USING OO METRICS AND RIGI TO EVALUATE JAVA SOFTWARE

TARJA SYSTÄ AND PING YU

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1999-9

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1999-9, JULY 1999

USING OO METRICS AND RIGI TO EVALUATE JAVA SOFTWARE

TARJA SYSTÄ AND PING YU

University of Tampere
Department of Computer Science
P.O.Box 607
FIN-33101 Tampere, Finland

ISBN 951-44-4626-7
ISSN 0783-6910

Using OO Metrics and Rigi to Evaluate Java Software

Tarja Systä
Department of Computer Science
University of Tampere
P.O. Box 607, FIN-33101 Tampere, Finland
cstasy@cs.uta.fi

Ping Yu
Department of Computer Science
University of Victoria
P.O. Box 3055, Victoria, BC, V8W 3P6, Canada
pingyu@csr.uvic.ca

July 22, 1999

Abstract

A prototype reverse engineering environment has been built to support understanding an existing Java software. The static software artifacts and their dependencies are extracted from Java byte code and viewed with Rigi reverse engineering environment as a nested graph. Several software metric values can be calculated from the byte code and analyzed with Rigi. The metric values can be used to study and structure the static dependency graph and hence support program comprehension. Rigi can be used to examine the metric values and to find software artifacts that have exceptional or extreme values.

1 Introduction

Software maintenance, re-engineering, and reuse are complex and expensive because of program comprehension difficulties. Thus, the need for software engineering methods and tools that facilitate program understanding is compelling. Reverse engineering tools provide support for analyzing software systems so that the software is more understandable. *Software metrics* play a significant role in a reverse engineering process. They are used to make numerical measurements of particular aspects of a target software. Metrics can be applied to support the identification of complex parts of the software that need restructuring. They can also reveal tightly coupled parts of the software. Such parts are inflexible for modifications and reuse. They also may represent potential subsystems. Identification of subsystems, in turn, supports program comprehension.

The usage of software metrics is not limited to reverse engineering. In fact, they have traditionally been used in forward engineering to improve the quality of the software. For example, software metrics can be used to measure the complexity of the software design and to predict properties of the final product. They can also be used to predict the amount of testing necessary or the total development costs [9].

In this paper software metrics are used to support reverse engineering of Java software. A set of selected metrics is used to estimate inheritance relationships, complexity, and communication of a target Java software. A metrics program is integrated with a prototype reverse engineering environment used to analyze Java software. The environment supports both *static* and *dynamic reverse engineering*. Static reverse engineering aims modeling the structure of a target software while dynamic reverse engineering intends modeling its run-time behavior. The dynamic event trace information is generated automatically as a result of running the target system under a debugger. SCED[KMST98] is used to view the event trace as scenario diagrams. The static information is extracted from the byte code and analyzed with Rigi reverse engineering environment[MWT94]. Rigi uses a nested graph to view the static dependencies. The metrics program calculates values for selected metrics from the information extracted by the byte code parser. The metric values calculated can be dumped into a file or added to a Rigi graph and used for analyzing the software. In Rigi Tcl/Tk [12] scripts can be run on the static dependency graph [16]. The scripts can be used to make queries about the graph or to modify it. The scripts provide a flexible way to analyze the metric values added to the Rigi graph. For example, by running a script the user can easily focus on parts of the software that have metric values in a desired value range. Because Tcl is an interpretable scripting language, the script library of Rigi can be easily extended; new scripts can be written and added to it dynamically. This allows the user to write and use scripts that have specific tasks, e.g., scripts that support the analysis of the metric values.

A target system has been analyzed using the metrics program and Rigi. The selected target system, FUJABA [14], is freely available software, developed in the University of Paderborn, Germany. The primary topic of the FUJABA project and environment is Round Trip Engineering with UML, SDM (Story Driven Modeling), Java and Design Patterns. FUJABA provides editors for defining both structural (class diagrams) and behavioral (activity diagrams, UML activity/story diagrams) aspects of a software. Furthermore, the Java source code can be generated from the design, which then can be compiled. FUJABA environment also supports animation of the designed system through the constructed models. FUBAJA is written in Java, containing almost 700 classes. The FUJABA version under examination was 0.6.3-0. The focus during the reverse engineering process was on studying the structure and complexity of *de.uni_paderborn.fujaba.uml* package of FUJABA. This package provides classes that implement UML [13, 15] modeling concepts for a class diagram and activity diagram editors of FUJABA. The package contains altogether 56 classes or interfaces.

2 Software metrics

Software metrics are often categorized into *product metrics* and *process metrics*. Product metrics relate essentially to product size, which can be measured either in terms of its structure or its modularity. Process metrics measure effort as a function of time [4]. Product metrics are used to control the quality of the software product, while process metrics are applied to measure the status and progress of the system design process. Process metrics can also be used to predict future effects or problems. In this paper the focus is on product metrics.

Product metrics can be categorized further in various ways. They can, for example, be divided into *static metrics* and *dynamic metrics*. Dynamic metrics have a time dimension and the values tend to change over time. Thus dynamic metrics can only be calculated on the software as it is executing. Static metrics remain invariant and are usually calculated from the source code, design, or specification.

Some of the product metrics can be used to measure software written in any language, while some of them can measure a specific kind of software only. *OO metrics* [8, 4] are used to evaluate object-oriented software. Some of the

OO metrics are variations of traditional product metrics used to measure software written in procedural languages. There are also OO metrics that cannot be applied for procedural languages, e.g., metrics that measure the inheritance hierarchy.

When applying software metrics it should be remembered that metric values calculated should be used as guidelines, not rules. They help the engineer to recognize parts of the software that might need modifications and reimplementation. The decision of changes to be made should not rely only on the metric values.

