Analyzing Java Software by Combining Metrics and Program Visualization

Tarja Systä
Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
tsysta@cs.tut.fi

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

Hausi Müller
Department of Computer Science
University of Victoria
P.O. Box 3055, Victoria, BC, V8W 3P6, Canada
hausi@csr.uvic.ca

Abstract

Shimba, a prototype reverse engineering environment, has been built to support the understanding of Java software. Shimba uses Rigi and SCED to analyze, visualize, and explore the static and dynamic aspects, respectively, of the subject system. The static software artifacts and their dependencies are extracted from Java byte code and viewed as directed graphs using the Rigi reverse engineering environment. The static dependency graphs of a subject system can be annotated with attributes, such as software quality measures, and then be analyzed and visualized using scripts through the end-user programmable interface.

Shimba has recently been extended with the Chidamber and Kemerer suite of object-oriented metrics. The metrics measure properties of the classes, the inheritance hierarchy, and the interaction among classes of a subject system. Since Shimba is primarily intended for the analysis and exploration of Java software, the metrics have been tailored to measure properties of software components written in Java. We show how these metrics can be applied in the context of understanding software systems using a reverse engineering environment. The static dependency graphs of the system under investigation are decorated with measures obtained by applying the object-oriented metrics to selected software components. Shimba provides tools to examine these measures, to find software artifacts that have values that are in a given range, and to detect correlations among different measures. The object-oriented analysis of the subject Java system can be investigated further by exporting the measures to a spreadsheet.

1. Introduction

Software maintenance, re-engineering, and reuse involving large software systems is complex, costly, and risky mainly because of the difficult and time-consuming task of program comprehension. Many reverse engineering tools have been built over the last fifteen years to help the comprehension of large software systems. These tools aid the extraction of software artifacts and their dependencies and the synthesis of high-level concepts. Moreover, these tools provide support for analyzing software systems and automate some of the mundane and repetitive understanding operations.

With the advent of object-oriented programming languages, such as Smalltalk, C++, and Java, object-oriented design and development methods have been widely adopted in the software industry. While popular, object-oriented programming is not a panacea. The need for assessing the quality of software systems has not changed. Just as object-oriented programming requires a different approach, compared to imperative programming, software metrics for object-oriented programs must differ from traditional software metrics. There are many object-oriented metrics, but the key metrics aim to measure design and code quality by investigating the coupling among classes, the cohesion within classes, complexity of classes, and complexity of the inheritance hierarchy.

These metrics can play a significant role when reverse engineering an existing software system. One approach to use object-oriented complexity metrics to identify high- and low-complexity parts of the subject system. The most experienced maintainers or reengineers can then be assigned to the most complex subsystems [22]. Another strategy is to

identify complex or tightly coupled parts in the subject software system. Such parts are difficult to modify and reuse and might be candidates for restructuring, refactoring, or significant redesign [6]. Metrics can also be used to identify highly cohesive and loosely coupled parts of the software that potentially represent subsystems [13]. Hierarchies of subsystems form the organizational axes for software exploration and, in turn, program comprehension.

In this paper we consider the Chidamber and Kemerer suite of object-oriented metrics to aid the reverse engineering and understanding of Java software [1, 2]. The metrics measure properties of the classes, the inheritance hierarchy, and the interaction among subsystems. The metrics were implemented in Shimba, a prototype reverse engineering environment, to analyze Java software [21].

Shimba supports the exploration, visualization, and analysis of both *static* and *dynamic reverse engineering*. Static reverse engineering aims to model the structure of a subject software system while dynamic reverse engineering intends to model its run-time behavior. Shimba integrates the reverse engineering environment Rigi, the dynamic analysis engine SCED, and an extensible suite of object-oriented metrics. In this paper, we only consider static reverse engineering. The static information is extracted from the byte code of the subject system and analyzed with Rigi [14]. Rigi uses a graph model to represent information about software entities, relationships, attributes, and the abstractions over them.

The collection of object-oriented metrics operates directly on Java byte code. The metrics suite can be applied to an entire Java program or subsets, such as individual packages and classes. The resulting measures can be attached to the Rigi graphs of the subject system or simply be stored in files.

