= \frac{\mathrm{Requirements}(x)}{65} \times 5 \\
\mathrm{I} &= \frac{\mathrm{Internalizations}(x)}{58} \times 1 \\
\mathrm{D} &= (1.0 - \mathrm{Data\_Encapsulation}(x)) \times 2 \\
\mathrm{C} &= (1.0 - \mathrm{Control\_Encapsulation}(x)) \times 3 \\
\mathrm{PQ} &= \mathrm{Partition\_Quality} \times 0 \\
\\
\mathrm{Score}(x) &= \mathrm{P} + \mathrm{R} + \mathrm{I} + \mathrm{D} + \mathrm{C} + \mathrm{PQ}
\end{aligned}
$$

Figure 4. Weighting equation for Rigi's output parameters

a single file was considered to be a module, contrary to Rigi's views. Again, this made the partition quality useless in the analysis. Furthermore, the concept behind partition quality is better measured by the data and control encapsulation metrics. Hence, this quality was weighted with a value of 0.

Figure 3 illustrates how Rigi's quantitative output was massaged to produce a list of modules and their associated metric scores and to transpose this data (through attribute weights) to produce a list of the best candidates for modification, considering only modularity.

4.2.2. The weighting equation. The equation shown in Figure 4 was used to produce the weighted list of QUIPU components. This equation (agreed upon by members from the majority of the teams), whose inputs are the six metrics produced by Rigi, produces a single value representing each module's score. The denominator values (113, 65 and 58) used to normalize the respective metrics were the maximum values for those metrics.

The higher the score for a given module, the more likely it is that the module requires improvement. Sorting the list into descending order, modules most in need of improvement will appear closer to the top of this list (a partial listing is presented in Table 1). Hence, these modules were examined in more detail to select the module most in need of re-writing and improvement. This is not to say that there are not exceptions; indeed some poor modules were examined that were not at the top of this list.

4.3. Limitations of Rigi

While Rigi provides great power and flexibility in its analytical capabilities, the students discovered several shortcomings. These include:

- Rigi can become corrupt or even crash when it is given a large system to evaluate. (Use of a newer version of Rigi might have eliminated this problem.) This occurred while attempting to have Rigi parse the entire QUIPU system including all of its header files. As a result, header files were not included in the evaluation, possibly leading to an incomplete or inaccurate analysis.
- Rigi appears to give some incorrect assessments of the number of provisions made by some source files it parses. (This occurred in files such as `pe.c`, in which it incorrectly measured 113 provisions made by a non-existing function in that file.)
- Without knowing exactly how the graph qualities generated by Rigi are calculated, it is difficult to determine their significance and influence on software quality.
- It is difficult to gauge the significance of some of the statistics Rigi produces. For example, it is difficult to say how a change in the data encapsulation value from, say, 0.60 to 0.75, corresponds to noticeable differences in the source code.
- Rigi does not explicitly tell the user anything about software qualities such as: accuracy, conciseness, consistency, correctness, integrity, performance, portability, reliability, or reusability [13]. Rigi is only as powerful as the interpretation of its output.

As a result, these shortcomings must be taken into consideration when using Rigi to evaluate other software products. One cannot simply accept Rigi's analysis without further validation.

4.4. Metrics

Rigi provided considerable valuable information on the QUIPU system, particularly relating to modularity. However, the team felt the need to gather more information from other tools before arriving at a decision as to which QUIPU component could be selected to most improve its overall quality. As a result, a collection of tools was gathered that could provide additional information on QUIPU. Most of the tools were part of a public domain collection, originally assembled by

Appeared in the Proceedings of the International Symposium on the Assessment of Software Tools, Toronto, Canada, May 1996

