REM4j - A framework for measuring the reverse engineering capability of UML CASE tools

Steven Kearney and James F. Power

Dept. of Computer Science, National University of Ireland, Maynooth, Co. Kildare, Ireland.

Abstract

Reverse Engineering is becoming increasingly important in the software development world today as many organizations are battling to understand and maintain old legacy systems. Today's software engineers have inherited these legacy systems which they may know little about yet have to maintain, extend and improve. Currently there is no framework or strategy that an organisation can use to determine which UML CASE tool to use. This paper sets down such a framework, to allow organisations to base their tool choice on this reliable framework.

We present the REM4j tool, an automated tool, for benchmarking UML CASE tools, we then use REM4j to carry out one such evaluation with eleven UML CASE tools. This framework allows us to reach a conclusion as to which is the most accurate and reliable UML CASE tool.

1. Introduction

Many UML CASE tools provide the ability to reverse engineer UML diagrams from source code, and these diagrams can be essential to software maintainers in understanding the design of a system.

Since there are many UML CASE tools available, the question many organisations face is: *Which one suits our needs best?* To answer this question they will need to evaluate all the available tools, measure the results of this evaluation and rank the tools based on the evaluation.

This paper establishes a framework for benchmarking and evaluating UML CASE tools. We describe the construction of the framework, and its use on an oracle program, designed to expose inaccuracies in reverse engineering. We then examine the impact these inaccuracies have by running the UML tools over a suite of real-world programs, and examining the variance in reported metrics.

2. Background and Related Work

Often legacy systems have an originally convoluted design, obsolete documentation, and the original developers may have left the company. Software may have numerous patches and fixes applied over time. It can be an arduous task to understand a legacy system [11, 12]. *Re-engineering* is the examination of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [2]. *Reverse Engineering* is the process of analyzing a subject system to create representations of the system in another form or at a higher level of abstraction [2, 16, 9, 1].

There has been much research carried out that investigates tools and techniques for reverse engineering. Notable examples include the RIGI toolset for reverse engineering [15], the Dali Workbench [8], CPPX [4] and Columbus/CAN [6]. More recently, reverse engineering tools have begun to use the diagrams of the Unified Modeling Language (UML) as a representation for reverse-engineered artifacts. One of the major challenges faced by designers of reverse engineering tools is the struggle to keep with continuously evolving UML versions, as well as version of the associated XML Metadata Interchange (XMI) [10].

Our approach exploits object-oriented software metrics to provide us with a means to collect information about the characteristics of a Java application [13, 14]. These characteristics are important since if we reverse engineer a Java application we would expect the characteristics exported in the XMI file to be an accurate reflection of the application.

While Cooper et al. studied the inaccuracies that occur from Forward Engineering vs. Reverse Engineering, they stopped short at evaluation the reliability of the tool to export XMI [3]. Jiang and Systä explored the differences in Exchange Formats between UML CASE tools and they investigated if UML CASE tools delivered on the OMG ideal of interchangeable XMI files, but they stopped short of automating the process[7].

Our approach, centered on the REM4j framework is unique in that it can automate the reverse engineering, metric capture and evaluation of metrics. We first calibrate the process on a specially-designed *oracle* program, and then use peer-evaluation to investigate the performance of tools on a suite of real-world Java programs.

3. Experimental Setup

REM4j (*Reverse Engineered Metrics 4 Java*) takes Java

Tool	Vendor	Version	Abbrv
ArgoUML	Tigris	0.22	AR
MagicDraw	Magicdraw	12.0	MD
Bouml	Bouml	2.17	BO
Metamill	Metamill	4.2	MM
Visual Paradigm	Visual Paradigm	3.1	VP
Jude	Change Vision	Prof 6.0	JU
Enterprise Architect	Sparx Systems	6.5	EA
UModel	Altova	2006 r.2	UM
ESS-Model	Ess-Model	2.2	ES
Ideogramic UML	Ideogramic	2.3.3	IC
Poseidon for UML	Gentleware	4.2	PO

Table 1. The 11 tools we have chosen for our study. The final column gives an abbreviation for each tool that we use in later tables.

source code as its input, and reverse engineers a class diagram from the code use a pre-recorded set of macros for a a particular UML CASE tool. The result is exported in XMI format and then run through a commercially available metric calculation engine, after which REM4j colllates all the results and saves them to a CSV file.

