A Qualitative Evaluation of a Software Development and Re-Engineering Project

Thomas Panas Rüdiger Lincke Jonas Lundberg Welf Löwe

Software Technology Group
MSI, University of Växjö, Sweden
{Thomas.Panas|Rudiger.Lincke|Jonas.Lundberg|Welf.Lowe}@msi.vxu.se

Abstract

The VizzAnalyzer is a framework for analyses and visualizations of software. It has been developed over years, to a great deal by students and PhD students. In between it has been re-engineered to improve the software quality. In this paper, we publish the results of the software quality measurements over different versions of the VizzAnalyzer framework with well established quality metrics. Some metrics uncover qualities we were aiming at in our re-engineering, e.g. maintainability, some others uncover qualities we were deliberately ignoring or did not even think about, e.g. good "object-orientedness". Our measurements validate our expectation: the former metrics significantly improve over the versions whereas the latter contain positive as well as negative surprises.

1. Introduction

Software maintenance has become important for today's software industry in order to prolong the lifetime of software products. However, maintaining a complex software over the years is not a simple task since many problems might occur during its lifetime. For example, the original developers may have left the company, the documentation and original design documents may be no longer available or updated, the size and complexity may have grown, bug fixes and new functionality might have increases beyond a comprehensible level.

These problems do not only appear in industrial projects, but are sometimes even worse in university projects. Student developers increase problems related to software quality. Students enter and leave projects frequently. Hence, expert developers are the exception. Furthermore, students are less trained and experienced and, thus, design faults are frequent. The VizzAnalyzer [1, 2], a tool that has been developed at Växjö University over the last two years now, is such a student project.

In this paper, we investigate the source code quality of the VizzAnalyzer, a source code extraction, analysis and visualization framework. For this, we apply well established metrics in order to measure the systems quality w.r.t. maintainability and other object-orientation qualities. We investigate six versions developed over the last two years.

In the next section, we document the VizzAnalyzer, its purpose and development history for the last two years. In addition, we describe the mainly intuitive development and re-engineering decisions for some major release versions over the years. In Section 3, we describe the metrics used to justify our intuitive development and re-engineering decisions. In Section 4, we introduce how to interpret the metrics in the context of maintenance and good object-oriented design in general. Section 5 presents our analysis results of applying the described metrics on our own tool. A discussion of the results follows in Section 6. Section 7 concludes the results and shows directions of future work.

2. The VizzAnalyzer

The VizzAnalyzer is a framework designed to aid programmers in software engineering activities like maintenance and re-engineering. It allows to integrate different reverse-engineering tools, i.e. software tools for program analysis and/or visualization can be plugged into the framework. With the plug-ins at hand, the VizzAnalyzer framework allows to interactively and iteratively retrieve program information, focus, analyze, and visualize it.

As depicted in Figure 1, the VizzAnalyzer framework connects different in-house and external tools via wrappers, where syntactical and semantical data adaptations are defined. For more information about architecture, design and functionality, we refer to [2]. Currently, we have extended the VizzAnalyzer with the following external reverse-engineering tools:

Recoder [3] is a Java framework for source code meta programming aiming at delivering an infrastructure for Java analysis and transformation tools. Recoder is used for

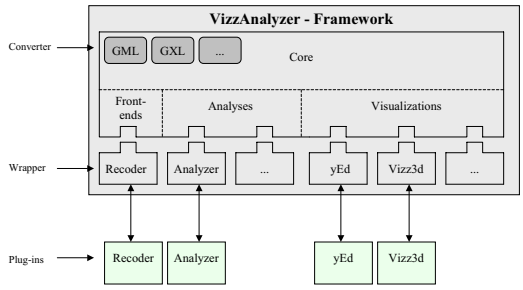


Figure 1. VizzAnalyzer Framework

source code information extraction. We have extended this API with our own data extraction client.

CrocoPat [4] manipulates relations of any arity. Its query and manipulation language is based on first-order predicate calculus. CrocoPat is used for program analysis.

yEd [5] is a Java graph editor that can be used to generate drawings and apply automatic layouts to all kinds of diagrams and networks. It is used for program visualization. Figure 16, e.g., is created with yEd.

Excel is used for statistic analyses on metrics and their visualizations. All diagrams are created this way.

WilmaScope [6] is a Java3D application which creates real time 3d animations of dynamic graph structures. It is used for program visualization.

Our own reverse-engineering plug-ins include:

Vizz3D, a 3D visualization framework, allowing the illustration of program information. Various layout algorithms can be added at run-time allowing visual complexity reductions. Bindings specify the mapping of metric results with visual properties (such as height, width, type, color etc.) in order to emphasize certain aspects of the system.

Analyzer is a analysis component, which was used to calculate all metrics we discuss in Section 3. Further, it allows to perform focusing on program information, i.e. aggregation, filtering and merge of information. Moreover, the analyzer allows to perform advanced analysis, such as architecture recovery.

The birth of the VizzAnalyzer is dated to July, 2000, when it was announced as a thesis project at Karlsruhe University, Germany. In that work, a first version of the analysis component was added to an algorithm animation tool named VizzEditor. From that time on, many students participated in its development and the size and complexity of the software grew rapidly.

In September 2002 the responsibility for the VizzAnalyzer (we refer to this version as VA^{0209} from here on) was moved to Växjö University with all its features (and problems). One of the major problems was a very high (intuitive) complexity of the tool. The reason for that was that the students at Karlsruhe University had repeatedly tried to add their own contributions to the tool by making as few changes in the original code as possible. At this stage, the VizzAnalyzer was no more maintainable, especially, not by a new development and maintenance team. The only way out was to re-engineer the entire tool.