File Name	Provisions	Requires	Internals	Control Encapsulations	Data Encapsulations	Weightings
dsap/common/pe.c	113	0	0	0.008772	1.000000	7.973684
quipu/dsa_chain.c	10	65	20	0.218750	1.000000	7.343750
quipu/ds_bind.c	4	16	0	0.050000	0.500000	3.850000
acsap/aetdase.c	2	7	2	0.300000	0.333333	3.433334
quipu/dsa.c	10	3	2	0.200000	0.500000	3.400000
quipu/turbo	3	3	1	0.285714	0.500000	3.142858
quipu/sys_init.c	0	3	0	0.333333	0.500000	3.000001
quipu/turbo/edb2dbm.c	3	2	1	0.333333	0.500000	3.000001
quipu/ds_init.c	42	0	0	0.023256	1.000000	2.930232
dsap/common/attrv.c	38	0	0	0.025641	1.000000	2.923077
quipu/shadow.c	3	24	0	0.035714	1.000000	2.892858
dsap/common/dn_seq.c	15	7	0	0.043478	1.000000	2.869566
dsap/common/quipu_sntx.c	4	17	0	0.045454	1.000000	2.863638
dsap/common/parse_error.c	20	0	0	0.047619	1.000000	2.857143
dsap/common/std_sntx.c	1	17	0	0.052632	1.000000	2.842104
dsap/net/dsaprovider.c	15	3	0	0.052632	1.000000	2.842104
⋮	⋮	⋮	⋮	⋮	⋮	⋮
dsap/net/ds_types.c	5	37	58	0.584158	1.000000	2.247526
⋮	⋮	⋮	⋮	⋮	⋮	⋮
quipu/ds_search.c	6	9	5	0.285714	1.000000	2.142858
⋮	⋮	⋮	⋮	⋮	⋮	⋮
quipu/dish/fred.c	12	24	16	0.307692	1.000000	2.076924

Table 1. Selected QUIPU module weighted scores

Brian Renaud, Rick Cobb and Marty Leisner, obtained through the Usenet newsgroup `comp.sources.unix` (Volume 20). The tools are described below.

4.4.1. The Halstead tool. The Halstead tool provides various indicators of a module's complexity based on Halstead's software science [6]. The module is examined to determine the following:

n_1 == number of unique operators
n_2 == number of unique operands
N_1 == number of total operators
N_2 == number of total operands

The program writes these output fields:

- File Name
- Program Length ($N = N_1 + N_2$)
- Program Volume ($V = N \log_2(n_1 + n_2)$)
- Program Level ($L = (\frac{2}{n_1}) * (\frac{n_2}{N_2})$)
- Mental Discriminations ($\hat{E} = V/L$)

The program volume (Halstead Volume) is most useful and seems to be reasonably well correlated with error counts for modules. In terms of output, the team found that the program length, volume, level and mental discriminations were all valuable indicators.

4.4.2. The McCabe tool. The McCabe tool determines function complexity based on the McCabe model of program complexity [8]. McCabe produces the following output:

- File name
- Function name
- Complexity
- Count of return statements

The program documentation recommends that a complexity over 10 is cause for further examination. The number of returns in a function is also important in predicting error counts for that function, perhaps even more important than the complexity. As a result, the team considered both the complexity and number of returns from this tool as valuable information.

4.4.3. The KDSI tool. The KDSI tool (for thousands of delivered source instructions) computes several source code metrics, including:

- Number of statements
- Lines of code
- Comments
- Comment lines
- Blank lines

Appeared in the Proceedings of the International Symposium on the Assessment
of Software Tools, Toronto, Canada, May 1996

4.4.4. The CCount tool. The CCount tool, written by Shane Hartman of Software Productivity Research Inc. gathers various statistics from C source programs. The main number counted is non-commented source statements (NCSS). The tool also reports the number of procedures and functions, comments, blank lines, and total source lines. NCSS is estimated by counting the number of semicolons, not counting those in string constants or comments. A line that contains a semicolon and a comment will be counted in both categories.

While this program duplicated some of the metrics calculated by tools in the Renaud, Cobb and Leisner package, it gave some additional information, and verified the data from the other programs.

