In Tibor Gyimóthy and Fernando Brito e Abreu, editors, Proc. 6th European Conference on Software Maintenance and Reengineering. IEEE, Los Alamitos, 2001

# Metric-Based Selective Representation of UML Diagrams

Ralf Kollmann & Martin Gogolla
University of Bremen
Department of Computer Science
PO Box 330440, D-28334 Bremen
{kollmann|gogolla}@informatik.uni-bremen.de

## Abstract

*UML diagrams are widely employed for modeling of object-oriented software systems. In addition to their application in forward engineering, it is also possible to use them for the redocumentation of existing programs. However, the inherent structure of UML diagrams, which consists of graphical as well as textual information, makes it difficult to read and oversee large diagrams generated from complex systems. The view on such diagrams can be compared with taking a look at a detailed map: the reader has to decide whether to read the fine details or to view the whole structure.*

*We present approaches to overcome this problem by using object-oriented metrics for program analysis and location of submodules in diagrams that belong together in terms of coupling. The shown techniques have been implemented into our reverse engineering tool Idea, which allows interactive selection and isolation of coherent regions of class diagrams.*

*Keywords: UML Class Diagrams, Software Metrics, Program Visualization, Reverse Engineering.*

## 1 Introduction

When using the Unified Modeling Language (UML) [23, 22] for visualization of reverse engineered systems, the resulting diagrams tend to show rather large amounts of information. The advantage of such high information content conveyed in UML diagrams comes not without problems: the provided level of detail and the characteristics of the UML to combine graphical and textual information result in voluminous diagrams that become difficult to oversee and read when displaying reverse engineered information from non-trivial software systems. When using CASE tools for displaying, the visualization suffers from a kind of "haziness": when zooming out far enough to see the *class context* i.e. the related structure of currently observed classes, detail information like role names or even class names cannot always be read. On the other hand when zooming in to be able to recognize details, the class context is likely to become concealed at least partially.

Approaches to solve this problem include a bird's eye view on diagrams for navigation [26], recovering a more abstract view on the system's architecture to conceal or discard implementation details [20, 14] or combination of abstraction and manual restructuring [15].

In this paper, we investigate the possibilities of using object-oriented metrics to solve the aforementioned problems with graphical representation using the UML. Metrics can be employed to draw the attention of developers to central or critical sections of an object-oriented design. When examining complex software systems, the numeral values produced by metrics become rather complex themselves. One way to overcome this problem is the combination of metrics with program visualization [25, 6].

In our approach, metrics are used for the creation of UML diagrams that show selected submodules from the reverse engineered architecture of software systems. We present an interactive approach that consists of several repetitive stages and uses metrics in different ways:

- Calculate metrics for an observed set of classes.
- Select prominent elements depending on the yielded metric values and determine the class context by calculating and combining metrics.
- Visualize and inspect the newly created sub-view on the architecture.

To understand the metric-based class context, the common detail level of UML class diagrams is not sufficient. We show the required information by means of dependencies extended by stereotypes. Additionally, we provide an information-stripped view on the class diagram that shows only the coupling between classes to provide an overview and understanding of the modules' connectivity.

The techniques presented here have been implemented and integrated into our analysis tool Idea, which supports reverse engineering and redocumentation of Java software on implementation and design level. Its functionality is extended into two directions: First, by enabling the user to understand complex systems in a bottom-up procedure by examining related submodules. Second, by providing additional information about the submodule architecture to help the user understand relations between classes that are not commonly conveyed by UML class diagrams.

This paper is organized as follows. In section 2, we discuss some basic concepts and present the object-oriented metrics employed in this paper. We then introduce an example software module in section 3, which will be used to demonstrate application of the metric-based selection techniques presented in section 4. The technical part is completed in section 5 with a presentation of our tool Idea. The paper ends with references to related work and a conclusion.

## 2 Background

The information contained in UML class diagrams is conveyed by means of graphical representations like squares and lines combined with text. The graphical part of the diagrams is well-suited for scaling, but the textual information is not. This means that looking at a class diagram from a greater distance still allows to recognize most parts of the class structure, but the text gets blurred (examples for class diagrams of different sizes are shown in figures 1 and 2). When reading large class diagrams at a detail level that allows reading the textual information, one can only examine parts of the diagram. However, as there are no inherent borders that divide class diagrams into logical parts, the user has to find groups of classes that belong together on his own. This means that he has to examine all connections between classes and then decide which ones are important and which are not. This weighting process can be taken from the user and be done at least partially automatically. In the following sections, we discuss requirements and metrics used to perform this procedure. In the scope of this work, we consider classes belonging together, if they are closely coupled, so we start with a short terminology.