3 The metrics suite selected

The metrics suite selected contains seven OO metrics. The metrics are categorized into *inheritance metrics*, *complexity metrics*, and *communication metrics*. The inheritance metrics measure the inheritance and implementation hierarchy of the software. Complexity metrics estimate the complexity of it and communication metrics measure coupling and cohesion between classes. Compared to the categorization of object-oriented metrics used in [4], inheritance and complexity metrics falls into a category of *module metrics* used to measure procedural complexity of the software, and communication metrics responds to *intermodule metrics* that measure system design complexity.

3.1 Inheritance metrics

Inheritance metrics measure the inheritance hierarchy of object-oriented software. When calculating the values for inheritance metrics both classes and interfaces are taken into account. Even though interfaces cannot have actual method bodies, they can have variables. Thus, including interfaces to the calculations can be argued. For example, information about the number of classes that implement an interface, i.e., classes that have access to its variables, might be an important piece of imformation.

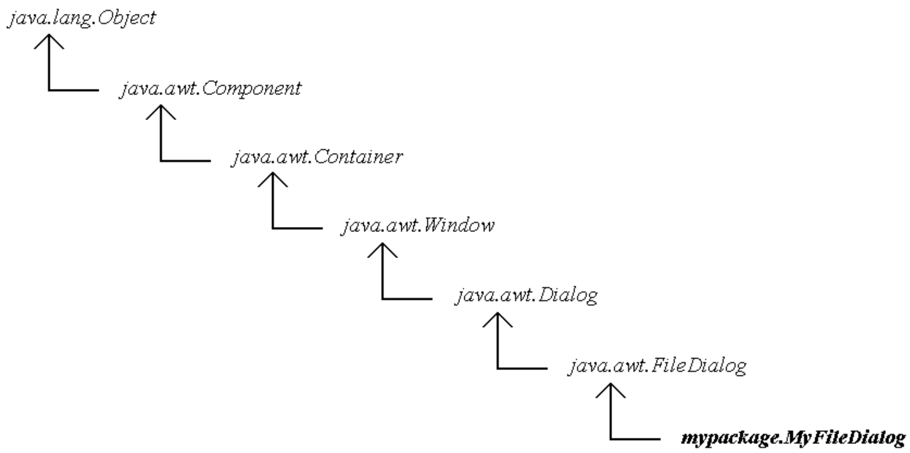

Figure 1: An example Java inheritance hierarchy.

In Java all the classes eventually inherit the root class *java.lang.Object*. The part of the inheritance hierarchy that does not belong to the target system can and in many cases is longer than the part belonging to the target system. For example, assume that the designer wants to implement her own file dialog class for selecting files using jdk's *java.awt.FileDialog* class. The inheritance hierarchy in that case is shown in Figure 1. In this research the system classes are ignored when calculating inheritance metrics. The system classes represent a stable part of Java software and hence does not vary from application to application. Hence, including them to the metrics analysis does not give any additional information from the software, but it does make the calculations more complicated and fades the application border. The primary purpose of software metrics is to define some quality attributes for the software that can then be used to point out the need of changes and reimplementation. Including the system classes to the calculations is then unnecessary since the user has usually no access to the implementation of system classes.

Two metrics are used to evaluate the inheritance/implementation hierarchy: Dept of Inheritance Tree (DIT) and Number of Children (NOC). Both of the metrics are introduced in [1].

3.1.1 Depth of Inheritance Tree (DIT)

DIT is a length from the class node to the root of the inheritance tree and is measured by the number of ancestor classes. The deeper a class is within the class hierarchy, the greater number of methods and variables it is likely to inherit. Such classes are typically more complex and their behavior is difficult to predict. Deeper inheritance trees constitute greater design complexity since more methods and classes are involved. On the other hand, they provide more possibilities for potential reuse. DIT primarily evaluates reuse but also relates to understandability and testability.

DIT is generally calculated either as a maximum or as an average of lengths of different paths to the root of the inheritance tree. When applied to languages that support multiple inheritance, e.g., C++, these two approaches can give very different results and also reveal different aspects of the inheritance hierarchy. In this research the latter approach is used. Java language does not support multiple inheritance, however, a class can implement several interfaces in addition to extending one super class. Interfaces are basically pure abstract classes that can have variables usable by implementing classes. Since classes/interfaces belonging to jdk are ignored, the root class/interface is considered to be the one that does not extend/implement any other class/interface belonging to the target system.

3.1.2 Number of Children (NOC)

NOC is the number of immediate subclasses subordinated to a class in the class hierarchy. NOC measures reusability of a class and gives an idea of the potential influence that a class has on the design.

For a class, NOC is a number of classes that extend the class. For an interface, NOC is a number of interfaces that extend the interface, added with a number of classes that implement it.

3.2 Communication metrics

Communication metrics are used to estimate and measure the internal and external communication of modules. It is commonly accepted that low coupling and high cohesion in a software design lead to better products, e.g., in terms of reliability and maintainability. This principle is also used in static reverse engineering tools, for instance, in Rigi, for subsystem composition:

a part of the software that has a lot of interactions between its elements but only few interactions with elements outside the part is a potential subsystem candidate.

In object-oriented systems the importance of coupling seems to be emphasized. First, the tighter the coupling of client objects to a server object, the harder the effects on the clients whenever a crucial aspect of the server is changed. Second, high coupling between two objects makes it harder to understand one of them in isolation. Third, high coupling increases the probability of remote effects, where error in one object cause erroneous behaviour of other objects. [5]

In this research three metrics are used to measure coupling and cohesion between classes or objects. These metrics are: Response For a Class (RFC), Coupling Between Objects (CBO), and Lack of Cohesion in Methods (LCOM). Chidamber and Kemerer have introduced all of them in [1]. For LCOM we have adopted a definition introduced in [4].

3.2.1 Response For a Class (RFC)

RFC metric looks at the combination of the complexity of a class through the number of methods and the amount of communication with other classes. The larger number of methods that can be invoked from a class in respond to messages, the greater the complexity of the class and the more complicated testing and debugging becomes.