4.4.5. The lc tool. The lc tool is a simple program that counts the number of lines of code, lines of comment, and blank lines in a C program. It outputs several values including:

- Pure code - lines of code with no comments.
- Pure comment - lines of comment with no code.
- Both code and comments - lines containing both.
- Blank lines - lines with neither code nor comments.

Once again, this tool duplicates some of the data gathered by other tools, thereby providing a certain degree of validation.

4.4.6. The CSize tool. The CSize program, written by Christopher Lott, performs lexical analysis on C files to collect several size measures: total lines, blank lines, lines with comments, non-blank and non-comment lines, semicolons, non-comment source statements (NCSS), and preprocessor directives.

Again, this program duplicates a lot of data calculated by other tools, but provided validation for the other programs.

4.4.7. The Metre v1.08 tool. The Metre tool, written by Paul Long, reports the following metrics for a file:

- McCabe's Cyclomatic Complexity Index
- Lines of code
- Lines of executable code
- Lines of comments
- Lines of whitespace
- Number of statements
- Number of functions

Maximum, minimum, average, and total metrics are provided where appropriate.

4.5. Summary of metrics results

The team found that the results from the metrics tools were essentially consistent with those produced by Rigi; no new insights were revealed.

There was, however, one problem with many of the tools used: compiler directives, especially `#ifdef` directives, are not handled directly nor consistently. The programs that use the C pre-processor to perform some analysis on the code suffer because the pre-processor handles and eliminates them first. Many other programs that do not use the pre-processor, like the Halstead and McCabe tools, completely ignore the directives. As a result, conditionals like `#ifdef` are not counted in many of the metrics calculated. Ordinarily, this would not be a concern, but some files in QUIPU, like `dsa.c` have 43 `#ifdef` statements. Since these are conditional statements, they should increase the calculated complexity of the code analysed. However, for the reasons given above, this is not the case. This is particularly unfortunate because, from a programmer's point of view, `#ifdef` statements are often more difficult to follow than `if` statements. As a result, the team added counts of such directives, where appropriate, to the metrics calculated.

Similar problems with inconsistent treatment of code by different tools have also been noted by Murphy, Notkin, and Lan in their study of C call graph extractors [11].

4.6. Departmental survey

We now move from the team's quantitative approach to a more qualitative assessment. The tools that were used were based on generally accepted software metrics, such as the work of McCabe and Halstead. However, these metrics cannot measure all aspects of software quality, including such aspects as readability and understandability. Evaluating these qualities requires subjective judgement of experienced programmers. The team also thought it wise to verify their interpretation of the numeric results by having a number of people examine candidate routines without the aid of automated tools.

Volunteers were solicited from the Department's undergraduates, graduates (both Masters and Ph.D. students) and faculty. The team assembled a list of four candidate modules based on data compiled with the use of the software evaluation tools and added one control module (`nrs_info.c`). The control module was one which, according to the metrics, would be an unlikely candidate as the worst module and was thus unlike the other modules. The software evaluators were given

Appeared in the Proceedings of the International Symposium on the Assessment of Software Tools, Toronto, Canada, May 1996

Module	Programmer							Average
	1	2	3	4	5	6	7	
ds_bind.c	2	3	3	3	2	3.5	2	2.64
ds_search.c	1	3	2	4.5	4	4	1	2.79
dsa.c	1	3	4	4	3	2	1	2.57
ds_chain.c	3	3	2	4	3	5	2	3.00
nrs_info.c	3	3	4	2	4	1	2	2.71

Figure 5. Readability ratings

Module	Programmer							Average
	1	2	3	4	5	6	7	
ds_bind.c	2	2	3	3	2	3.5	2.5	2.57
ds_search.c	1	4	2	1	4	4	2	2.57
dsa.c	1	3	4	2	3	2	1	2.29
ds_chain.c	3	3	2	2	3	5	2	2.86
nrs_info.c	3	3	4	4	4	1	2	3.00

Figure 6. Understandability ratings

listings of the candidates and asked to examine each module and make observations based upon the following definitions:

Readability how easy it is to read the code (comments, spacing, indentation, variable and function names);

Understandability how easy or difficult the code is to understand.