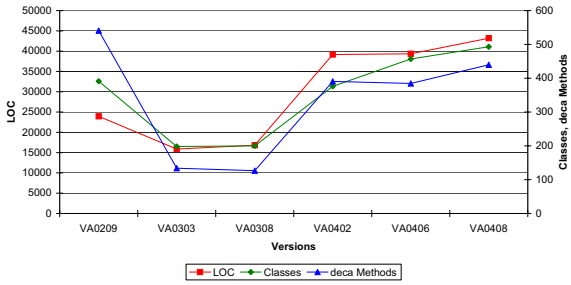


Figure 2. The Size of the VA versions

Intuitively, the re-engineering resulted in a less complex system where about $1/3$ of the original code was removed (VA^{0303}). The results of our re-engineering are documented in [7]. Figure 2 shows how the system size has changed over time. It compares Lines of Code (LOC), number of methods and number of classes. Note that the reduction of number of methods is around the factor 5. Intuitively, this leads to the assumption that the internal code quality might have been improved between the versions VA^{0209} and VA^{0303}. We will investigate this hypothesis in the next section.

With the version VA^{0303} the refactoring of the essential components began. What was one huge intervened system before, was refactored in an initial attempt to the following components: core (program entry points and actual framework), analysis (program information extraction and program analysis), visualization and a graph package. Our hypothesis here is that the system should be less coupled. We will investigate our intuitive observation with actual metric measurements in the next section.

The development on the VizzAnalyzer continued during the period of August 2003 (VA^{0308}) to February 2004 (VA^{0402}). The graph package became a separate component through a course initiative, today known as *Grail*. The visualization component evolved through many theses and student projects to a separate 3D visualization tool (Vizz3D). At the same time, the VizzAnalyzer was re-engineered once more towards a framework with plug-in functionality. The goal here was to facilitate the integration of other reverse engineering tools. In order to achieve this, all components of the VizzAnalyzer were decoupled into separate modules:

Core: program entry point and the actual framework, **Recoder**: information extraction (a plug-in module), **Analyzer**: program analysis engine providing metrics computations, pattern detection, etc. (a plug-in module), **Vizz3D**: visualization (a plug-in module), and **Grail**: the database of the framework and orthogonal to all the other components (not a plug-in). The hypothesis is again that the system should be less coupled, separating intended components and increasing therefore the systems maintainability.

From version VA^{0402} to version VA^{0406} Vizz3D was refactored in order to allow for better reusability of common classes and interfaces between the Java3D and OpenGL implementations. Figure 2 shows that although the number of methods has increased between those two versions, the number of LOC and number of classes remained almost constant. This indicates our re-engineering efforts. Finally, towards version VA^{0408} the VizzAnalyzer gained size again due to the implementation of the metrics used for this paper.

3. Metrics

In this section, we give a brief presentation of the metrics used to measure the quality of the different VizzAnalyzer versions. The packages being measured are the Core, the Analyzer, the Recoder (our connection to the Java frontend), Vizz3D, and Grail. The choice was simple, since the original system VA^{0209} consisted of exactly those packages. However, at that time the packages were not consistent with system components and intervened with each other. Throughout the years, the VizzAnalyzer was re-engineered to a system with separated components along the above package boundaries. Hence, the packages above have survived over the years.

The program information for the metric computations, the basic program model, is represented in a number of graph structures. By default, the VizzAnalyzer computes two graphs: a class hierarchy graph containing class and interface (nodes) and extends and implements relations (edges), and a call graph containing methods, constructors, fields and initialization blocks (nodes) and their mutual accesses (edges). This graph is derived using the Rapid Type Analysis algorithm [8].

By configuration, other program information can be retrieved, as well. Then the VizzAnalyzer computes further graphs. Virtually any Abstract Syntax Tree node can be a node in such a graph; any relation from the static semantic analysis can be an edge. The latter include type-, field-, method-, and constructor-references, extends and implements relations, and package containment. For each metric applied, we configure a tailored program information graph. These graphs are described below together with the respective metrics.

The quality model we used as a basis for re-engineering tasks was targeted at maintainability. Other object-orientated qualities were considered secondary. To measure qualities of a system, we have implemented well established metrics from literature. Even that we targeted at maintainability, we measured metrics directly supporting maintainability but other standard metrics, as well. What follows is a short description of the metrics we selected:

Weighted Method Count (WMC) [9] computes the complexity of the methods of a class and is also a measure for the complexity of a class. It gives a good idea about how much effort is required to develop and maintain a class itself. The weighed method count was implemented according to the suggestion by Li and Henry [10], where the methods are weighted according to McCabe's Cyclomatic Complexity Metric. Our implementation counts the possible execution branches in a method for the branching statements: if, for, while, do. It is assumed that each branch has the same complexity/weight. McCabe's Cyclomatic Complexity is a measure on rational scale. This metric is computed on a program information graph containing class, method, while, until, for, and if nodes (AST nodes) and the syntactical contains relation (edges).

Depth of Inheritance Tree (DIT) [11] is the length of the path from a class to the root class of the inheritance tree. The deeper a class is in the hierarchy, the higher is its potential to reuse inherited methods. However, if the hierarchy is too deep, this could be an indication of misuse of inheritance. The DIT metric can not absolutely indicate good code quality. It is rather used to allocate outliers for bad design. In our implementation, the DIT values are calculated for each class and interface considering implements and extends relations. It expresses the longest distance to the root of the hierarchy. This measure is on an absolute scale ranging from 0 to the maximum depth of the inheritance tree; it is computed on the class hierarchy graph (computed by default by the VizzAnalyzer).