When calculating RFC calls between methods, constructors, and static blocks are taken into account. The formula used is defined next. For a class C, let M_i be a set of all methods, constructors and static blocks in C. Let M_o be a set of methods, constructors, and static blocks belonging to any other classes that are called by the members of M_i. Then RFC for class C is calculated as the size of a set $M_i \cup M_o$.

By the definition of RFC all couples with external methods are of equal strength. However, calling a method of a super class cannot be seen as harmful as calling methods of other classes. For example, the default constructor of the super class is called automatically from the constructors of subclasses in Java. Furthermore, overloading a method in a subclass typically contains a call to the overloaded method of the super class. Next we discuss CBO metric that distinguishes these cases.

3.2.2 Coupling Between Objects (CBO)

CBO measures coupling between classes that are not related through inheritance. Class A is coupled to class B if methods of A use methods or variables of B. A class that is very coupled contradicts encapsulation and prevents reuse. The more independent a class is, the easier it is to be reused. In order to promote encapsulation, inter-class couples should be kept to a minimum. A software with a large number of couples becomes sensitive to changes and therefore difficult to be maintained.

For calculating CBO, both constructors and methods are taken into account. Following relationships between two classes that are not in a super class — subclass relationship are considered to cause coupling: method calls, constructor calls, instance variable assignments, or other kind of instance variable accesses (usage). Thus, static blocks need not to be examined.

CBO also has its weaknesses as a measure of object coupling [5]. Direct access to foreign instance variable has generally been identified as the worst type of coupling. However, CBO assumes that all the couples are of equal strength and hence does not distinguish that from, e.g., a method call. Moreover, calling methods of an object is generally considered less harmful than calling methods of one of its components, and so on.

3.2.3 Lack of Cohesion in Methods (LCOM)

The methods of a class should be logically related. If a class exhibits low method cohesion it indicates that the design of the class has probably been partitioned incorrectly. In that case the design could be improved if the class was split into more classes with individually higher cohesion. LCOM helps to identify such flaws in the design.

The formula for calculating LCOM is presented next. Consider a class C with methods $M_1, \ldots, M_n$. Let I_i be a set of instance variables used by method $M_i, i = 0 \ldots n$. For n methods, there are n such sets: $I_1, \ldots, I_n$. LCOM metric is given by Chidamber and Kemerer in [1] as "the number of disjoint sets formed by the intersection of the n sets". This formula would lead to very small or empty single set if all the intersections are taken into account. Because of the obvious weakness of the formula, several interpretations of it has been discussed in the literature [7, 3]. A revised version of the formula is presented in [2]. Altogether new formulas for calculating LCOM have also been introduced, e.g., in [4]:

$$LCOM* = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right) - m}{1-m},$$

where $M_i, i = 1, \ldots, m$ is a set of methods, $A_j, j = 1, \ldots, a$ is a set of attributes, and $\mu(A_j)$ is the number of methods which access attribute A_j.

In this research the formula presented above and introduced in [4] is used.

3.3 Complexity metrics

Metrics introduced next are used to measure complexity of software. A lot of metrics fall into this category, e.g., metrics that measure size or logical structure of the software. One of the most commonly used complexity metric is Lines Of Code (LOC), which simply measures the number of lines in a method, module, or class. Even though LOC is widely used, partly because it is easy to calculate, it has been criticized for being too simple and vague measure. For example, consider a constructor of a dialog class that is responsible for initializing several GUI components belonging to that dialog. Such a constructor has typically high LOC value but a very simple structure; it contains several constructor invocations but has only a few (if any) repetition or conditional constructs. Should such a constructor be considered to be complex ? LOC has not been included in the available set of metrics in this research since the information needed for it can not be concluded from Java class files.

3.3.1 Cyclomatic Complexity (CC)

CC measures the logical structure of the software. It is used by several other metrics. CC is calculated using the following formula:

$$V(G) = e - n + 2 * p, \tag{1}$$

where G is a complexity graph, e is the number of edges in G, n is the number of nodes in G, and p is the number of disconnected components in G. The complexity graph G for a single method is a control flow graph.

The used formula for calculating CC (1) is adopted from [4] The original McCabe's cyclomatic complexity metric [10] was based on the control flow graph as well. The above formula (1) is a variation of that formula, introduced as:

$$V(G) = e - n + 2. \tag{2}$$

3.3.2 Weighted Methods per Class (WMC)

WMC is a sum of complexities of methods of a class. Hence it measures the size as well as the logical structure of the software. The number of methods and the complexity of the involved methods are predictors of how much time and effort is required to develop and maintain the class. The larger the number of methods in a class, the greater the potential impact on inheriting classes. Furthermore, classes with large number of complex methods are likely to be more application specific, limiting the possibility of reuse. Thus WMC can also be used to estimate the usability and reusability of the class.

In this research WMC is calculated as a sum of the methods, weighted by their static complexity. The static complexity is calculated using CC metric.

4 Normalization of metric values

In Section 3 some complexity, communication, and inheritance metrics were introduced. The complexity of the software can be measured in various ways. There are several other metrics that can be used in addition to those discussed in Section 3. A metrics suite typically includes several metrics. It might, for example, include various complexity metrics. If the values of those complexity metrics correlate when applied to a certain part of the software, then the user can be more convinced about the overall complexity of that part. However, most of the metrics have different ranges of values, which makes it difficult to compare the metrics.

For the convenience of comparisons and analysis the generated values for complexity and communication metrics can be normalized in Rigi by running a script. Each value is normalized by subtracting the mean form it and dividing the result by the standard deviation. The changed values will then have zero mean and unit deviation. The scripts used are relatively presented next. All the commands beginning with *rcl_* and used in Algorithm 2 belong to the script library of Rigi.