The evaluation was divided into two main questions. In part one, the candidates were asked to rate each module (on a scale from one to five, where one is poor and five is good) based on the understandability and readability criteria. The evaluators were encouraged to include justifications for their ratings. Part two asked the evaluators to estimate how long it would take to understand each of the candidate files. The survey was concluded by providing an area for the evaluators to make comments or observations.

4.6.1. Departmental survey results. Only seven accurate results (from nine respondents) were obtained from the surveys because some of the questions were not followed as they should have been, apparently due to time constraints on the evaluator.

Figures 5 and 6 summarize the readability and understandability ratings, respectively, for the various modules.

The observation sections of the returned surveys contained comments such as “atrocious style”, “hideous code”, “this is hack code” and many suggestions to improve both understandability and readability. Most of the comments were directed at the code in general, without identifying specific files. Most people complained of a lack of comments, spacing, and good variable names, and felt that the code could only be understood with the assistance of external documentation, such as an X.500 directory standards manual. They also observed that the functions were too long and too complex—one respondent questioned whether or not it was actually C code. The time estimated to understand the code varied from respondent to respondent, but most felt that it would take days to understand most of the modules examined. Without doubt, the respondents felt that, given the choice, they would not like to work with any of the modules.

The numeric results from the survey did not show a high degree of discrimination, however, they confirmed the team’s conjectures about the quality of the modules. Underscoring the subjective nature of quality assessment, the team noted that the module they considered to be the control in the survey did not fare significantly better than the others.

4.7. Mailing list survey

The final step in the team’s analysis was to contact actual QUIPU users. Their rationale was that their prior analyses had been conducted without input from any experienced QUIPU users, and it would be reasonable to expect that such users might have formed opinions that would assist the team in coming to a decision.

A mailing list of QUIPU users exists; the people on it were mailed a survey asking a series of questions relating to the accuracy, portability, modifiability, readability, and general quality of QUIPU. Given the short time frame (one 13-week term) in which the entire project was collected, it is not surprising that only six responses were received. The responses received, however, did contain a considerable amount of valuable information. The responses, and some conclusions are listed in the following section.

4.7.1. Mailing list survey responses. Actual responses are given below. Not all respondents answered all questions, those received are associated with the appropriate questions. The term(s) in parentheses at the end of the question give the software quality metric the question was intended to address.

Question 1 *Have you experienced any incorrect results (bugs) from any function (or file)? Under what conditions did this occur? (correctness, reliability)*

Response 1 Many. Stress testing, ie. reliability. Functional bugs in QUIPU are generally easy to find.

Appeared in the Proceedings of the International Symposium on the Assessment of Software Tools, Toronto, Canada, May 1996

Historically, QUIPUs biggest problem was memory leakage which was exacerbated by the complexity of its internals and the fact that it predated tools like Purify. All my attempts at introducing malloc debugging blew up, even GNU malloc.

Response 2 We programmed DUA using quipu. When we used ds_read() function. The return values of that were good. But stdout may be closed, so couldn't use "printf". Additionally, couldn't reuse ds_read() in the same process. If you know reason why of it, please tell me that, sir.

Question 2 *If you have installed QUIPU personally, were there any files you had to alter in order to get it running on your system? If so, which files? (portability)*

Response 1 Generally not. At this stage (IC R2.0v3) there are definitely none. Portability of ISODE and QUIPU was always good and has got better over subsequent releases.

Response 2 Many. :-) (When I compiled on NetBSD...)

Question 3 *Are there any aspects of QUIPU (other than speed) which you would like to see improved? Please explain. (usability)*

Response 1 Internal documentation. IC are moving in all the right directions - back end database, OOP redesign, X.500 1993 etc.