Rigi is end-user programmable [23] through its built-in Rigi Command Language (RCL) [25] which is based on Tcl/Tk [15]. Queries and analyses can be encoded in RCL scripts to operate on the subject system's static dependency graph and the annotated object-oriented complexity measures. Thus, the scripts are a flexible and versatile tool to explore and investigate the measures. For example, the script depicted in Figure 1 allows the reverse engineer to identify those parts of a subject software system that have a given metric in a desired value range. The script takes four arguments *type*, *metric*, *lowerbound*, and *upperbound*. The argument *type* defines a node type in a Rigi graph. Accepted types correspond to the extracted Java artifacts (i.e., class, interface, method, constructor, and static initialization block). The argument *metric* defines the object-oriented metric to be examined. Arguments *lowerbound* and *upperbound* represent the minimum and maximum values, respectively. A threshold value representing a limit value is given by the last argument *threshold*. The script selects all nodes of type *type* in a Rigi graph that have higher value than *threshold* of *metric*. An example call of the script could be *java_select_metric Class CC 3.5 10.3*. Because Tcl is an interpretable scripting language, the script library of Rigi can easily be extended; new scripts can be added on the fly. This allows the reverse engineer to write and use scripts that have specific tasks, such as scripts that support the analysis of the metric values.

```
proc java_select_metrics {nodetype metric lowerbound upperbound} {
    rcl_select_none
    rcl_select_type $nodetype
    set winnodes [rcl_select_get_list]
    rcl_select_none
    foreach n $winnodes {
        set val [rcl_get_node_attr $n $metric]
        if {$val >= $lowerbound && $val <= $upperbound} {
            rcl_select_id $n 1
        }
    }
}
```

Figure 1. A script that identifies parts of the target software for which a given metric is in a desired value range.

2. The object-oriented metrics suite

The Chidamber and Kemerer metrics suite contains six object-oriented metrics, which have been discussed extensively in the object-oriented metrics literature. We distinguish three metrics categories: *inheritance metrics*, *communication metrics*, and *complexity metrics*. Inheritance metrics are used to examine the inheritance hierarchy of object-oriented programs; the communication metrics estimate the *internal* and *external* communication of software components; and the complexity metrics measure the logical structure complexity of selected components.

2.1. Inheritance metrics

In Java the inheritance hierarchy is a single tree and, hence, all classes inherit from a single root or object class called *java.lang.Object*. For the purpose of this discussion we distinguish between *foundation* and *application* classes to separate the inheritance tree into two parts. The foundation classes encompass all those classes that are part of the Java Development Kit (JDK), the Java foundation classes, and the classes from any other Java library. The application classes include all other application-oriented or subject-system classes.

Since our goal is to measure design and code qualities of the subject system, we do not take the foundation

classes into account when computing measures for the inheritance metrics. If we were to include the foundation classes it would skew our results towards the quality of the foundation classes. Applying the metrics to the foundation classes, however, provides a solid base line against which we can compare the measures derived from the application classes.

We employ two metrics to evaluate the quality of an inheritance hierarchy:

1. the Depth of Inheritance Tree (*DIT*) and
2. the Number of Children (*NOC*).

These metrics were first introduced by Chidamber and Kemerer [2].

In a language with single inheritance, such as Java, the depth of a class or interface or the Depth of an Inheritance Tree (*DIT*) is simply the number of its ancestor classes or interfaces, that is, the number of classes or interfaces along the path to the single root class or interface. The *DIT* value of a class indicates how many ancestor classes potentially affect it. This metric measures the size or design complexity of a class or an interface. The size of a class (i.e., the number of methods and instance variables of a class) and, hence, the complexity of a class, increases with the depth of the inheritance tree. Since classes and interfaces belonging to foundations classes are ignored, the root class/interface is considered to be the one that does not extend/implement any other class/interface belonging to the subject system.

The Number of Children (*NOC*) of a class is the number of classes that extend this class. For an interface, *NOC* is the sum of the number of interfaces that extend it and the number of classes that implement it. The *NOC* value of a class is the number of classes in its subtree or how many classes are potentially affected by it. The *NOC* value of a class or an interface is a good indicator of how the design of the system is affected it is changed. Classes or interfaces with a high *NOC* value should be maintained by the most experienced software engineers.

The *DIT* and *NOC* numbers of a class are good indicators for the design complexity of a class. Thus, the inheritance metrics can be used to predict reusability and design complexity.

2.2. Communication metrics

We employ three metrics to measure coupling and cohesion among classes and objects:

1. Response For a Class (*RFC*);
2. Coupling Between Objects (*CBO*); and
3. Lack of Cohesion in Methods (*LCOM**).

As the inheritance metrics these metrics were also introduced by Chidamber and Kemerer in their seminal 1991 OOPSLA paper [2].

RFC is a measure for the size or the complexity of a class and the interaction or communication of the class with the rest of the system. *RFC* is the sum of the number of methods in a class and the number of external methods that are potentially called by this class. To compute the number of methods of a class, we count the *regular methods*, the *constructors*, and the *static initialization blocks* that belong to it. It ignores the calls to members of the same class. Thus for a class C, let M_i be the set of all member functions in C. Let M_o be the set of all member functions, belonging to other classes, that are called by the members of M_i. Then *RFC*(C) is the size of the set $M_i \cup M_o$.