The first step in this experiment was to choose a set of UML CASE tools for the evaluation. We tried to gather as wide a range of tools as possible, with the only selection criterion being their ability to reverse engineer Java source code back to UML class diagrams, and to export the diagrams in XMI format. These tools are listed in Table 1.

Our REM4j framework is designed to provide a modular environment within which these UML tools can be run and their output data analysed. The framework needs to be able to open a UML CASE tool, import Java source code, reverse engineer it, export it to XMI, pipe the XMI into a metric calculation engine and gather and collate the results into one readable CSV file. REM4j uses a number of third-party applications, and serves to coordinate the interaction between these tools.

At the center of the framework is the *SDMetrics* tool [17], which is a powerful commercial application that is capable of analysing XMI and computing metrics based on that XMI. Automation is achieved using *Autohotkey*, a macro utility for Microsoft Windows that has the ability to record keystrokes and mouse clicks. Autohotkey provides the ability to write a macro for a particular CASE tool and then compile it to an executable. Finally, *Chart2D* is an open-source charting class library, and is used by REM4j to visualise its results.

REM4j takes two inputs when starting: the directories of the Java source files you would like to reverse engineer, and the AutoHotKey directory. If for example three source code directories are selected and then four UML tools are selected, the reverse engineering loop would execute twelve times. Each source code directory is reverse engineered and exported to XMI and then piped into the metric calculation engine tool. When the REM4j automation tool has finished executing it generates its results in both text and graphical formats.

4. Exploratory Analysis using an Oracle

We chose the term *Oracle* to describe a piece of Java source code for which all its characteristics, elements and attributes were known. The *Oracle* application was designed and written explicitly for this paper. In particular, all of the metric values were calculated in advance and the code was constructed to have as many different metric scenarios as was feasible.

The *Oracle* application was written with a 0-1-2 (ZOT) metric policy in place. For example, the *Oracle* application had at least one class with no *Public Methods*, at least one class with exactly one *Public Method* and at least one class with more than one *Public Method*.

Figure 1 shows a Class Diagram which illustrates the structure and design of the *Oracle* application. On this diagram, overlapping classes are inner classes, dotted lines represent an *implements* relationship and a solid line represents an *extends* relationship.

In the next three sections we will break down the metrics that this paper investigates into *Size Metrics* and *Inheritance Metrics*. The SDMetrics tool also reports on *Coupling Metrics* but, since these must be evaluated on a per-class basis, we have not considered them further here.

4.1. Size Metrics

Size metrics measure the size of design elements, this is simply a count of the elements that are contained within an application.

- **Number of Variables (NoV)** The NoV metric refers to sum of the number of variables in all classes regardless of type, visibility, changeability or scope. It does not count inherited variables, or variables that are members of an association [13].

As shown in the Class Diagram the *Oracle* application clearly has 12 variables or attributes. However 4 of the 11 tools produced a figure other than 12. Both Jude and Bouml had the lowest figure as they reported the *Oracle* application having only 7 variables, while Poseidon reported 8 and Ideogramic UML reported 9.

- **Number of Methods (NoM)** The NoM metric has a value that is the sum of the number of all methods in all classes regardless of type, visibility, changeability or scope. It does not count inherited methods, but it does count abstract methods [13].

The *Oracle* code contains exactly 23 Methods, so any derivation from this figure would suggest an inaccurate tool. All UML CASE tools with the exception of Bouml which reported 20, produced the correct figure of 23.