Number of Children (NOC) [11] represents the number of immediate subclasses of a class in a class hierarchy. More children indicate better reuse, since inheritance is a form of reuse. However, if a class has a large number of children, it may be the case of misuse of subclassing. In [9], it is suggested that classes high in the hierarchy should have more subclasses than those lower down. In our implementation the NOC is calculated for each class and interface by counting classes or interfaces extending or implementing it directly (number of children). The values are integer values ranging from 0 for no children to the maximum number of children a class has on an absolute scale. This metric is computed on the class hierarchy graph (computed by default).

Data Abstraction Coupling (DAC) [11] represents the number of references to abstract data types (ADT's) defined in another class. The higher the number is, the more com-

plex is the coupling of a class with other classes. Counted are the fields defined in a class referencing a user defined type, not a primitive, language or library defined type. Inherited fields are not counted. The DAC is calculated for each class and interface. The values are integer values ranging from 0 (indication no other ADT is referenced) to a maximum number on an absolute scale. This metric is computed on a program information graph containing class and field nodes (AST nodes) as well as contains and type reference relations (edges).

Package Data Abstraction Coupling (PDAC) lifts the DAC to package level. It represents the number of ADT's referred from one class to another crossing package boundaries. The higher the number is, the tighter is the coupling of two packages. Counted are the fields defined in classes within a base package (top-level package), referencing a user defined type in another base package. Base packages that logically belong the same component are considered as one package. The values are integers ranging from 0 (indicating that no other ADT is referenced) to a maximum number on an absolute scale. This metric is computed on a package graph containing package declarations, class and field nodes (AST nodes) as well as contains and type reference relations (edges).

Change Dependency Between Classes (CDBC) [12] determines the potential amount of follow-up work to be done in a client class when a server class is modified. It indicates the strength of coupling. The goal is to keep changes in the system local by reducing the system coupling. Lower values indicate lower coupling and, thus, a better stability in the system. The CDBC value is defined between a client and a server class as the number of methods which need to be (potentially) changed in the client if a server changes. The CDBC value is between 0 and the count of methods in the class. We compute the average CDBC value of each (client) class over all (server) classes it is directly connected with. The scale is rational. This metric is computed on a program information graph containing class, method, constructor, field, and initialization block nodes (AST nodes) and (besides the syntactical containment) the type reference, extends, and implements relations (edges).

Tight Class Cohesion (TCC) [11] is the relative number of directly connected methods in a class. TCC indicates the degree of connectivity between visible methods in a class. Given the number n of local methods (excluding inherited methods), TCC is defined as ndp over np with $np = \frac{n \times (n-1)}{2}$ the possible pairs of these methods and ndp the number of method pairs actually calling another. Note, that this is a slight deviation from the TCC definition in [11]. The TCC for a class is 0 if $np = 0$. The resulting values range from 0.0 to 1.0 on an rational scale. Higher values indicate better cohesion of the classes. Low values indicate that a class has a low cohesion. This metric is computed on the call graph (computed by default).

Tight Package Cohesion (TPC) lifts the TCC metric to package level and retrieves the relative number of classes directly connected by calls in a package. TPC indicates the degree of connectivity between classes within a package. Given the number n of package local classes, TPC is defined as ndp over np with $np = \frac{n \times (n-1)}{2}$ the possible pairs of these classes and ndp the number of class pairs actually calling another. The TPC for a package is 0 if $np = 0$. The TPC metric is calculated similar to the PDAC on base packages containing components. The resulting values range from 0.0 to 1.0 on an rational scale. Higher values indicate higher cohesion of the package. This metric is computed on a call graph with package declarations.

Lack of Documentation (LOD) measures the amount of undocumented declarations per class (counted are the class declaration itself and the method declarations, but not field declarations). Only JavaDoc style documentation is taken into account. Documentation within methods is ignored. Only the syntax of the comments is parsed, not the semantics. The LOD value is calculated for each class or interface as the number of undocumented declarations. A LOD of 0 indicates that all possible entities are documented; a higher value indicates the lack of documentation. The LOD is on an absolute scale. A problem with this metric is apparently automatic JavaDoc generation. Tools supporting this feature prevent the exact measurement of undocumented declarations. Nevertheless, this problem merely implies that we cannot discover all undocumented code, i.e. we cannot recover all problems inherent.

All metrics, except LOD, PDAC, and TPC, are well defined and validated in the literature. Especially, [10, 11, 13, 14] give a detailed description of these metrics and a validation of their use and acceptance within the maintenance and re-engineering community. The results received from these metrics give a good idea about the complexity of the system in respect to the classes themselves, their hierarchical arrangement, their coupling and their cohesion.

PDAC and TPC are natural extensions of the corresponding metrics DAC and TCC to package level. The LOD metric is an exception. We could not find any similar specification in the literature. The closest specification was found in [15, 16], where the density of comments is calculated by the amount of comment lines divided by the amount of lines of code.

4. Interpretation of the Metrics

The metrics described in Chapter 3 help to measure different aspects of maintainability as well as other qualities of a system. In the following, we discuss maintainability and those other qualities on system design, class design and complexity level.

4.1 System Level Design

On this level we look at the design of the *system architecture* and the system's *inheritance structure*.

The system architecture is defined by the system's components and their interactions. Components are interacting sets of classes and smaller components, i.e. the notion of a component is recursive. Interactions include method calls and field accesses.

An architecture is considered well maintainable, if its components have a low external coupling to other components. Then they are easy to change without affecting other components. Not directly related to maintainability but also reported as desirable is a high internal cohesion; otherwise one might consider splitting the component.