### 2.1 Class Coupling

Our goal is to localize sets of classes with a tight connectivity, i.e. classes depending on each other's visibility. This kind of connectivity is often expressed by means of a classes' *coupling*. According to [2] as quoted by [18], coupling is defined as follows: *When one object depends implicitly on another, they are* **tightly** *coupled. Object instances are tightly coupled with their classes. When one object depends directly on the visibility of another, they are* **closely** *coupled. Smalltalk instances, class and pool variables are closely coupled to instances that reference them. When one object references another only indirectly through [...] the other's public interface [they are]* **loosely** *coupled.*

In object-oriented systems, close coupling should be avoided in favour of loose coupling and high cohesion, to achieve a well modularized system that allows reuse. Localizing areas of high coupling has therefore different reasons:

- Finding architecture parts that may need refactoring.
- Finding classes that require the context of their surrounding (closely coupled) classes for understanding and thus breaking the system down into smaller parts that are easier to comprehend.

### 2.2 Metric Calculation Conventions

For the calculation of metrics that require examining the bodies of operations (e.g. for method or attribute accesses), we examine class initialization blocks, constructors and methods. In general, we do not take classes from the Java core packages into account. These consist of implementation specific classes that usually have no direct impact on the abstract design of the software system. For example, Container classes from `java.util` package can be resolved (cf. section 2.3) and Exceptions (from `java.lang`) are a language specific mechanism that should not be reflected in the design. Similar considerations should also be done for other external frameworks, depending on their relevance for the system architecture.

We do not consider messages that a class sends to itself in the metric calculation. We do so for several reasons: Self messages do not appear in graphical representations on the class diagram level. It would be possible to show them in a similar way as in collaboration diagrams, using a self link, but the concept of the metrics employed by us is to get an idea of interconnection and interaction between distinct classes. Self messages do not cause class coupling and therefore do not contribute to the understanding of inter class communication. However, we do include inheritance coupling in our measurements, as it allows showing reuse by inheritance.

### 2.3 Examined Metrics

When determining the metrics that are suitable for isolation of diagram parts, we firstly distinguish between intra class metrics and inter class metrics. Intra class metrics measure characteristics of class interna and may therefore be used to select single classes whose respective metric values exceed a given threshold. Inter class metrics measure features between a set of classes and can therefore be used to select subsets of the given set of examined classes (e.g. a package). Below, we present several established object-oriented metrics that measure characteristics about classes or their relation to other classes.

The fan-in and fan-out metrics [12] measure the information flow between classes. Fan-in counts the number of locations from which control is passed into a module (a class, in our case), plus the number of data structures from which information is retrieved by the examined class. We use the following definition for fan-in. Let C be a class and S the set of classes calling methods from C. Then $FI = |S|$. The fan-in metric can be used to find classes whose services are used by many others, i.e. who have a high reusability. As the fan-out metric can be used synonymously with CBO (*coupling between objects* [11]), we only use the latter term.

CBO measures the number of classes to which a class is coupled. According to remarks and comments on CBO and coupling [5, footnote 5] [11, p.112], we include coupling through inheritance. Two classes are considered coupled, if methods declared in one class call methods or access attributes defined in the other class. We use the following definition for CBO:

Let C be a class and $M = \{m_1, ..., m_n\}$ the set of methods of C, $R_i$ the set of methods called by $m_i$ and $A_i$ the set of attributes accessed by $m_i$. Let $\omega() : Set(Feature) \rightarrow Set(Class)$ determine from a set of features (i.e. attributes or methods) the set of their owners (i.e. classes). Then

$$CBO(C) = |\bigcup_{v=1}^{n} \omega(R_v) \cup \bigcup_{w=1}^{n} \omega(A_w)|$$

High CBO values indicate large numbers of interconnections between classes. If a class with a high CBO value also has a high NOC (*number of children* [5]), this part of the design might deserve refactoring, as the high number of couplings propagates to all subclasses.