By this definition the *RFC* metric treats all calls to external methods the same. However, calling a method of a super class does not add as much to the complexity as calling a method of another class. For example, in Java the default constructor of the super class is called automatically from the constructors of its subclasses. Furthermore, overloading a method in a subclass typically contains a call to the overloaded method of the super class.

The goal of the *CBO* metric is to distinguish between these cases. It measures the coupling of a class with those classes with which it is not related through inheritance. To compute the *CBO* measure, both constructors and methods are taken into account. Following the dependencies between two classes that are not in a super class-subclass relationship constitutes coupling including method calls, constructor calls, instance variable assignments, or other kind of instance variable accesses.

In the literature, several formulas have been introduced to compute the Lack of Cohesion metric *LCOM* [1, 2, 7, 8, 10]. We adopted a definition introduced by Henderson-Sellers to analyze Java programs [8]. It measures the lack of cohesion or dissimilarity among all the methods of a class except the inherited methods but including overloaded methods. The *LCOM** value denotes the number of pairs of methods without shared instance variables, minus the number of pairs which do share instance variables.

Consider a class C, its set M of m methods $M_1, M_2, \ldots M_m$, and its set A of a data members $A_1, A_2, \ldots A_a$ accessed by M. Let $\mu(A_k)$ be the number of methods that access data attribute A_k where $1 \leq k \leq a$. Then *LCOM**$(C(M, A))$ is defined as follows:

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

The methods of a class should be logically related. If a class exhibits low method cohesion it indicates that the de-

sign 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. The *LCOM** metric helps to identify such flaws in the design.

2.3. Complexity metrics

Most object-oriented metric suites include traditional code complexity metrics. We employ the *Cyclomatic Complexity (CC)* and the *Weighted Methods per Class (WMC)* metrics to measure the complexity of control flow. McCabe's Cyclomatic Complexity [12] is used to assess the logical structure or the complexity of a sequential algorithm, such as a method, a function, or a procedure. It counts the number of test cases that are needed to test the method comprehensively. The *CC* metric is used by several other metrics. We use the following formula, adopted from Henderson-Sellers [8], to compute *CC*:

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

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

Finally, *WMC* is defined as the sum of the complexities of all the methods of a class except the inherited methods but including overloaded methods. The Henderson-Seller Cyclomatic Complexity *CC* is used to compute the complexity of a method.

$$WMC = \sum_{i=1}^{n} CC_i \tag{3}$$

Thus, *WMC* is proportional to the number of methods in a class and to the complexity of the logical structure of all methods. The higher a class' *WMC* measure is, the more difficult it is to understand and maintain it.

3. Collecting and visualizing information

The software artifacts and their dependencies are directly extracted from Java class files [24]. The extracted information includes the following components: classes, interfaces, methods, constructors, variables, and static initialization blocks. The extracted dependencies among these artifacts include extension relationships (i.e., a class extends another class), implementation relationships between classes and their interfaces, containment relationships (i.e., a class contains a method), call relationships (i.e., a method calls another method), access relationships (i.e., a method accesses a variable), and assignment relationships (i.e., a method assigns a value for a variable). The *extractor,* written in Java, uses some of the public classes of the *sun.tools.java* package of *JDK 1.2*. Other Java byte code extractors are discussed, for instance, in [17, 16].

Rigi is used to visualize the constructed static dependency graphs. In Rigi, the software artifacts are depicted as nodes and relationships as directed edges between nodes. Different types of nodes or edges are represented by different colors.

Using the Shimba reverse engineering environment, the user can interactively select any subset of the metric suite to be applied. Measures are computed for those software components that are in the current context, that is, those artifacts for which the static information has been extracted from the byte code.

The measures are then added as attribute values to Rigi nodes. By default, 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 screen snapshot of a Rigi session in Figure 2. In this case, the user has selected node *de.uni_paderborn.dis.DisRow* and opened an attribute visualization widget that lists all the attribute values of the node. Using measures as node attributes, Rigi provides flexible and powerful mechanisms to analyze the values and the static dependencies.

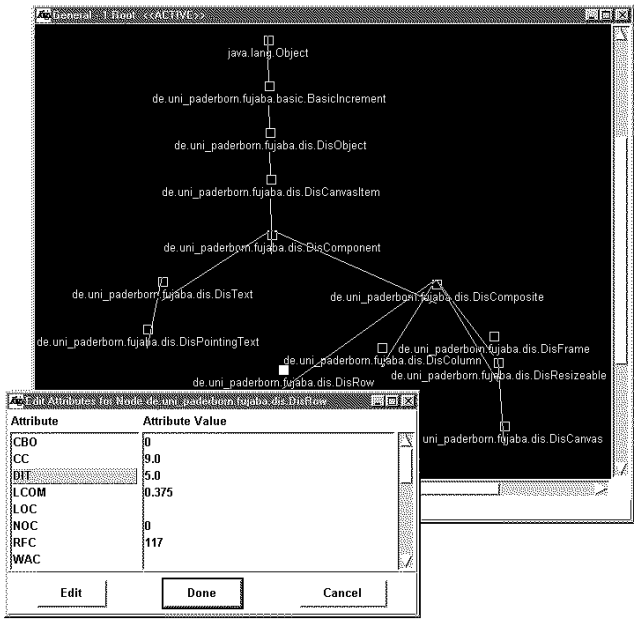

Figure 2. Attribute values of a selected node can be examined in Rigi by opening a popup window.