Response 2 Usage of memory.

Question 4 *Have you tried to modify any of the source code? What files were the most difficult to modify? (modifiability)*

Response 1 Yes. Anything to do with the operation/task structure.

Question 5 *If you have worked on the source code files, did you find any that were significantly less readable or understandable than others? If so, which were the worst? (understandability)*

Response 1 Parts of QUIPU suffered from what was clearly student engineering. This is understandable given its background. It also suffered from creeping featurism due to its huge success in pilot. There were some features, notable attribute inheritance which did not work reliability until the second IC release. QUIPU is horribly complex but does work well.

Response 2 Having worked on some of the source code for QUIPU/ISODE, my main problem with it is that it's complexity arises principally from the constant pre-occupation with exceptional cases - e.g. in the unlikely event that a program cannot allocate a block of memory, the lowest level routine passes NOTOK back to it's caller, this must get passed up about 10 levels of subroutine call before anything gets done with it.
The style of the code is generally:

```
if (perform_some_step(...)!= OK
   { do_something_trivial();
     return NOTOK;
```

I would be very interested to see how much shorter/more maintainable/faster/better it would get if the code concentrated on normal events coupled with some unified means of handling exceptions.

4.7.2. Mailing list survey analysis. The first observation is that even with only six responses, a wide variety of results were received. Most of the responses appear to have been submitted by QUIPU users, as opposed to developers. QUIPU does appear to have some problems in terms of reliability and portability, but this seems to vary from site to site. All of the respondents agreed to some extent that the code is difficult to understand, and thus can be difficult to work with. The users felt that redesign was needed, that bugs needed to be fixed, and that the 1993 X.500 standards and functionality should be adhered to.

The team conjectured that the small number of responses implied that either most users do not pay close attention to the mailing list, or that most users are reasonably satisfied with QUIPU. Generally, dissatisfied users tend to be quite vocal with respect to program inadequacies.

4.8. Team A conclusions

In making its decision, the team relied heavily on the company metaphor underlying the structure of the course. The scenario under which the teams were operating was one in which they were evaluating the software of another company to assess its quality. The team was of the view that customer satisfaction was a vital consideration in assessing software quality. Their conclusion was that one of the main routines, `dsa.c` would profit most from rewriting. This module forms the core of the QUIPU DSA and must be modified in order to meet the needs of the customer and to make

Appeared in the Proceedings of the International Symposium on the Assessment of Software Tools, Toronto, Canada, May 1996

doing so easy and efficient. This module is critically important because the system very heavily depends on it. No matter what problems there are in other files, one can simply go nowhere if there are problems with this file. The results from Rigi and the other metrics also supported the choice of this module.

Based on the QUIPU user survey (users being "customers" in the scenario), the team identified three customer concerns that informed their decision beyond blindly following the results of the tools used.

The number one customer concern with QUIPU was X.500 1993 standard support, which implies that the code must be both expandable and flexible. Experience from developers [5] indicated that the code in `dsa.c`, part of the core QUIPU code and task handling routines, cannot support the new functionality that is needed to provide this new support. The modularity report from Rigi, ranking this module the fourth worst overall, made this even more clear. Without sufficient modularity and encapsulation (this module had poor control and data encapsulation), building upon this code is particularly difficult because this code is the foundation of the system.

The second largest concern with customers was the correctness and reliability of the QUIPU code. From a brief code inspection of the file, it appeared that this file had more potential problems than any of the other code files that were examined. Ensuring that correctness and reliability is maintained through testing could be made even more difficult by the routine's disabling of faults and core dumps until the occurrence of several critical errors, making it difficult of even impossible to determine the initial fault.