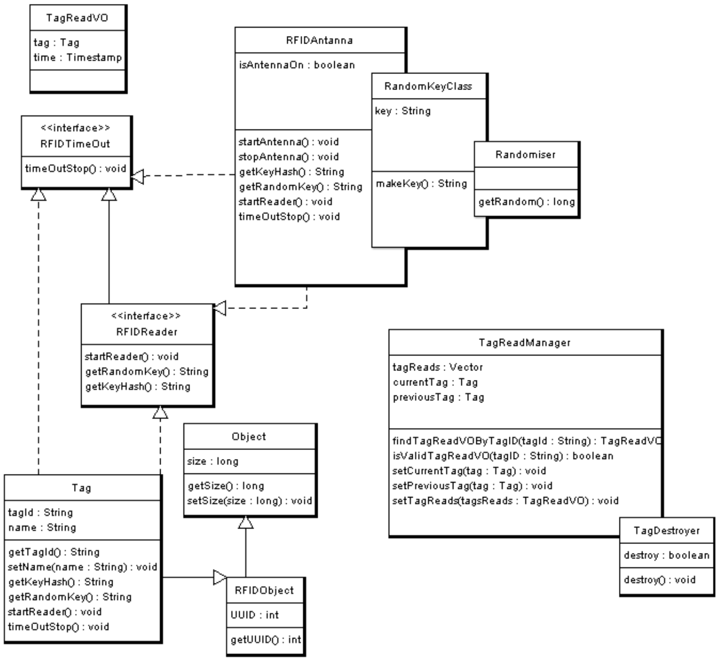

Figure 1. This Class Diagram shows the 11 classes of the *Oracle* application.

- **Number of Public Methods (NoPM)** The $NoPM$ metric has a value which is the sum of the number of methods in all classes that have public visibility [13].

The *Oracle* application was written with exactly 20 public methods, all of the tools agreed with this figure except one, Bouml, which reported 18. This is not surprising as Bouml only reported a total of 20 methods when there was actually 23 methods in the application.

- **Number of Setters (NoS)** The NoS metric counts any Method that begins with 'set'. The *Oracle* application contains 5 such methods, and all tools produced a NoS value of 5.
- **Number of Getters (NoG)** This metric counts any Method that begins with 'get', 'is' or 'has'. The *Oracle* application contains 9 such methods, and all tools produced a NoG value of 9.
- **Inner Classes (IC)** This metric will return the total number of inner classes nested within an application. The Oracle application contains the class *RFIDAntenna* which has an inner class nesting level of 2. The *RandomKey* class has a inner class nesting of 1, the *TagReadManager* also has a nesting of 1. These are summed to produce a value of 4. Ideogramic UML, Bouml, Enterprise Architect and ESSModel reported 0 inner classes, while MagicDraw UML reported 7. ArgoUML, Metamill, Poesideon, Visual Paradigm and Jude correctly reported 4.
- **Total Number of Classes (ToC)** This is a count of the total amount of classes that the UML CASE tool exported in its XMI document. The class diagram in Figure 1 clearly shows 11 classes however none of the UML CASE tools reported this figure, they all reported figures of between 9 and 53. Table 2 shows how Jude reported 42 classes above the actual of 11 classes.

This error has most likely occurred due to the fact that different tools treat imported packages differently, such as the java.util.String class, while not part of the *Oracle* application it is used by it. Some tools count classes like the java.util.String class in the ToC metric, Thus making the metric unreliable. The ToC metric will not be evaluated further.

Size Metrics Summary As we know the correct value for all the size metrics, we can state if the tools passed or failed.

As Table 2 shows, Argo UML, Metamill and Visual Paradigm were the only tools to be correct on all evaluated metrics. The table displays the actual metric value, if a tool reported an incorrect value, the difference between the reported and the correct value is displayed.

4.2. Inheritance Metrics

Inheritance Metrics deal with polymorphism, depth and width of the Inheritance tree, the number of ancestors or descendants of a class.

- **Interfaces Implemented (II)**

The total number of interfaces that are implemented within an application. The *Oracle* application contains two interfaces, both of which are implemented by *RFIDAntenna* and *Tag*. As both interfaces are implemented twice, the correct value for the II metric is 4.

	NoV	NoM	NoPM	NoS	NoG	IC	ToC
Actual	*12*	*23*	*20*	*5*	*9*	*4*	*11*
AR	0	0	0	0	0	0	+2
MD	0	0	0	0	0	+3	+4
BO	-5	-3	-2	0	0	-4	-5
MM	0	0	0	0	0	0	-3
VP	0	0	0	0	0	0	-2
JU	-5	0	0	0	0	0	+42
EA	0	0	0	0	0	-4	+1
UM	0	0	0	0	0	-4	+5
ES	0	0	0	0	0	-4	+4
IC	-3	0	0	0	0	-4	-2
PO	-4	0	0	0	0	0	+2

Table 2. Size Metrics Results: For each size metric on the top row and each UML tool in the left column, this table shows the difference between the actual value and the value calculated.