Packages could contain components. They could also be designed after other concerns crosscutting components. For example, one could group all interfaces in a package. Only the former kind of packages should show high cohesion and low coupling to others. We applied TPC and PDAC only to those packages that are intended components. For those packages, low cohesion (TPC) and high coupling (PDAC) can be considered as design faults.

The inheritance structure of a system is defined by implements and extends relations of classes and interfaces. Such a structure is expected to follow a few general design rules.

Chains in the inheritance structure of length one are considered a sign of unnecessary abstraction. Moreover, they should neither be too deep nor too wide, i.e. extreme deviations from the average should be inspected. These rules are not directly related to maintainability.

4.2 Class Level Design

In the same way that package cohesion and coupling indicate good design of components, they are applicable to assess the design of individual classes.

Since, high cohesion and low coupling are desired, a correlation between DAC and TCC is essential to indicate outliers. The design of a system should be changed if the correlation indicates classes with a high DAC (coupling) and low TCC (cohesion) value. However, only the DAC is directly related to maintainability: classes with high coupling to others cannot be understood in isolation. This, however, is essential to maintain the classes.

Another design problem can be found, when correlating classes that have a huge affect on other classes upon change (CDBC); even worse, when those classes are bad documented (LOD). Therefore it is wise to correlate the CDBC and LOD metric in a diagram to find such outliers.

Metric	Version	Minimum	Average	Maximum	Norm
WMC	VA0209	0	17.61	152	1.30
	VA0303	0	10.53	152	0.77
	VA0308	0	12.10	276	0.89
	VA0402	0	15.85	178	1.17
	VA0406	0	12.41	182	0.91
	VA0408	0	13.05	223	0.96
DIT	VA0209	0	1.4987	8	1.73
	VA0303	0	1.0808	5	1.25
	VA0308	0	0.5950	3	0.69
	VA0402	0	0.7207	5	0.83
	VA0406	0	0.6368	5	0.74
	VA0408	0	0.6552	5	0.76
NOC	VA0209	0	0.5908	10	1.17
	VA0303	0	0.5606	11	1.11
	VA0308	0	0.4200	19	0.84
	VA0402	0	0.5053	20	1.00
	VA0406	0	0.4595	12	0.91
	VA0408	0	0.4807	12	0.96
DAC	VA0209	0	0.6087	9	0.92
	VA0303	0	0.4394	20	0.66
	VA0308	0	0.4500	5	0.68
	VA0402	0	0.6117	8	0.92
	VA0406	0	0.9453	27	1.43
	VA0408	0	0.9168	32	1.38
CDBC	VA0209	0	1.5070	59.00	1.11
	VA0303	0	0.7683	11.00	0.57
	VA0308	0	1.0871	9.08	0.80
	VA0402	0	1.7600	28.00	1.29
	VA0406	0	1.4340	24.00	1.05
	VA0408	0	1.6010	25.60	1.18
TCC	VA0209	0	0.0353	1.00	0.81
	VA0303	0	0.0262	1.00	0.60
	VA0308	0	0.0520	1.00	1.19
	VA0402	0	0.0374	1.00	0.86
	VA0406	0	0.0571	1.00	1.31
	VA0408	0	0.0542	1.00	1.24
LOD	VA0209	0	13.6573	72	1.95
	VA0303	0	3.3409	51	0.48
	VA0308	0	4.4800	46	0.64
	VA0402	0	6.7394	44	0.96
	VA0406	0	6.4617	72	0.92
	VA0408	0	7.2495	61	1.04
TPC	VA0209	0	0.2479	1.000	2.09
	VA0303	0	0.0685	0.152	0.58
	VA0308	0	0.1354	0.300	1.14
	VA0402	0	0.1024	0.227	0.86
	VA0406	0	0.0893	0.236	0.75
	VA0408	0	0.0681	0.154	0.57
PDAC	VA0209	0	65.80	217	1.04
	VA0303	0	140.80	393	2.23
	VA0308	0	105.00	200	1.66
	VA0402	0	33.25	76	0.53
	VA0406	0	22.50	60	0.36
	VA0408	0	11.75	47	0.19

Figure 3. Metric Results

4.3 Complexity

Classes that are both highly complex and large are critical when it comes to understanding and maintenance. Since large complex classes are potentially problematic with respect to understandability and maintainability, the WMC and LOC metric should be correlated to find such outliers.

5. Analysis

Applying the quality metrics discussed above on the different versions of the VizzAnalyzer, we retrieve the information depicted in Figure 3. We followed the method of the FAMOOS consortium reported [11] and measured for each metric the minimum, the maximum, and the average over all occurrences. The average allows to compare differ-

ent versions with another, since it takes the number of the analyzed entities (e.g. classes, methods, etc.) into account.

In order to relate different metrics of the same version the metrics are normalized in the way, that they vary around the value 1. This normalization concept is similar to [17]. The normalization $Norm$ is defined as:

$$Norm = \frac{Average}{\bar{A}} \; with \; \bar{A} = \frac{\sum Average}{\#\; versions}$$

5.1 System Design

Much effort for improving the VizzAnalyzer's maintainability was spend on the decoupling of components. Figure 4 shows our clear trend in the corresponding metric PDAC. Note that the edges connecting measure points are just an aid to easier follow the trend between the values.