Algorithm 1. Normalizing all communication and complexity metric values of all the nodes of type *Class* in a Rigi graph.
Method:

```
proc java_norm_metrics { } {
    java_norm_metric WMC
    java_norm_metric CC
    java_norm_metric LCOM
    java_norm_metric RFC
    java_norm_metric CBO

}
```

Algorithm 2. Normalizing the values of a given metric of all the nodes of type *Class* in a Rigi graph.
Input: The metric, values for which will be normalized.
Method:

```
proc java_norm_metric { metric } {
    rcl_select_none
    rcl_select_type Class
    set winnodes [rcl_select_get_list]
    rcl_select_none
    set sum1 0
    # calculate the mean value
    foreach nodeid $ winnodes {
        set val [rcl_get_node_attr $ nodeid $ metric]
        if { $ val > 0 } {
            set sum1 [expr $ sum1 + $ val]
        } else {
            rcl_set_node_attr $ nodeid $ metric 0
        }
    }
    set mean [expr [expr $ sum1 * 1.0] / [llength $ winnodes ]]
    # calculate the deviation
    set sum2 0
    foreach nodeid $ winnodes {
```

```
        set val [expr [rcl_get_node_attr $ nodeid $ metric] - $ mean]
        set val [expr $ val * $ val]
        set sum2 [expr $ sum2 + $ val]
    }
    set sum2 [expr $ sum2 / [llength $ winnodes]]
    set med [expr sqrt($ sum2)]
    foreach nodeid $ winnodes {
        set val [expr [rcl_get_node_attr $ nodeid $ metric] - $ mean]
        rcl_set_node_attr $ nodeid $ metric [expr [expr $ val * 1.0] / $
        med]
    }
}
```

5 Calculating and using the metric values

The prototype reverse engineering environment can be used for calculating values for the metrics discussed in Section 3. The user can select any subset of the seven metrics provided to be included in the metrics suite and generate values using a menu command. The values will be calculated for classes and interfaces that are known at that point, i.e., for classes and interfaces for which the static information has been generated by the byte code parser. Values of some of the metrics will also be calculated for methods, constructors, and static blocks of the classes.

Rigi uses a nested graph to view the static dependencies. Each node in the graph represents a software artifact. For Java such artifacts are classes, interfaces, methods, constructors, static blocks, and variables. The calculated metric values are added as attribute values for Rigi nodes. The attribute values of nodes are not visible in Rigi but they can be used for analyzing the graph. They can, however, be examined using the graph editor by selecting a node and opening a pop-up dialog for it. This is shown in a snapshot of a Rigi session in Figure 4: the user has selected node *de.uni_paderborn.dis.DisRow* and opened a dialog that enumerates all the attribute values of the node.

Using metric values as node attributes in Rigi provides a flexible and powerful way to analyze the values and the static dependencies. The metric values help understanding the static dependencies. For example, they can be used for finding highly cohesive and weekly coupled parts of the software. On the other hand, the scripts of Rigi help studying the metric values by providing a way to make queries. Like discussed in Section 4, the metric values can be normalized by running a single script.

6 Threshold values

Product metrics are language and programming style dependent. Language dependent threshold values for the metrics are presented in literature to give heuristic ranges of better and worse values. They are usually based on experiences over several software projects and hence should be treated as heuristics and recommendations. In [8] threshold values for several OO metrics are given for C++ and Smalltalk, based on experience in various C++ and Smalltalk software projects. There is much less experience on Java software projects and thus there is much less reserch on threshold values for Java software. When compared to C++ and Smalltalk it can be assumed that usable threshold values for Java are closer to those for Smalltalk than those for C++. This assumption can be reasoned several ways:

1. C++ is a hybrid language, while Smalltalk and Java are pure OO languages. Nearly all C++ programs are, in fact, mixtures of C and C++. Usually, all the functionality and code is not captured inside classes like in Smalltalk and Java.

2. Smalltalk and Java are simpler languages than C++, mostly because C++ is a hybrid language. In Smalltalk and Java, there is usually one way to implement a certain primary task, while in C++ there are usually several ways.

3. In both Smalltalk and Java all the classes are finally subclasses of a certain, single root class (class *java.lang.Object* in Java). This makes the inheritance hierarchies potentially similar.

4. The usage of interfaces in Java resembles usage of pure abstract classes in Smalltalk, again making the inheritance hierarchies potentially similar. In C++ multiple inheritance is supported.

In this research threshold values are not given or used for estimating acceptable metrics value ranges for Java software. However, the user has a possibility to try some threshold values using Rigi. This can be done by running *java_select_attributes_thresh* script. The script takes three arguments *type*, *metric*, and *threshold*. Argument *type* defines a type of nodes in the Rigi graph. Accepted types are a class, an interface, a method, a constructor, and a static block. Argument *metric* defines the OO metric to be examined. A threshold value representing a limit value is given by the last argument *threshold*. The script selects all nodes of type *type* in Rigi graph that have higher value than *threshold* of metric *metric*. By running this script the user can quickly find software artifacts that have critical or extreme metric values, e.g., classes that are most complex. The implementation of *java_select_attributes_thresh* is presented by Algorithm 3. Like in Algorihtm 2 the commands beginning with *rcl_* belong to the script library of Rigi.

Algorithm 1. Selects all nodes of a given type in Rigi graph for which the value of a given metric is higher than a given threshold value.
Input: The first argument *type* defines a type of nodes in the Rigi graph to be taken into account. The second argument *metric* defines the OO metric to be examined. The third argument *threshold* gives a limit value for the metric values.
Method:

```
proc java_select_attributes_thresh { type metric threshold } {
    rcl_select_none
    rcl_select_type $ type
    set winnodes [rcl_select_get_list]
    rcl_select_none
    foreach n $ winnodes {
        set val [rcl_get_node_attr $ n $ metric]
        if {$ val > $ threshold} {
            rcl_select_id $ n 1
        }
    }
}
```

7 An example: calculating metric values for FUJABA software

Metric values calculated for FUJABA software are studied next. Figure 2 shows a Rigi graph representing the whole FUJABA software. The information has been generated using a Java byte code extractor of the prototype reverse engineering environment. From the left down corner it can be noticed that the whole software consists of 25854 software artifacts (classes, interfaces, class/interface members etc.).