4. Threshold values

The object-oriented metrics literature discusses language-dependent heuristics for threshold values which correspond to high and low-quality software components. They are usually based on experiences over several software projects and hence should be treated as heuristics and recommendations. Lorenz and Kidd propose threshold values for several object-oriented metrics for C++ and Smalltalk based on their experience with selected C++ and Smalltalk projects [11]. Few papers report on the experiences with Java projects. As a result there are few heuristics for threshold values for assessing Java programs. The threshold values for Smalltalk are probably a better starting point for Java than those for C++, since C++ is not a pure object-oriented language.

In this paper we do not present fixed ranges or threshold values for Java components, but we recommend that the engineers, who assess Java software, experiment with value ranges using the end-user programmable scripts provided through the Shimba reverse engineering environment. However, the reverse engineer has the option of running a script called *java_select_attributes_thresh* to define initial threshold values. The script takes three arguments *type*, *metric*, and *threshold*. As in script *java_select_metric* in Figure 1, the argument *type* defines a node type in a Rigi graph and the argument *metrics* defines the object-oriented metrics to be examined. A threshold value representing a limit value is given by the last argument *threshold*. The script selects all nodes of a given type in a Rigi graph that have higher value than type *type* in a Rigi graph that have higher value than *threshold* of *metric*. By running this script the user can quickly find software artifacts that have critical or extreme measures (i.e., classes that are most complex).

5. Applying the metrics to the FUJABA system

To gain experience with the object-oriented metrics for program understanding purposes, we analyzed the FUJABA system using Shimba. FUJABA was developed at the University of Paderborn, Germany and is freely available and downloadable from the Web [18].

The primary objective of the FUJABA project and environment is it Round Trip Engineering using the Unified Modeling Language (UML), Story Driven Modeling (SDM), Design Patterns, and Java.

We investigated FUJABA Version 0.6.3-0. Extracting static information and applying the metrics suite resulted a dependency graph of almost 26,000 software artifacts, annotated with the computed measures.

5.1. Extreme measures

To locate extreme measures, we ran RCL scripts. Figure 3 shows how the methods with highest *CC* values and their call dependecies are displayed in Shimba. The nodes can be easily found by running the *java_select_attributes_thresh* script (see Section 4). By running the script again five most complex methods were found. Those nodes are selected and their names are shown in the figure. The rest of the graph has been filtered out. Note that the subgraph of seven nodes on the bottom of the figure forms a complex structure in which methods with high *CC* values call each other. The *java_select_attributes_thresh* script was also used to construct a Rigi graph in Figure 2 to search classes that have DIT values higher than four. The graph also includes the whole inheritance hierarchy of those classes. That can be easily achieved by running scripts of the standard RCL library of Rigi.

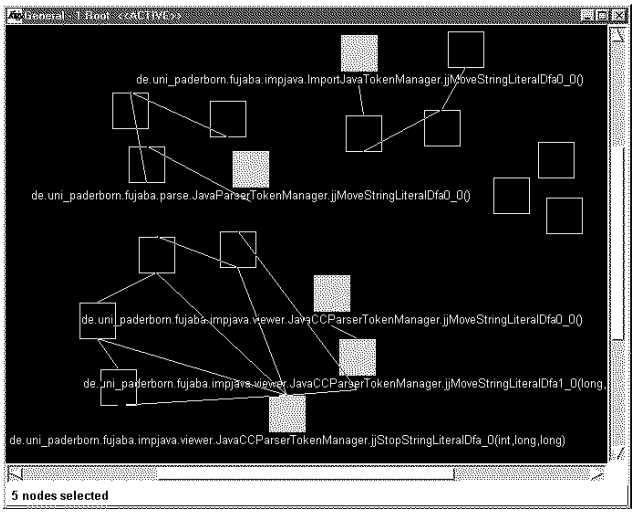

Figure 3. FUJABA methods with highest *CC* values

We then investigated the communication metrics *RFC*, *CBO*, and *LCOM** for FUJABA. By running the *java_select_attributes_thresh* script for the communication metrics, it was readily apparent that most coupled classes with respect to the *RFC* metric belong to *de.uni_paderborn.fujaba.uml* package. From the top 25 classes with the highest *RFC* values over 20 classes belong to this package. Similar ratios were obtained for the *CBO* and *LCOM** metrics: 13/39 and 11/119, respectively. One conclusion we might derive is that *de.uni_paderborn.fujaba.uml* is one of the largest packages in FUJABA which is indeed correct.