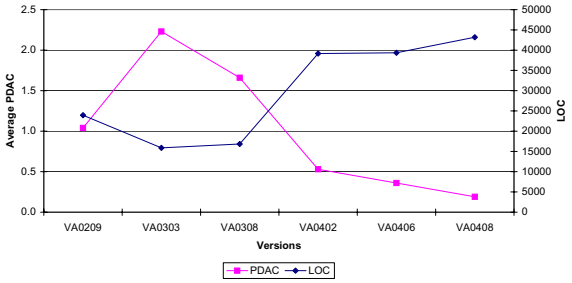


Figure 4. PDAC Metric

Even with increasing system size (the LOC increase throughout the versions), the coupling decreases between version VA^{0303} and VA^{0408} (lower values indicate better coupling). The only exception occurs during our first re-engineering effort from version VA^{0209} to VA^{0303}, where we aimed at throwing away useless wrappers. Then LOC decreased and the coupling between components somehow increased.

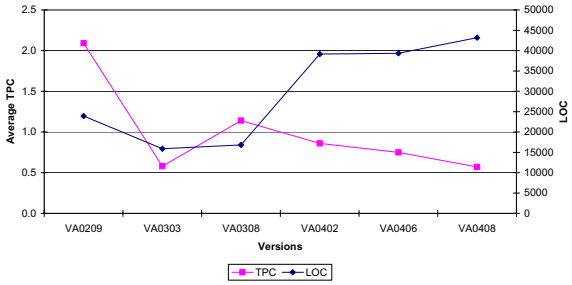


Figure 5. TPC Metric

Low coupling does not necessarily induce high cohesion. In Figure 5, we see that the trend for the cohesion metric is bad (remember, higher values indicate better cohesion). However, cohesion is not a maintainability metric and, hence, not a goal in our re-engineering efforts.

However, the negative trend of the TPC metric does not explicitly indicate a bad result. Comparing VA^{0402} to VA^{0406} shows an TPC average decrease of 0.0131. The decrease can be explained by our refactoring efforts. Increasing the amount of classes while retaining the amount of methods in a package, results in an increased np, while the ndp remains constant. This leads to a lower TPC.

Figure 6 shows the distribution of the DIT value. Notice that VA^{0209} has a high depth of inheritance (there exits many classes with a DIT of 4 and even 5), which indicates possibly useless abstractions. Our interpretation is that the system was highly unstable at that version and students enhancing the system preferred to use the available classes and inherit from them, rather than to modify them directly. After the first re-engineering, these abstractions have been reduced.

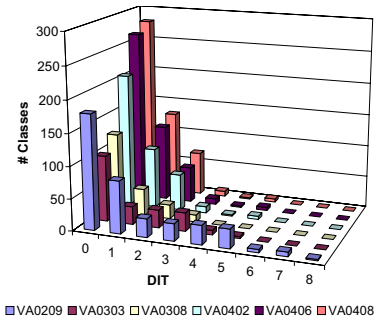


Figure 6. DIT Distribution

Classes with a NOC of one also indicate faults in the design of the inheritance relationship because they indicate unnecessary abstraction. They need to be inspected individually to verify whether they were meant to be extended for polymorphism in the future. Figure 7 shows that quite many classes with a NOC of one exist in all versions. This, again, is not a problem directly related to maintainability.

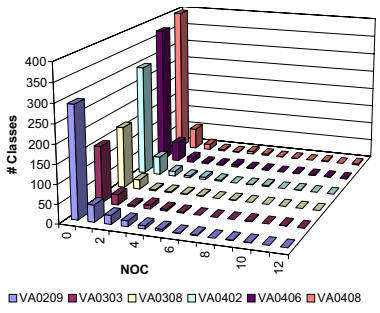


Figure 7. NOC Distribution

5.2 Class Design

Figure 8 shows that the coupling between the classes in each version has increased. Hence, the system has become harder to understand, or at least the individual classes thereof. Interesting to notice here is the increasing complexity between VA^{0402} and VA^{0406}. Although, that the size of the VizzAnalyzer was rather stable, the coupling increased, which was an unexpectedly negative result.

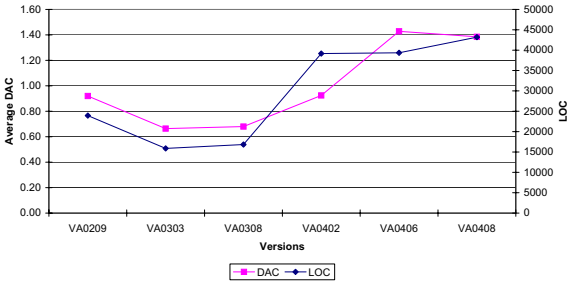


Figure 8. DAC Metric

The result of the TCC analysis is depicted in Figure 9. We had to realize that our TCC values are very poor in general. It becomes obvious that almost none of our classes were optimized for being cohesive (i.e. values close to 1). Cohesion on class level is not supporting maintainability directly.

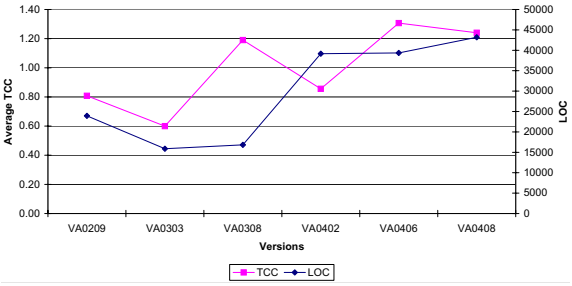


Figure 9. TCC Metric

The general low TCC (around 0.1, 0.2) can be explained as a result of our deviated implementation of the TCC metric. Our implementation allows a much higher number of possible method pairs, which is never reached in reality. Examining some classes with a very low TCC shows that these classes are well implemented in respect to cohesion, even so the results indicate a value of 0.1. Hence, rather our TCC deviation than the design of the classes should be considered inappropriate.