	II	NoC	IT	CLD	MI	VI
Actual	*4*	*2*	*3*	*3*	*5*	*2*
AR	-4	0	0	0	0	0
MD	0	0	0	0	0	0
BO	-4	0	0	0	0	0
MM	-4	0	0	0	0	0
VP	0	0	0	0	0	0
JU	-4	0	0	0	0	0
EA	0	+1	+1	+1	0	0
UM	0	0	0	0	0	0
ES	0	0	0	0	0	0
IC	-4	0	0	0	0	0
PO	0	0	0	0	0	0

Table 3. Inheritance Metrics Results: For each inheritance metric on the top row and each UML tool in the left column, this table shows the difference between the actual value and the value calculated.

Ideogramic, ArgoUML, Jude, Metamill and Bouml returned 0 as the metric value, all other tools agreed and returned 4.

- **Number of Children (NoC)** This measures the number of immediate subclasses subordinated to a class in the class hierarchy.

In the *Oracle* application, two classes, *Object* and *RFIDObject* are extended by another class, *RFIDObject* extends *Object* and *Tag* extends *RFIDObject*. The value of NoC for the *Oracle* application is 2. All of the UML CASE tools agreed that it was 2, with the one exception of Enterprise Architect, which stated that it was 3.

- **Inheritance Tree (IT)** This metric represents the sum of the ancestors or descendants of each class within the application. The *Oracle* application has two instances where a class has a inheritance depth of greater than 0, this is the *Tag* class which has a depth of 2 and the *RFIDObject* class which has a depth of 1. Hence the total IT value is 3.

With the exception of Enterprise Architect which reported 4, all tools agreed on 3 being the correct metric value.

- **Class to Leaf Depth (CLD)** With this metric we calculate the longest path from a class to a leaf in the inheritance hierarchy. For example the *Object* class has a depth of 2, the distance from its furthest leaf, the *Tag* class. Whereas the *RFIDObject* has a depth of 1 to reach its furthest leaf. These distances are then summed for each class in the application giving a total CLD value of 3.

All of the UML CASE tools agreed that the CLD was 3 with one exception again, the Enterprise Architect tool.

- **Methods Inherited (MI)** This metric is the sum of the number of methods inherited by each class. This is calculated as the sum of *Number of Methods* taken over all ancestor classes of the class [13, 17].

The *RFIDObject* class in the *Oracle* application extends the *Object* class, therefore it inherits the two methods in the *Object* class. The *Tag* class extends the *RFIDObject* class, so it inherits the one method in the *RFIDObject* class and it also inherits the two methods from the *Object* class. The MI metric value is thus 5, and all of the tools reported this value.

- **Variables Inherited (VI)** This is the sum of the number of variables each class in the application has inherited. It works in much the same manner as MI. The *Oracle* application produces a metric value of 3 and all the UML CASE tools agree on this figure.

Inheritance Metrics Summary As the table, 3 shows, the UML CASE tools were in agreement most of the time, with the exception of Enterprise Architect, which frequently over rated the metric value by 1.

5. Analysis of Real World Programs

While the results for the Oracle class diagram show some differences in the metric values over the UML tools, it is not clear whether these are significant. In particular, it is important to know how the differences reflected in the previous section impact the analysis of larger programs.

To this end, we have assembled a test suite of "real-world" Java programs, and used the REM4j framework to extract class diagrams and calculate metrics. In this section we examine the results of comparing the accuracy of the eleven UML tools.

5.1. Control Charts

When using real-world programs, we have no oracular value for the metrics, and so we use peer evaluation to rank the UML tools. That is, we assume the collective judgement produced by the tools to be a “correct” answer, and judge each tool in this context. To this end, we adopt the *control chart* or *Shewhart chart*, which is a chart with five horizontal lines running across it. We use the threshold values of Lanza and Marinescu [14], and define the *center line* as a middle line reflecting the mean value of the metric. The *upper* and *lower control limits* are then 1.5 standard deviations above or below this line, and define the boundaries of “trustworthy” results. A wider threshold is provided by the *upper* and *lower warning limits*, which are 1 standard deviation above or below the center line, and flag values that require further investigation.

5.2. Java Application Selection

A selection of Java applications was chosen for this evaluation. Some were drawn from the DaCapo benchmark suite [5] while other applications were chosen from sourceforge.net for diversity. The programs are:

antlr	generates a parser and lexical analyzer
eje	a simple editor for Java
fop	renders pages to a specified output e.g. PDF
hsqldb	a database written purely in Java
jameleon	an automated test framework written in Java
java2d	is a java 2d graphics package
jolden	a benchmark test application written in Java
junit	a well-known framework to write repeatable tests
pcj	a set of collection classes for primitive data types
pmd	analyzes Java source code for potential problems
xalan	an XSLT processor for transforming XML

We refer to these Java applications collectively as the *test suite* from now on.