The high coupling measures encouraged us to take a

closer look at the measures generated for this package. The classes with highest *RFC* and *CBO* values are listed in Table 1 in decreasing order of their original metric values. Table 2, in turn, shows the highest *LCOM** measures for this package. Some of the classes listed in Table 2 are inner classes. The fully qualified 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*
UMLClass (629)	TestProject (23)
UMLProject (587)	UMLClass (16)
TestProject (500)	UMLActivity (10)
UMLFile (465)	UMLActivityDiagram (10)
UMLClassDiagram (391)	UMLMethod (9)
UMLTypeList (377)	UMLObject (9)
UMLStoryPattern (371)	UMLStoryActivity (9)

Table 1. Classes in the de.uni_paderborn.fujaba.uml package with the highest *RFC* and *CBO* measures. The classes are listed in decreasing order of their measures. The values are shown in braces.

*LCOM**
UMLFile$UMLPackageComparator (2.0)
UMLStoryPattern$collabStatLessThan (2.0)
UMLLink (0.987)
UMLTransitionGuard (0.980)
UMLIncrement (0.980)
UMLLinkSet (0.975)
UMLClass (0.963)

Table 2. Classes in the de.uni_paderborn.fujaba.uml package with the highest *LCOM measures. Classes are listed in decreasing order of their measures. The values are shown in braces.**

To examine the dependencies of the class *UMLClass* (with the highest *RFC* value), we executed some queries on the static dependency graph using RCL scripts. A standard RCL script *select_neigbors* was used to identify the classes that are coupled with the class *UMLClass*. The script was used in three phases. It was first used to find methods, constructors, and static initialization blocks of the class *UMLClass*. Then the same script was used to find all the other methods, constructors, and static initialization blocks that have a call dependency with those member functions. The script was used once more to select the owner classes of all the member functions found. Many of the methods found do no belong to FUJABA but have a call dependency with the member functions of the class *UMLClass*. For example, several methods of class *java.lang.String* are called. Using couple of RCL scripts, such methods were collapsed into a high-level Rigi node that represents the package they belong to. Finally, the rest of the graph was filtered out. Figure 4 exhibits the resulting graph. The inheritance relationships shown in the figure indicate that calls of superclass methods form a large part of coupling. In many cases, such calls cannot be considered harmful.

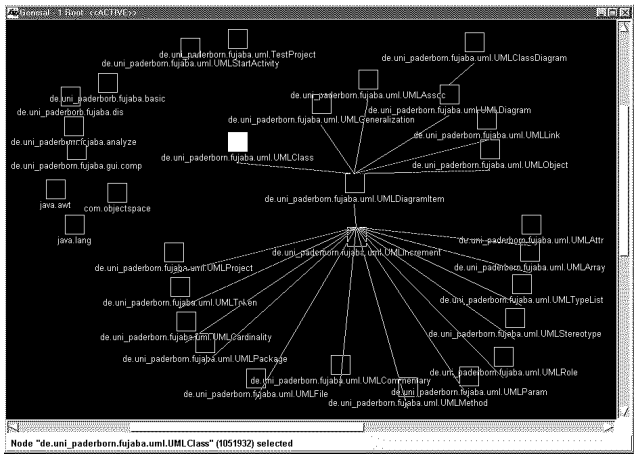

Figure 4. Classes and packages the class *UMLClass* is coupled with.

5.2. Correlation of the metrics

We can investigate the measures further by exporting the data to a spreadsheet. We again use the end-user programmable Rigi interface to write measures into a file readable by the Microsoft Excel spreadsheet. Now we can take advantage of all the functions, programming capability, and diagraming techniques Excel provides. Figure 5 depicts a line diagram, produced using Excel, of the original communication measures for package *de.uni_paderborn.fujaba.uml*. From the diagram, it is difficult to conclude whether the values of different metrics correlate, because the metrics have different value ranges. By running another RCL script on the static dependency graph, we normalized complexity and communication measures in order to compare and analyze the generated measures more effectively. Each measure is normalized by subtracting the mean from it and dividing the result by the standard deviation. The resulting values are depicted in Figure 6 and have zero mean and unit deviation. The correlation between the metrics is easier to recognize from diagram in Figure 6 than from diagram in Figure 5.