Figure 10 shows a correlation graph between the TCC and DAC values over the different VizzAnalyzer versions. Outliers, indicating low cohesion and high coupling, are to be found in the left upper corner of the figure. We can conclude that the amount of outliers is minimal. However, the general cohesion of the classes is poor.

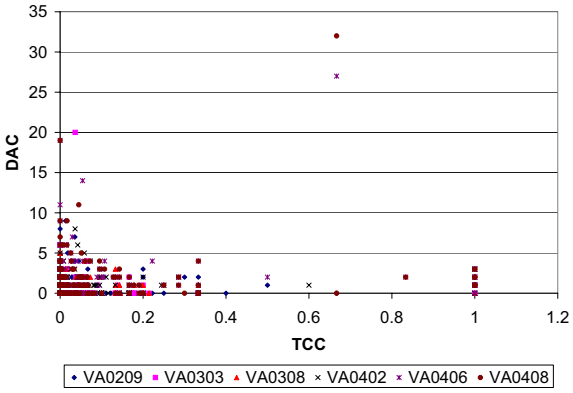


Figure 10. TCC/DAC Correlation

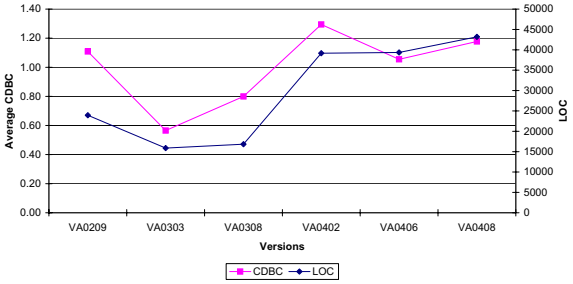


Figure 11. CDBC Metric

Figure 11 reveals that, while the system was re-engineered during the phases VA^{0209} to VA^{0303}, and VA^{0402} to VA^{0406}, the CDBC value decreased, i.e. the change dependency decreased and the maintainability increased. The final re-engineering step, and some others in between, increased the CDBC, i.e. decreased maintainability.

Maintenance becomes even harder, when the documentation of classes with a high CDBC is poor. Figure 12 shows the lack of documentation throughout the versions.

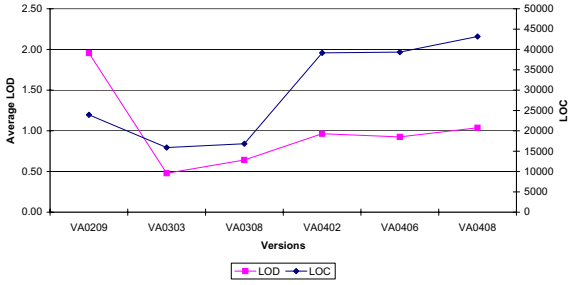


Figure 12. LOD Metric

Notice the high degree of documentation in version VA^{0303}, after the VizzAnalyzer was entirely re-engineered. Unfortunately, the documentation was slightly neglected

during further development. The figure maps our intuition (from a developer's point of view) and uncovers therefore no surprise.

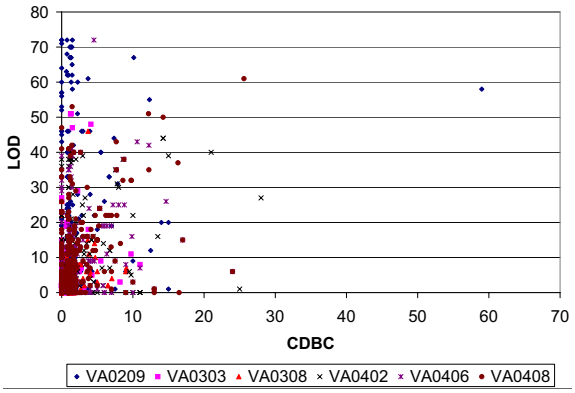


Figure 13. CDBC/LOD Correlation

A correlation graph between CDBC and LOD is shown in Figure 13. Quite a few outliers can be noticed with a high change dependency and poor documentation. The only outlier in the top right corner of the diagram comes from the very first version.

5.3 Complexity

Figure 14 shows that the first version of the VizzAnalyzer had a higher complexity than the following version VA^{0303}. This is an indication that our re-engineering efforts between those two versions were successful. However, while we continued to add functionality to the VizzAnalyzer again, the complexity increased again. This trend can be observed best between version VA^{0308} and VA^{0402}, where the code size has more than doubled.

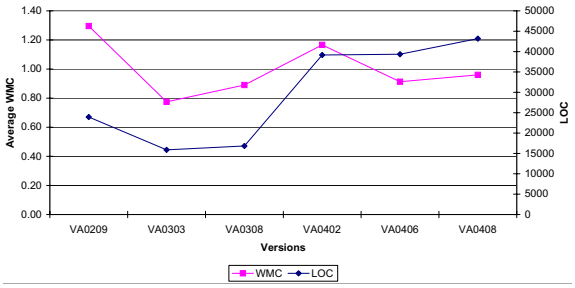


Figure 14. WMC Metric

From version VA^{0402} to VA^{0406} the complexity has decreased, while the code size was about the same. This can be explained as an re-structuring effort to decrease the class complexity of the newly added sources in the previous version.

Figure 15 shows a correlation graph between WMC and LOC. Outliers (in the top right quadrant of the diagram) indicating large and complex classes are frequent between the versions. However, for the current version only one outlier with over 2000 lines of code and a WMC value of 223 is visible. This class serves currently as an event listener and will be restructured in the next release.