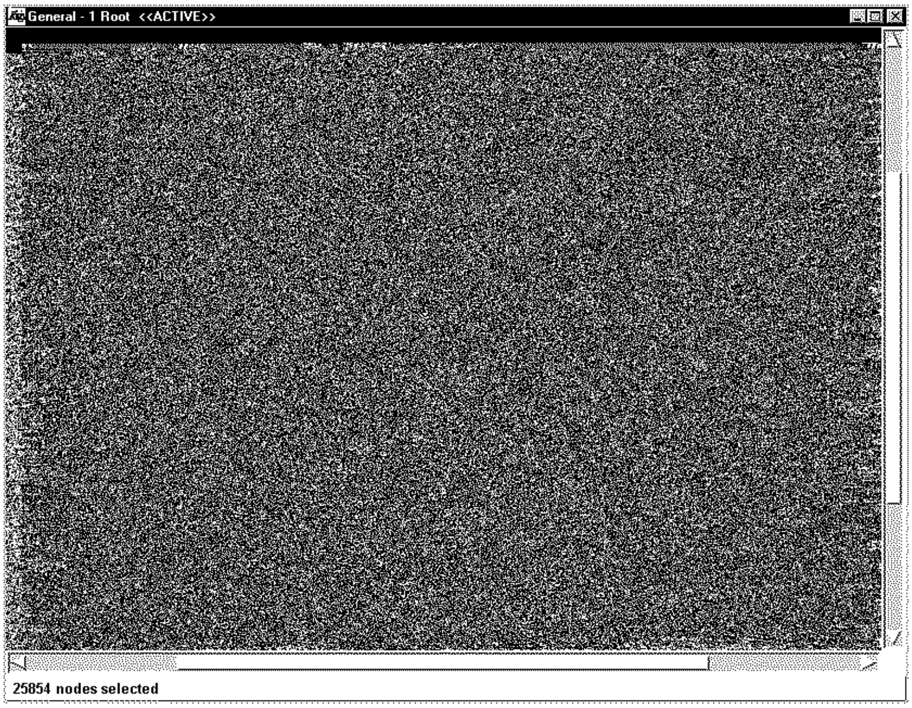

Figure 2: The initial Rigi graph representing the whole FUJABA software.

Values of all seven metrics have been calculated and added to static dependency graph of Rigi. Some scripts has been run on the Rigi graph to study the extreme values of some of the metrics. Figure 3 shows methods nodes that have highest CC values. The nodes can be easily found by running the *java_select_attributes_thresh* script (see Section 6). The rest of the graph has been filtered out.

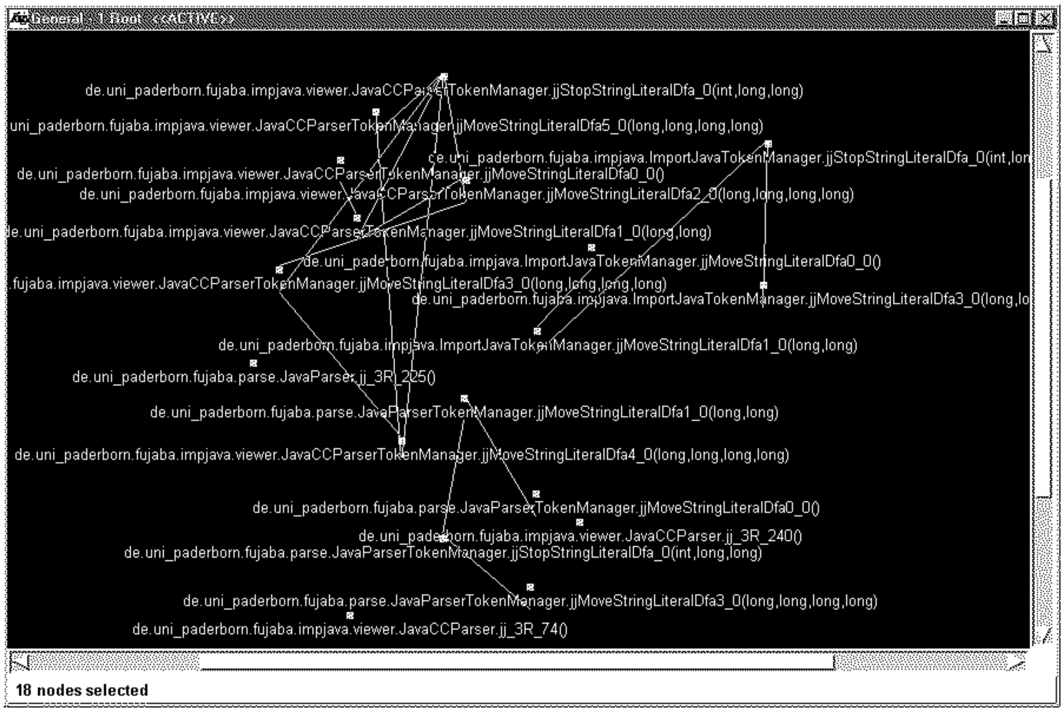

Figure 3: Method nodes that have the highest CC values and represent methods in FUJABA software.

First we examine values of inheritance metrics DIT and NOC. Again by running the *java_select_attributes_thresh* script the classes with extreme DIT values can be easily found. Figure 4 shows class nodes with highest DIT values. It results when a command *java_select_attributes_thresh Class DIT 4* is executed on the initial graph 2 and the rest of the graph is filtered out. The user has selected a node *de.uni_paderborn.fujaba.dis.DisRow* and opened a pop-up dialog showing all the attribute values of that node. It can be seen that DIT value for that class is 5 and NOC value is 0. There are only 6 classes, for which the average length of the inheritance hierarchy is more than 4. This implies that the inheritance hierarchy of FUJABA software is not very deep. The same way it can be found out that there are 8 classes for which the NOC value is greater than 5. This in turn implies that the inheritance hierarchy is not very flat either. Hence, it can be concluded that inheritance is not heavily used in FUJABA. This, in turn, could be a motivation for the designer to take a closer look at the inheritance hierarchy to examine if restructuring would improve the design. From the low DIT and NOC values it can also be guessed that the software is not built as an extensible object-oriented framework.