5.3. Metric Capture

Table 4 summarises the pass/fail results for each tool when run over the test suite. The top row in Table 4 lists the 11 reverse engineering tools under study, and the leftmost column lists the 11 Java applications in the test suite. Each cell records either “P” for pass or “F” for fail, indicating whether or not the UML tool exported valid XMI for this benchmark program. For example, we can see that the Argo tool, represented by the column labelled “AR”, exported valid XMI for all benchmark programs other than `hsqldb` and `pcj`.

The bottom row of Table 4 summarises the results for each Reverse Engineering tool, by recording the number of benchmark programs that passed. From this column, we can see that 7 of the 11 tools failed for at least one benchmark program.

5.4. Results Per Tool

Each tool is run over 11 test programs and then evaluated with 12 metrics, giving a total of 132 potential metric values.

	AR	MD	BO	MM	VP	JU	EA	UM	ES	IC	PO
antlr	P	P	P	P	P	P	F	P	P	P	P
eje	P	P	P	P	P	P	P	P	P	P	P
fop	P	P	P	P	P	P	P	P	P	P	P
hsqldb	F	P	P	P	P	P	P	P	P	P	P
jameleon	P	P	P	P	P	P	P	P	P	P	P
java2d	P	P	P	P	P	P	P	P	P	P	P
jolden	P	P	P	P	P	P	P	F	P	P	P
junit	P	P	P	P	P	P	P	P	P	P	P
pcj	F	P	F	P	P	P	F	P	P	P	F
pmd	P	P	P	P	P	P	P	P	P	F	P
xalan	P	P	F	P	P	F	P	P	P	P	P
ΣP	9	11	9	11	11	10	9	10	11	10	10

Table 4. This table lists the test suite applications in the left column and the UML tools in the top row. Each cell records the production of valid XMI, indicating either “P" for pass or “F" for fail.

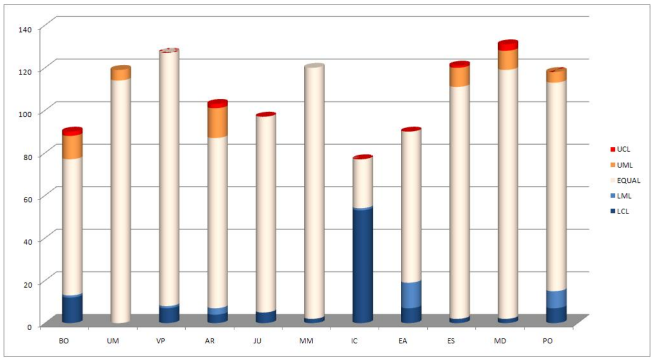

Figure 2. For each UML tool on the horizontal axis, this chart shows their success measured in terms of the metric values on the vertical axis

Figure 2 sums up our findings on the eleven reverse engineering tools. In this figure, deep blue represents metric values that are under the lower control limit while deep red shows metric values that are above the upper control limit. Metric values that are shaded light blue are in the lower warning zone while metrics that are shaded orange are in the upper warning zone. The white-coloured section represents the number of metric values within the warning limits. Thus, for each UML tool, the larger the white shading in its bar in Figure 2, the greater the level of reliability we ascribe to the tool. The different height of the bars in Figure 2 reflects the different pass/fail results as shown in Table 4.

- **Bouml** Of the 9 applications Bouml produced an output for, it failed to capture the IC and the II metric. There was a trend of Bouml underestimating size metrics, while overestimating inheritance metrics. The MI metric is the only metric that Bouml produced that was completely accurate, with the CLD being slightly overestimated but acceptable.