The last main customer concern was from customers using QUIPU for their own development—they want to code to be readable, understandable, and maintainable. The departmental survey indicated that this file was the most unreadable and difficult to understand module of those included in the survey. The large number of `#ifdef` statements supported this as well—one can hardly determine what the code does when one cannot tell which code is being compiled in the first place. This also contributes to poor readability because there is no flow to the code whatsoever; the `#ifdef` statements break up this flow. As for maintainability, developers' experience indicated that this code is indeed very difficult to maintain [5].

5. Concluding remarks

We viewed the course as a success, a view supported by comments from the students at the end of the course. Putting software metrics to the test with a specific goal in mind helped the students to develop their understanding of the strengths and weaknesses of the metrics.

It is clear that evaluating software quality automatically is still an unrealistic goal for tool designers. The best that tools seem able to do is to point people in the right direction; the final decision comes down to the application of human judgement.

There are also lessons to be learned by the designers of tools like Rigi.

- Support for code that doesn't compile.
- Support for incremental examination of source code.
- The ability to tailor reports to meet specific needs.
- QUIPU is large and poorly structured, however, it is not likely the most poorly structured program to which tools could be applied (if only because of its realtive youth). The students found that Rigi dealt poorly with QUIPU and required considerable efforts to design workarounds. This suggests that tools of production quality will themselves require the adoption of stringent software engineering techniques to achieve an appropriate level of quality.

In fairness to the developers of Rigi, we should note that the students were using an early version, and that once they learned how to work around some of the incompatibilities between it and QUIPU, they were generally satisfied with it. The assistance they were provided with to learn it was minimal to non-existent. (If we were to offer a similar course again, we would attempt to arrange more formal orientation to whatever tools were used, rather than applying the sink or swim approach again.) Finally, some of the problems identified by the students have subsequently been addressed by Rigi's developers [9].

The students located several other commercial tools with similar functionality, however, they were not available at low or no cost to educational institutions. We are of the opinion that the students of today are the customers of tomorrow, and strongly urge vendors to make such allowances.

6. Acknowledgements

One of us (Katchabaw) was a member of "Team A", the group whose results are cited in this paper. The other members were Ken Dunne, Eamonn Kent and Fady Mankal.

Appeared in the Proceedings of the International Symposium on the Assessment
of Software Tools, Toronto, Canada, May 1996

Jacob Slonim, the other instructor, contributed enthusiatically to maintain a lively and thought-provoking classroom environment.

References

[1] M. A. Bauer and J. Slonim. Computer Science 434a Course Outline, September 1994.

[2] CCITT. *The Directory - Overview of Concepts, Models and Services, CCITT X.500 Series Recommendations.* CCITT, December 1988.

[3] CCITT. *The Directory - Overview of Concepts, Models and Services, Draft CCITT X.500 Series Recommendations.* CCITT, December 1991.

[4] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990.

[5] J. Farrell. Personal communication (Electronic mail message to Mike Katchabaw). Developer with ISODE Consortium.

[6] M. H. Halstead. *Elements of Software Science.* Elsevier North-Holland, New York, 1977.

[7] S. E. Kille. *Implementing X.400 and X.500: The PP and QUIPU Systems.* Artech House, Boston MA, 1991.

[8] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[9] H. Müller. Personal communication. Electronic mail message to Mike Katchabaw.

[10] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse-engineering approach to subsystem structure identification. *Software Maintenance: Research and Practice*, 5:181–204, 1993.

[11] G. C. Murphy, D. Notkin, and E. S.-C. Lan. An empirical study of static call graph extractors. Technical Report 95-08-01, University of Washington Department of Computer Science and Engineering, July 1995.

[12] C. J. Robbins and S. E. Kille. The ISO Development Environment: User's Manual Volume 5: QUIPU, July 1991.

[13] N. VanSuetendael and D. Elwell. *Software Quality Metrics.* U. S. Deparment of Transportation, Atlantic City, New Jersey, July 1991.

Appeared in the Proceedings of the International Symposium on the Assessment of Software Tools, Toronto, Canada, May 1996