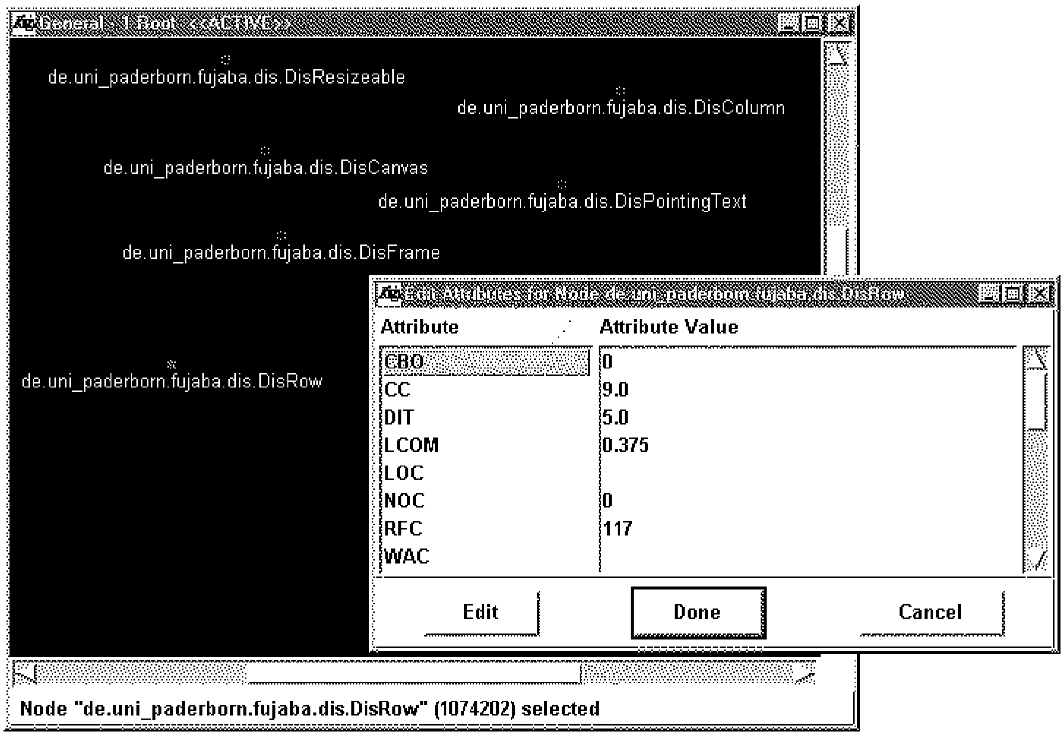

Figure 4: A Rigi graph showing all the class nodes in FUJABA that have a DIT value greater than four.

Next we analyze communication metrics RFC, CBO, and LCOM. By running the *java_select_attributes_thresh* script it can be noticed that most coupled classes in respect of RFC metric belong to *de.uni_paderborn.fujaba.uml* package: from the top 25 classes with the highest RFC values as many as 20 belongs to that package. Similar ratios for CBO and LCOM are 13/39 and 11/119, respectively. Package *de.uni_paderborn.fujaba.uml* might hence contain classes that interact a lot with other classes. Another explanation could be the fact that *de.uni_paderborn.fujaba.uml* is one of the largest packages in FUJABA. The high coupling values encouraged us to take a closer look at the metrics values generated for that package. The classes with highest RFC, CBO, and LCOM values are listed in Table 1 in a decreasing order of their original metric values. Some of the classes listed in the table are inner classes. The full name of an inner class consists of the name of the owner class separated with a character "$ " from the name of the inner class itself.

RFC	CBO	LCOM
UMLClass	TestProject	UMLFile$ UMLPackageComparator
UMLProject	UMLClass	UMLStoryPattern$ collabStatLessThan
TestProject	UMLActivity	UMLLink
UMLFile	UMLActivityDiagram	UMLTransitionGuard
UMLClassDiagram	UMLMethod	UMLIncrement
UMLTypeList	UMLObject	UMLLinkSet
UMLStroyPattern	UMLStoryActivity	UMLClass

Table 1: This table shows those FUJABA classes that belong to *de.uni_paderborn.fujaba.uml* package and have highest communication metric values. Classes are listed in a decreasing order of their metric values.