To complete the investigation of the communication metrics, we now study the *LCOM** metric for the

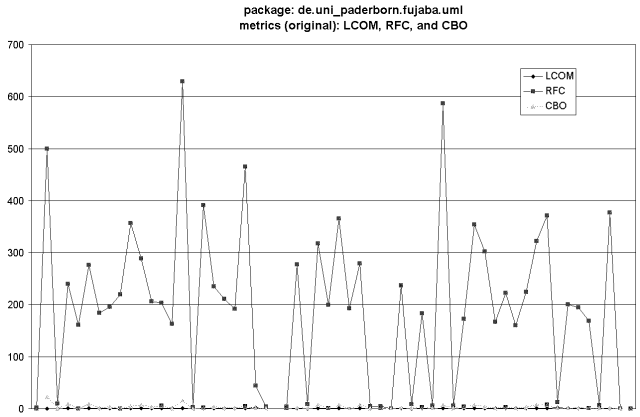

Figure 5. The original values of *RFC*, *CBO*, and *LCOM metrics for classes in de.uni_paderborn.fujaba.uml package.**

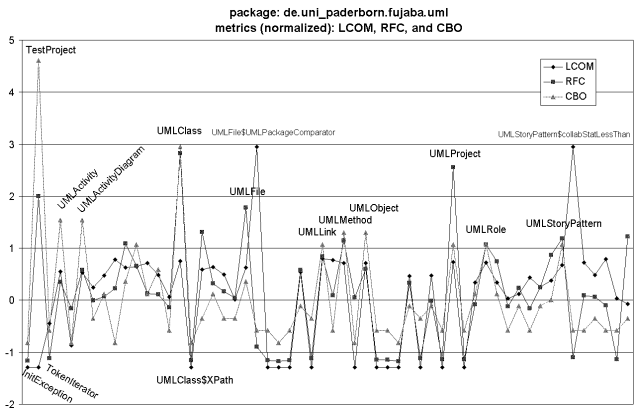

Figure 6. The normalized measures by applying the *RFC*, *CBO*, and *LCOM metrics to the classes of the *de.uni_paderborn.fujaba.uml* package.**

de.uni_paderborn.fujaba.uml package. The *LCOM** metric not only measures communication but also logical complexity. Thus, it is useful to compare the *LCOM** measures with the *CC* and *WMC* measures. Figure 7 exhibits a line diagram, produced with Excel, of the normalized complexity measures for all classes in the *de.uni_paderborn.fujaba.uml* package.

If we compare the graphs in Figures 6 and 7, we observe that the shapes of the lines in both figures are similar, that is, most of the classes that have high communication measures also have high complexity metric values. The most obvious exception is the class *TestProject*, for which *RFC* and *CBO* values are very high, but the *CC*, *WMC*, and *LCOM** measures are low. Such classes typically consists of methods that mostly call and/or are called by other classes and, hence, do not implement complicated algorithms. This is the case also with class *TestProject*.

Figures 6 and 7 illustrate the fact that class *UMLClass* has high measures for both the communication and complexity metrics. When examining the size of the *UMLClass.class* and *UMLClass.java* files, we observe that the *UMLClass* class is clearly the largest class in the *de.uni_paderborn.fujaba.uml* package (i.e., the size of the *UMLClass.java* file is more than double the size of *UMLIncrement*, the second largest class.

To study the correlation of metrics in more detail, we used an Excel macro $CORREL$ to generate a correlation matrix from the normalized measures. Table 3 exhibits the pairwise correlation values. If the coefficient is greater than 0,4, we consider two metrics to be correlated. The larger the coefficient is (the maximum being 1), the more correlated two metrics are. Note that dependecies between metrics cause high correlation coefficients in some cases. For instance, the coefficient for *WMC* and *CC* is 0,98, which is explained by the fact that *CC* is used to calculate *WMC*.

	CC	WMC	LCOM	RFC	CBO	NOC	DIT
CC	**1,00**	**0,98**	0,38	**0,69**	**0,53**	0,15	0,04
WCM	**0,98**	**1,00**	0,37	**0,69**	**0,54**	0,14	0,06
LCOM	0,38	0,37	**1,00**	**0,41**	0,18	-0,01	0,31
RFC	**0,69**	**0,69**	**0,41**	**1,00**	**0,72**	-0,02	**0,45**
CBO	**0,53**	**0,54**	0,18	**0,72**	**1,00**	-0,13	0,12
NOC	0,15	0,14	-0,01	-0,02	-0,13	**1,00**	-0,10
DIT	0,04	0,06	0,31	**0,45**	0,12	-0,10	**1,00**

Table 3. A correlation matrix of normalized measures.