- **UModel** UModel showed a trend of being correct a large proportion of the time. UModel failed to capture one metric NoG for one application. None of UModel's metric values were outliers, with only one metric value being in the warning zone.
- **Visual Paradigm** Visual Paradigm was accurate with size metrics but failed to capture metrics for 5 of the inheritance metrics and reported metric values below the lower control level for 3 inheritance metrics.
- **ArgoUML** ArgoUML has both under and over estimated many of the metrics, with a clear trend showing, the tool under-estimating the size metrics while over-estimating the inheritance metrics.
- **Metamill** Metamill failed to capture the II metric for all the applications, it also failed to capture the IC metric for one application. Out of 132 possible metric values, Metamill captured 120, or 91% of the metrics.
- **Enterprise Architect** Enterprise Architect failed to capture IC and II metrics for any of the applications. Investigating the results further shows that Enterprise Architect consistently under-estimates all the metrics.
- **ESSModel** ESSModel both over and under estimated metrics for a variety of different metrics, there was no noticeable trend.
- **Magicdraw** We can see that Magicdraw over and under reported the IC metric. The majority of the tools struggled to report this correctly, with many tools failing to capture it at all. In total Magicdraw correctly reported 117, or 89%, of the metric values.
- **Poseidon** Out of the 10 applications Poseidon did report metrics for, it failed to capture 2 metrics for one application, so in total Poseidon captured 119 metric values. For approximately half the metrics, Poseidon under reported them, with it only over reporting for the II metric.
- **Jude** Jude failed to capture the II metric but, of the remaining metrics, only two reported values in outside the Control Limits.
- **Ideogramic UML** Ideogramic failed to capture the II and IC metric. In the majority of metrics Ideogramic UML failed to capture a metric for each tool, in fact it only captured all the metrics values for a third of the metrics. In total it captured 23 metrics (17%) that were deemed to be correct.

6. Conclusions

Figure 2 shows Visual Paradigm to be the most reliable tool since it reported metric values that were accurate 90.15% of the time. It was closely followed by Metamill, Magicdraw and UModel. The worst performing tool was the Ideogramic UML tool, reporting just 17% of the metric values correctly.

Summary REM4j is an automation framework that evaluates a UML CASE tool's ability to reverse engineer and export valid XMI for class diagrams. It has been used to evaluate 11 UML tools using software metrics over a variety of input programs and clearly highlighted differences in the results obtained.

References

[1] L. A. Barowski and J. H. Cross II. Extraction and use of class dependency information for Java. In *9th Working Conf. on Reverse Engineering*, 2002.

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

[3] D. Cooper, B. Khoo, B. R. von Konsky, and M. Robey. Java implementation verification using reverse engineering. In *27th Australasian Conf. on Computer Science*, pages 203–211, 2004.

[4] T. R. Dean, A. J. Malton, and R. Holt. Union schemas as a basis for a C++ extractor. In *8th Working Conf. on Reverse Engineering*, page 59, 2001.

[5] S. M. B. et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *21st Conf. on Object-oriented programming systems, languages, and applications*, pages 169–190, 2006.

[6] R. Ferenc, F. Magyar, A. Beszédes, A. Kiss, and M. Tarkiainen. Columbus - tool for reverse engineering large object oriented software systems. In *7th Symposium on Programming Languages and Software Tools*, pages 16–27, 2001.

[7] J. Jiang and T. Systä. Exploring differences in exchange formats - tool support and case studies. In *7th European Conf. on Software Maintenance and Reengineering*, page 389, 2003.

[8] R. Kazman and S. J. Carriére. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Eng.*, 6(2):107–138, 1999.

[9] M. Keschenau. Reverse engineering of UML specifications from Java programs. In *19th Conf. on Object-oriented programming systems, languages, and applications*, 2004.

[10] C. Kobryn. UML 2001: a standardization odyssey. *Commun. ACM*, 42(10):29–37, 1999.

[11] R. Kollmann and M. Gogolla. Re-documentation of Java with UML class diagrams. In *7th Reengineering Forum, Reengineering Week*, pages 41–48, 2000.

[12] R. Kollmann and M. Gogolla. Metric-based selective representation of UML diagrams. In *6th European Conf. on Software Maintenance and Reengineering*, page 89, 2002.

[13] M. Lorenz and J. Kidd. *Object-oriented Software Metrics*. Prentice Hall, 1994.

[14] R. Marinescu and M. Lanza. *Object-oriented Metrics in Practice*. Springer, 2006.

[15] M.-A. D. Storey, K. Wong, and H. A. Müller. Rigi: a visualization environment for reverse engineering. In *19th Intl. Conf. on Software Engineering*, pages 606–607, 1997.

[16] A. Sutton and J. I. Maletic. Mappings for accurately reverse engineering UML class models from C++. In *12th Working Conf. on Reverse Engineering*, pages 175–184, 2005.

[17] J. Wüst. SDMetrics, 2006. http://www.sdmetrics.com.