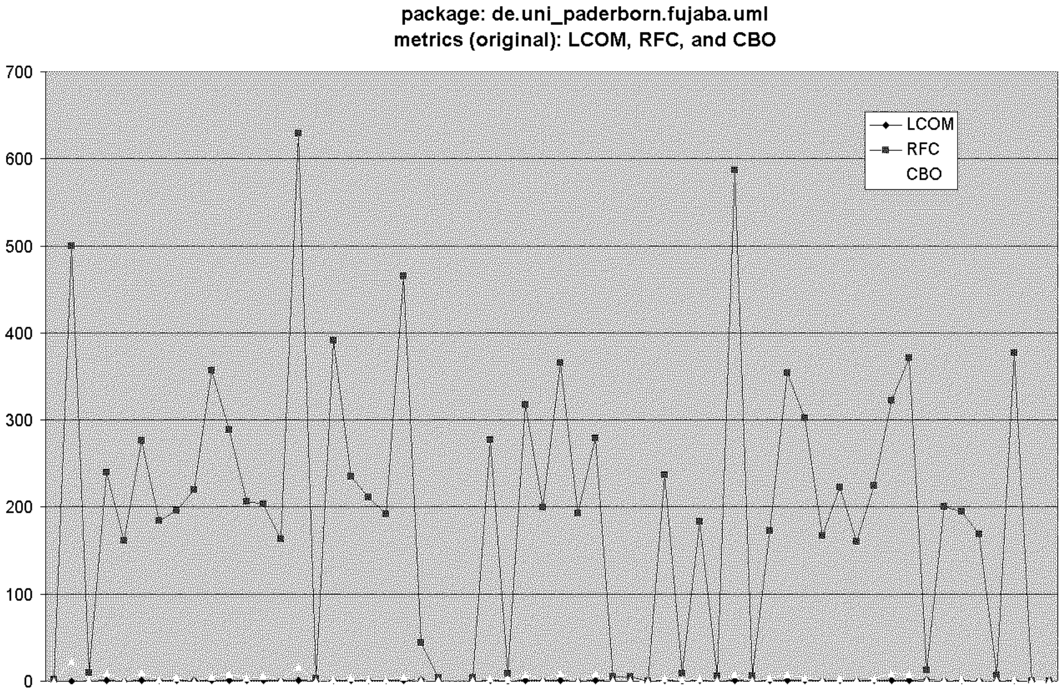

Figure 5: The original values of RFC, CBO, and LCOM metrics for classes in *de.uni_paderborn.fujaba.uml* package.

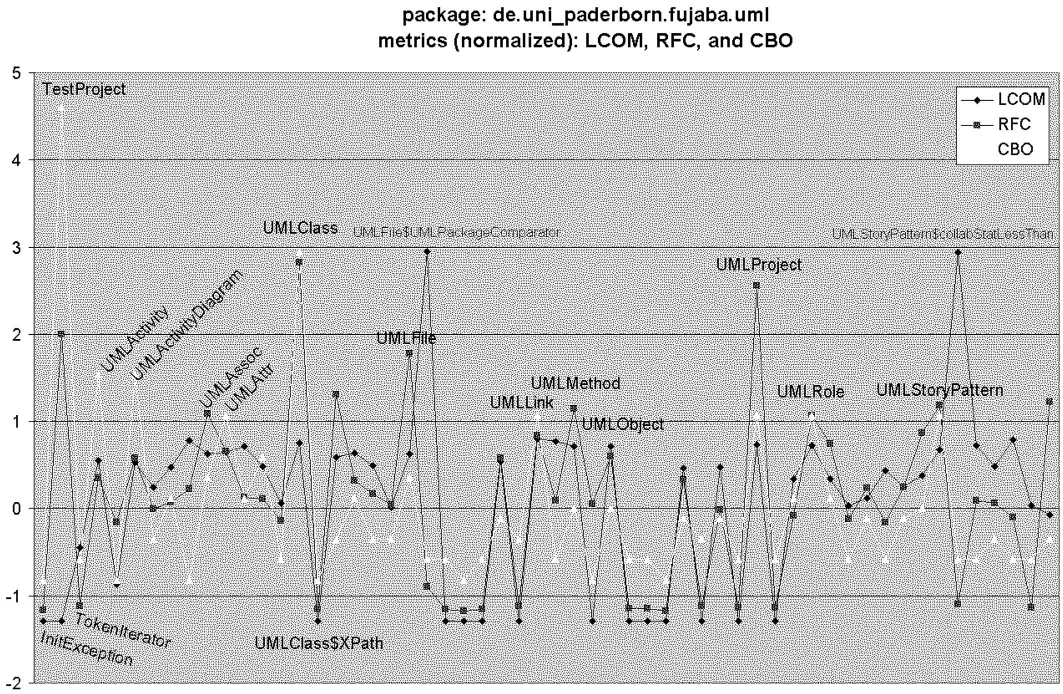

Figure 6: The normalized values of RFC, CBO, and LCOM metrics for classes in *de.uni_paderborn.fujaba.uml* package.

Figure 5 shows a line diagram of the original communication metric values for package *de.uni_paderborn.fujaba.uml*. From the diagram it is difficult to conclude whether the values of different metrics correlate. Figure 6, in turn, shows the normalized values. From the diagram it is easy to see that there is a strong correlation between the metrics. Especially, the curve describing CBO values is more descriptive and the correlation with the RFC and LCOM curves is easier to recognize than in Figure 5.

Next we study the complexity of *de.uni_paderborn.fujaba.uml* package. LCOM metrics can be regarded to measure both communication and complexity. For comparisons we thus examine LCOM together with CC and WMC. Figure 7 shows a line diagram of the complexity metric values for all classes in *de.uni_paderborn.fujaba.uml* package.

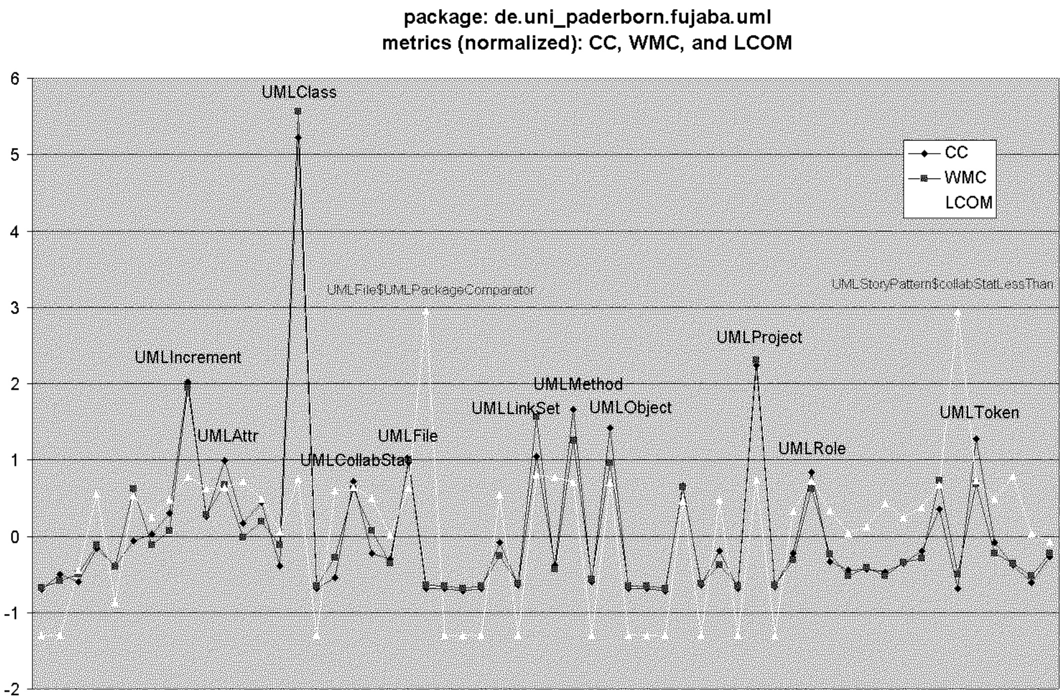

Figure 7: The normalized values of CC, WMC, and LCOM metrics for classes in *de.uni_paderborn.fujaba.uml* package.