6. Related research

Software metrics, including object-oriented metrics, are used in many enchanced reverse engineering and re-engineering environments to help the user analyze constructed views of the software being investigated. Such environments and tool sets include McCabe Reengineer from McCabe & Associates Inc. provides views of the system architecture and views of the interaction among modules based on the analysis of the source code. Metrics are used to measure the complexity and structuredness of software components. The results are illustrated by coloring the views (e.g., to recognize exceptional metrics values). In our approach, we run scripts which execute queries on the dependency graphs. While Shimba is a prototype environment that supports dynamic and static reverse engineering of Java software, McCabe Reengineer supports several languages and provides a large set of tools that can be used for testing the subject software and to assist the re-engineering process in various ways.

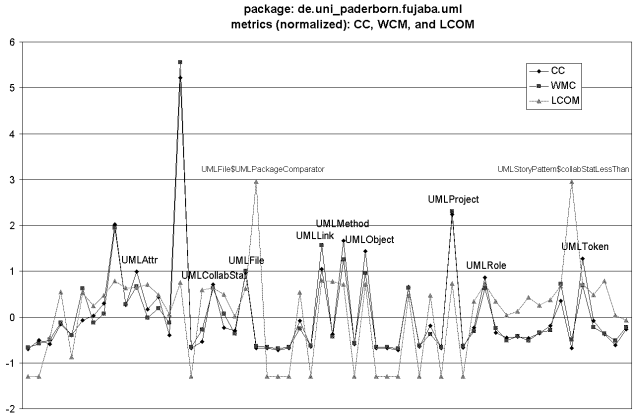

Figure 7. The normalized measures obtained by applying the *CC*, *WMC*, and *LCOM metrics to the classes of the *de.uni_paderborn.fujaba.uml* package.**

CodeCrawler is a platform built to support program understanding by combining metrics and program visualization [5]. CodeCrawler provides views that show selected structural aspects of the software as a simple two-dimensional graph. As in Rigi, nodes in a dependency graph represent software artifacts (e.g., a C++ class). CodeGrawler is able to visualize up to five metric values simultaneously on a single node: the size of a node can be used to render two measurements (i.e., the width and the height); the position of the node can also render two measurements (i.e., the X and Y coordinates); and the color of the node, a gradient color between white and black, can be used to visualize one measurement. Unlike the approach presented in this paper, the visualization technique of CodeCrawler is not able to show all the static dependencies within the software at the same time. Furthermore, the graph is not editable and does not support querying techniques.

Logiscope from CS Verilog supports both static and dynamic analysis of a software system. It is able to produce static call and control graphs of the subject system. In Shimba, the control graph information is used to calculate *CC* and *WMC* values but the graphs are not presented graphically. In addition to call dependencies, the constructed dependency graph in Shimba contains information about inheritance, implementation, variable accesses, and variable assignments. In Logiscope, quantitative information based on software metrics and graphs can be generated to help the user to diagnose defects. The large set of metrics supported by Logiscope includes inheritance, communication, and complexity metrics. The Kiviat metric graphs are used to identify components that have exceptional measures. In Shimba, the exceptional measures are identified by making queries on the annotated static dependency graph.

Sneed and Dombovari [20] introduce an approach to model the requirement specification and the system implementation of a large, distribute C/C++ system. The approach combines forward and reverse engineering techniques and supports software maintenance. Shimba does not corrently support forward engineering but it can be extended for that purpose. For example, the dynamic reverse engineering process results UML type of sequence diagrams and state diagrams that are visualized with SCED. Those diagrams can then be used in the analysis and design phases of object-oriented software construction. Several size metrics and code characteristic measurements are used in [20]. The values are presented in a metric report. Tool called CppSpec is used to model the entities and relationships extracted from the code. CppSpec can generate various kinds tree diagrams and cross reference diagrams on demand. Unlike in our approach, the metrics are not directly mapped with the program visualization. As in Shimba, add hoc queries are supported in CppSpec. The usage of queries in CppSpec is limited to a fixed set of questions that can be asked. In Shimba, the amount or type of queries is not limited. The queries can be used to construct different views to the target software. This is not supported in [20].

The Hindsight reverse engineering tool from IntegriSoft Inc. can produce different kinds of reports, charts, and diagrams that help program understanding. Hindsight uses software metrics for analyzing the complexity of the subject system. Metrics are presented in cross reference reports, exception reports, Kiviat diagrams, and metric charts. Metrics can also overlaid on a call graph. In Shimba, metrics can be annotated to the static dependency graph or saved in a file readable, for example, by the Microsoft Excel spreadsheet. Hindsight gives automated support for analyzing the impact of code changes and supports testing. These facilities are not supported in Shimba.

7. Discussion

This paper presented an approach on how to apply object-oriented metrics using a reverse engineering environment for program understanding purposes. Combining metrics information with a graphical reverse engineering tool helps both the reverse engineering process and the analysis of the measures.