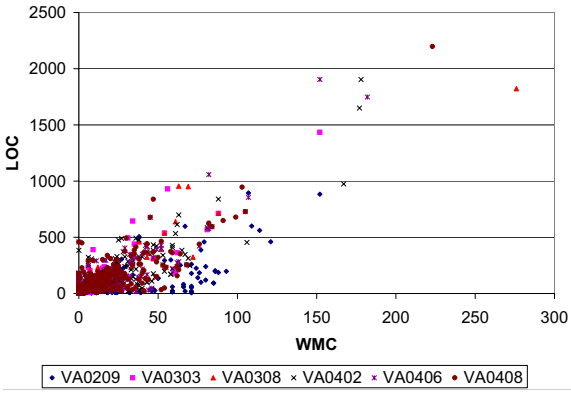


Figure 15. WMC/LOC Correlation

6. Discussion

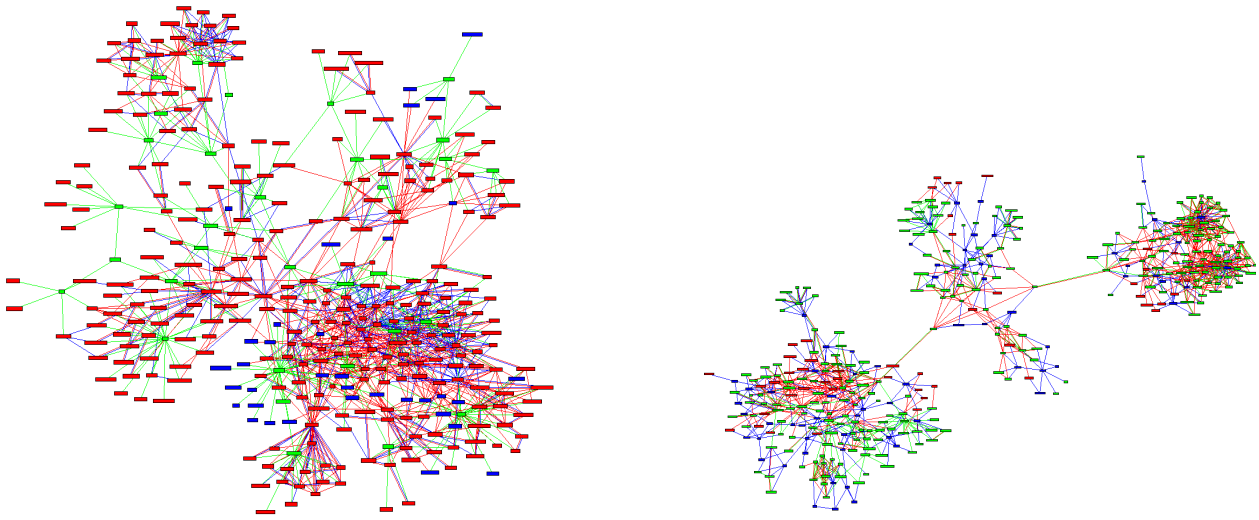
Maintainability was poor in version VA^{0209} of the VizzAnalyzer; this initial intuition could be objectified by our measurements. Improving this quality was, along with bug fixing and adding functionality, the major goal in our re-engineering efforts.

Figure 16 a) shows the VizzAnalyzer architecture of the VA^{0209} version; Figure 16 b) shows the architecture of version VA^{0408}. In both pictures, nodes are classes and interfaces, edges mutual usage (method and constructor calls, field accesses).

In version VA^{0209}, the system consisted of mainly one huge and highly unstructured component. Due to re-engineering, coupling of components, and thereby maintainability, has improved significantly in version VA^{0408}. Figure 16 at least gives that impression. In both drawings, a spring embedding layout algorithm tries to place coupled classes and interfaces together. Hence, the obvious impression of a lower coupling of components in version VA^{0408} compared to version VA^{0209} is not artificially induced by an unfortunate drawing of the latter.

The VizzAnalyzer consists of the components that correspond to the packages introduced before: the Core is illustrated in the middle-upper part, the Vizz3D to the left, the Recoder to the right and the Analyzer is adjacent to and below the Core. It is no mistake that the Analyzer has a higher coupling to the Core since it offers more than one service (focusing, metric and other high level analysis).

The PDAC metric objectively supports the illustrated trend in Figure 16 of a more decoupled system. This trend is also depicted in Figure 17. This figure contains also the



(a) (b)

Figure 16. VizzAnalyzer Architecture. (a) Version 0209 (b) Version 0408

trend of the other metrics related to maintainability that we observed. Besides PDAC, these metrics are WMC, CDBC, DAC, and LOD.

WMC indicating the complexity of methods and classes has decreased after the first re-engineering effort and than remains constant even though the system was growing.

The CDBC and DAC developments, indicating different coupling of classes, are not quite as positive. Improvements in decoupling happened rather in the large on component level than in the small on class level. However, the CDBC and DAC developments showed only a mild increase with increasing system size, which can be considered natural.

The LOD, i.e. the lack of documentation, indicates a clear improvement again. After a huge effort in the first re-engineering step, the degree of documentation is now stable with increasing system size. Altogether, maintainability has improved according to our metrics.

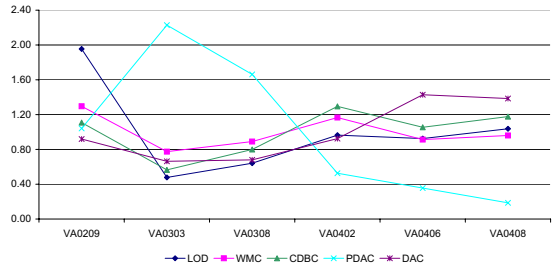


Figure 17. Maintainability Metrics at a Glance.