By comparing Figures 6 and 7 it can be seen that the shapes of the lines in both figures are somewhat similar, i.e., most of the classes that have have high communication metric values also have high complexity metric values. The most obvious exception is class *TestProject*, for which RFC and CBO values are very high but CC, WMC, and LCOM values are low. Such classes typically consists of methods that mostly call and/or are called by other classes and do not implement complicated algorithms or code structures. This is the case also with *TestProject* class. From Figures 6 and 7 also show that class *UMLClass* has high values of both communication and complexity metrics. When examining the size of *UMLClass.class* and *UMLClass.java* files it can be noticed that *UMLClass* class is clearly the largest class in package *de.uni_paderborn.fujaba.uml*. The size of the *UMLClass.java* file, for example, is more that twice the size of the second biggest class *UMLIncrement*. Hence, there is no reason to suspect a flaw in the design.

8 Discussion

An approach to calculate and examine metric values has been discussed. Combining the information about software metrics with a graphical reverse engineering tool helps both the reverse engineering process and the analysis of the metric values. For example, in reverse engineering one of the most challenging tasks is building abstract views from the parsed static dependencies. This can be done by building high level components that represent software artifacts being highly cohesive and loosely coupled with other parts of the software. Metrics that measure communication between classes/objects can be used to support this task. Going the other way round, a reverse engineering tool can be used to find software artifacts that have extreme or exceptional metric values. Such values need to be recognized in order the make suggestions for possible restructuring of the software.

In this research Rigi reverse engineering tool is used to visualize the static dependencies in the target software as a nested graph. The information is extracted from the Java byte code. The calculated metric values are added to the graph. Rigi provides an extensible script library that can be used for making queries on the graph to modify it. In this paper examples of using scripts for finding extreme metric values are presented. Furthermore, the metric values can be normalized by running a script. The normalization is needed in order to be able to conclude whether different metrics correlate or not.

The metric values are calculated by running a metrics program integrated with the prototype reverse engineering environment. Some of the metrics could also be calculated using Rigi. If the information needed is included in the static dependency graph, a new script that calculates the values and adds that information to graph could be written and added to the script library of Rigi (dynamically, if desired). However, this is not possible for all the metrics. For example, CC is calculated using the control flow information that is not included in Rigi graph but is generated by the byte code extractor.

When studying the metric values calculated for FUJABA software no big flaws in the design could be suspected. By examining the communication metrics RFC, CBO, and LCOM, design flaws in structuring classes and in information hiding issues might be recognized. If LCOM value for a class is high but RFC and CBO values are low, then it can be suspected that the class might have unused variables or the variables are not properly selected for the class. By examining the complexity metrics CC and WMC complex data structures could be recognized. The inheritance metrics NOC and DIT can be used to study the inheritance hierarchy and hence help estimating the reusability and extensibility of the software.

It should be remembered that metrics should not be used as design rules. They are hints that might reveal parts of the software that needs to be examined more detailed in order to find design flaws. Also, the metric values depend on the type and functionality of the software. For example, GUI classes differ from classes that implement algorithms in terms of inheritance, communication, and complexity. Hence different metric values should be "allowed" for them.

References

[1] S. R. Chidamber and C. F. Kemerer, "Towards a Metrics Suite for obect-oriented design", In Proc. of the *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*, ACM, 1991, pp. 197–211.

[2] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object- Oriented Design", *IEEE Trans. Softw. Eng.*, **20**, 6, 1994, pp. 476-493.

[3] I. M. Graham, "Migrating to Object Technology", Addison- Wesley, 1995.

[4] B. Henderson-Sellers, *Object-Oriented Metrics, Measures of Complexity*, Prentice Hall, 1995.

[5] M. Hinz and B. Montazeri, "Measuring coupling and cohesion in object-oriented systems", In Proc. of *International Symposium on Applied Corporate Computing (ISAA'95)*, Oct. 1995.

[6] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi, "Automated Support for Modeling OO Software", *IEEE Software*, **15**, 1, January/February 1998, pp. 87–94.

[7] W. Li and S. Henry, "Object-oriented metrics that predict maintainability", *J. Sys. Softw.*, **23**, 1993, pp. 111–122.

[8] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics, A Practical Guide*, Prentice Hall, 1994.

[9] T. DeMarco, *Controlling Software Projects*, Yourdon Press, 1982.

[10] T.J. McCabe, "A complexity measure", *IEEE Trans. Software Eng.*, **2**, 4, pp. 308-320, 1976.

[11] H. Müller, K. Wong, and S. Tilley, "Understanding software systems using reverse engineering technology", In *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)*, 1994.

[12] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[13] Rational Software Corporation, *The Unified Modeling Language Notation Guide v1.3*, [http://www.rational.com], 1999.

[14] I. Rockel and F. Heimes, *FUJABA - Homepage* [http://www.uni-paderborn.de/fachbereich/AG/schaefer/ag_dt/PG/Fujaba/fujaba.html], February, 1998.

[15] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[16] K. Wong, *Rigi User's Manual Version 5.4.1* [http://www.rigi.csc.uvic.ca/rigi/manual/user.html], September, 1997.