In reverse engineering, one of the most challenging tasks is building abstract views from the parsed static dependencies. This can be accomplished by synthesizing high level components or concepts that represent software artifacts which are highly cohesive and loosely coupled with other components. Metrics can be used to find such parts and, hence, support this task effectively. A reverse engineering tool can also be used to find software artifacts that have extreme or exceptional measures. Such values need to be rec-

ognized in order to propose restructuring or refactoring of the offending components.

We used the Shimba reverse engineering tool to implement our ideas about combining metrics and program understanding technology. Shimba contains Rigi, SCED, and the object-oriented metrics suite. Rigi is used to visualize the static dependencies of the subject system as nested graphs, which is extracted from its Java byte code. The graphs are then annotated with the measures obtained from applying the metrics suite to the subject system. Rigi provides an extensible script library that can be used for executing queries on the graph to explore, inspect, and modify it.

We contributed several new scripts to the RCL scripting library to identify extreme measures and to find measures that fall within a given range. Having identified interesting or highly cohesive parts within the subject system, the reverse engineer can investigate the static dependency graph further to figure out how the complex parts have been built, how they are related to the rest of the system, and what the measures of those related components are. Different views to the subject software can be quickly generated using the scripts. The views help the reverse engineer to find answers to such questions. Furthermore, the measures can be normalized by running another script. The normalization is needed to correlate different metrics.

The measures are computed by running a metrics program integrated with the Shimba reverse engineering environment. Some of the metrics could also be calculated using Rigi scripts. 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 (even dynamically, if desired). However, this is not possible for all the metrics. For example, the *CC* metric is computed using the control flow information that is not usually included in Rigi graphs but can be generated by the byte code extractor. The Java compiler translates conditional statements to specific binary instructions [24]. The control flow can be determined by examining the usage of such instructions.

Shimba also supports dynamic reverse engineering. During the run-time analysis, weight values for method calls, constructor invocations, and thrown exceptions can be added to the static Rigi graphs. Again, the weight values are added as node attributes the same way as measures from the metrics suite. This provides sufficent information for computing selected dynamic metrics. For example, the user can calculate the actual communication between objects, based on the actual usage of components.

When we investigated the measures for the FUJABA software, we did not discover big flaws in the design. By examining the communication metrics *RFC*, *CBO*, and *LCOM**, design flaws in the class structure and in information hiding strategy were exhibited. If the *LCOM** measure of a class is high, but the *RFC* and *CBO* values are low, then it can be suspected that the class might have unused variables or the variables have not been properly selected for the class. By examining the complexity metrics *CC* and *WMC*, complex data structures can be recognized. The inheritance metrics *NOC* and *DIT* can be used to study the inheritance hierarchy and, hence, help in estimating the reusability and extensibility of the subject system.

Acknowledgements

We wish to thank Michael Przybilski and the anonymous reviews for their helpful comments.

References

[1] S. Chidamber and C. Kemerer. A metrics suite for object-oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[2] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object-oriented design. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*, pages 197–211, 1991.

[3] R. S. Corporation. The unified modeling language notation guide v1.3. [http://www.rational.com], 1999.

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

[5] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering approach combining metrics and program visualization. In *the 6th Working Conference on Reverse Engineering (WCRE99)*, pages 175–186, 1999.

[6] M. Fowler. *Refactoring*. Addison-Wesley, 1999.

[7] I. Graham. *Migrating to Object Technology*. Addison-Wesley, 1995.

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

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

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

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

[12] T. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, 1976.

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

[14] 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.

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

[16] S. Porat, B. Mendelson, and I. Shapira. Sharpening global static analysis to cope with java. In *CASCON98*, 1998.

[17] D. Rayside and K. Kontogiannis. Extracting java library subsets for deployment on embedded systems. In *the 3rd European Conference on Software Maintenance and Reengineering (CSMR99)*, 1999.

[18] I. Rockel and F. Heimes. Fujaba - homepage. [http://www.uni_paderborn.de/fachbereich/AG/schaefer/ag_dt/PG/Fujaba/fujaba.html], February 1998.

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

[20] H. Sneed and T. Dombovari. Comprehending a complex, distributed, object-oriented software system - a report from the field. In *the 7th International Workshop on Program Comprehension (IWPC99)*, 1999.

[21] T. Systä. On the relationships between static and dynamic models in reverse engineering java software. In *the 6th Working Conference on Reverse Engineering (WCRE99)*, pages 304–313, 1999.

[22] S. Tilley and H. Müller. Using virtual subsystems in project management. In *IEEE Sixth International Conference on Computer-Aided Software Engineering(CASE)*, pages 144–153. IEEE Computer Society Press, 1993.

[23] S. Tilley and H. Müller. Using virtual subsystems in project management. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

[24] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, 1998.

[25] K. Wong. Rigi user's manual version 5.4.1. [http://www.rigi.csc.uvic.ca/rigi/manual/user.html], September 1997.