Metrics under investigation not directly coupled to maintainability but rather to generally good object-oriented programming style are DIT, NOC, TPC, and TCC. These metrics are shown once more in Figure 18.

The high DIT value of version VA^{0209} indicates a too extensive use of inheritance. This has improved over the versions. The NOC does neither turn too good nor bad. Low TCC and TPC values, respectively, indicate low class and package cohesion, respectively, which is considered bad object-oriented style. Even though our TCC increases over the versions, the distribution shows that almost all classes have a rather low cohesion. TPC indicates an unfortunate trend over the versions; it is decreasing while an increase would be desirable.

Altogether, there is no general positive trend in the object-oriented programming style according to our metrics. Again, the metric trends do not contain a surprise: one cannot expect to get a quality if it is not directly a goal in the re-engineering process.

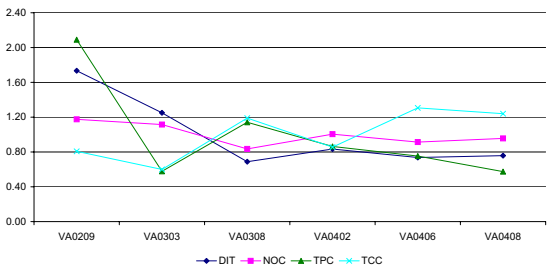


Figure 18. The Other Metrics at a Glance.

7. Conclusion

Over the last years, we re-engineered our tool VizzAnalyzer with the goal of improving the maintainability. We succeeded in these efforts. Our analyses have confirmed that VA^{0209} was not only intuitively hard to maintain, but also objectively. In version VA^{0408}, four main developers and a number of students maintain the four main components. They works to a large degree independent of another, which gives an idea of the system being now not only objectively easier to maintain, but also intuitively.

This paper describes experiments. We applied well-established metrics detecting maintainability (and other object-oriented programming qualities) to historical and current versions of the VizzAnalyzer framework. The trend of the maintainability metrics correlates to the positive experiences of the developers regarding system maintainability. In general, the experiments can be interpreted as a support of the appropriateness of these metrics. We found the PDAC, CDBC, WMC, LOD and DIT metrics very useful to measure the maintainability of the VizzAnalyzer itself.

The evaluation also brought up some new aspects. Low coupling between packages does not automatically go along with high cohesion within the packages. Moreover, the VizzAnalyzer is less coupled on package level now than in the first versions, which is supported by PDAC and, intuitively, by Figure 16. On class level, the metrics CDBC and DAC indicate a slightly higher coupling degree now. We understood that these metrics are rather reflecting the coupling in the small and not the coupling on architecture level.

The main quality goal in the re-engineering targeted at improved maintainability whereas other object-oriented programming qualities were considered secondary. It is therefore not surprising that the trend of these metrics is not that positive. One does not get these quality automatically when improving only maintainability.

Since we applied the VizzAnalyzer metrics on the framework itself, the paper shows two more results as a side effect: the VizzAnalyzer metrics are applicable in larger software projects and the tool itself has a decent internal quality.

Currently, we are evaluating software quality in industrial projects. This requires the implementation of further metrics, not all directed to maintainability. Our long term goal is to verify (or falsify) these metrics in experiments.

References

[1] VizzAnalyzer. `http://www.msi.vxu.se/~tps/VizzAnalyzer`, 2003.

[2] T. Panas, J. Lundberg, and W. Löwe. Reuse in Reverse Engineering. In *12th International Workshop on Reverse Engineering,Bari,Italy*, June 2004.

[3] A. Ludwig, R. Neumann, U. Aßmann, and D. Heuzeroth. Recoder homepage. `http://recoder.sf.net`, 2001.

[4] D. Beyer and C. Lewerentz. CrocoPat: Efficient Pattern Analysis in Object-Oriented Programs. In *11th International Workshop on Reverse Engineering,Portland,USA*, May 2003.

[5] yWorks. `http://www.yworks.com/en/products_yed_about.htm/`, 2004.

[6] Wilmascope. `http://www.wilmascope.org/`, 2004.

[7] R. Lincke. Development of a Graph Visualization Framework, Master Thesis. Växjö University, July 2003.

[8] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. *ACM SIGPLAN Notices*, 31(10):324–341, 1996.

[9] S.R. Chidamber and C.F. Kemerer. A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[10] W. Li and S. Henry. Maintenance Metrics for the Object Oriented Paradigm. In *IEEE Proc. of the 1st Int. Software Metrics Symposium*, pages 52–60, May 1993.

[11] H. Bär, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, T. Richner, M. Rieger, C. Riva, A. M. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, and J. Weisbrod. The FAMOOS Object-Oriented Reengineering Handbook. Technical report, Forschungszentrum Informatik, Karlsruhe, Software Composition Group, University of Berne, ESPRIT Program Project 21975, 1999.

[12] M. Hitz and B. Montazeri. Measuring Coupling in Object-Oriented Systems. *Object Currents*, 1(4), 1996.

[13] V.R. Basili, L. Briand, and W.L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. Technical report, Univ. of Maryland, 1995.

[14] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1996.

[15] F. Lanubile and G. Visaggio. Evaluating Empirical Models for the Detection of High-Risk Components: some lessons learned. In *Proc. of the Twentieth Annual Software Engineering Workshop, Goddard Space Flight Center, Greenbelt, Maryland*. Software Engineering Laboratory Series, SEL-95-004, 1995.

[16] refactorit. `http://www.refactorit.com/`, 2004.

[17] J.Bansiya and C.G.Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEE Transactions on Software Engineering*, 28(1), January 2002.