MPC (*message passing coupling*) counts the number of send statements in a class [17]. This is similar to CBO but considers in addition to the number of distinct coupled classes the number of messages, or the quantity of inter class communication. This can be used as a measure for the intensity of coupling between classes. When determining the distribution of sent messages to receiving classes, the resulting MPC numbers can be used to indicate the need for refactoring between subject classes.

### 2.4 On the Need for Container Resolution

Container classes can be considered auxiliary means on the implementation level to handle multiplicities. When viewing class diagrams on an abstract design level, we resolve containers and use the classes stored within instead [14]. With this approach, the measured values reflect better on the class design and leave out classes that are irrelevant for understanding the architecture (cf. section 2.2).

We use visitors to derive information the metrics base on from our data repository (cf. section 5). Our metric visitors recognize access to the classes held in containers like the Java collection classes. It is irritating if "contained classes" are shown in the diagram, but a relation is missing. To show the relations, it is necessary to resolve the containers and replace the associations to the container class with associations to the contained class. An example for container resolution is given in section 4.3.2.

## 3 Example Software Module

To validate the applicability of our approaches, we reverse engineered and analyzed several software systems, the largest being USE (*UML-based Specification Environment*) [19], a system for the specification and validation of UML models and OCL constraints. USE consists of about 450 classes with more than 2600 operations. To get an idea of the size and structure of a non-trivial system, we have included a roughly average-sized class diagram (consisting of about forty classes including binding to package-external classes) in figure 1, showing one of the USE packages. We discuss experiences with this project in section 6.

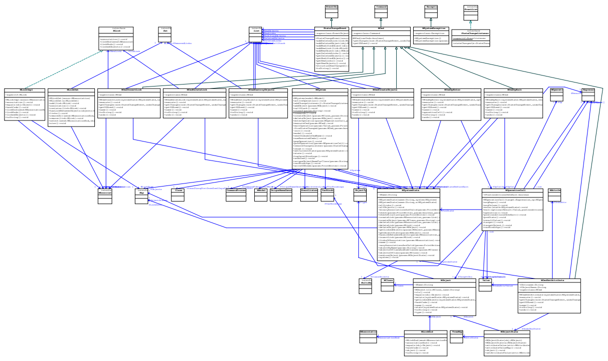

**Figure 1. Package org.tzi.use.uml.sys shown as a Class Diagram**

As can be seen, the resulting diagrams are by far too large to show them in the scope of this paper. Therefore, we present a sufficiently small system for the exemplary study in figure 2. Throughout section 4, we will expand and analyze it using the metric based approaches presented in this paper.

The module CFGraph is a Java package that provides a class structure for creating, maintaining and analyzing directed graphs. A graph in this sense is a set of nodes that are connected by edges. Every graph has a definite root and end node. An arbitrary number of edges may start at each node, but every edge has exactly one source and one destination node. No constraints are added concerning the identity of source and destination nodes, i.e. cycles are allowed.

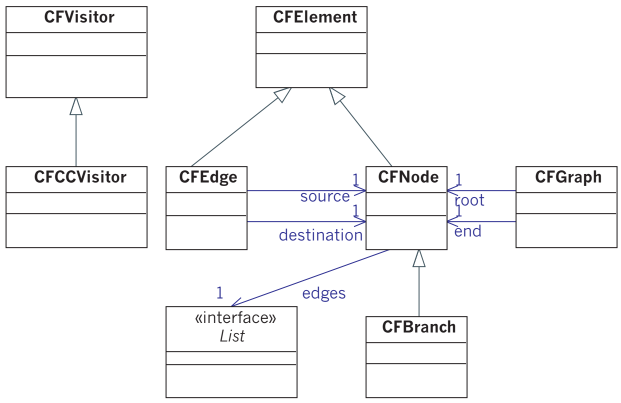


**Figure 2. Class Diagram for Graph Package**

The class diagram in figure 2 shows the static structure for the graph module. It contains two inheritance trees: one for graph elements, which are specialized to nodes, edges and branches. The second tree implements the "Visitor" design pattern [9] that can be used to traverse object trees and perform actions on them.

From the first impression, the central class appears to be CFNode: it is referenced twice by both CFEdge and CFGraph, with associations to the source and destination node of each edge as well as the absolute root and end nodes of the graph.

Note that the class diagram shows the structure on the implementation level. CFNode has an association with the role name edges that goes to the interface java.util.List. Association and role name are derived in the source code from the attribute of same denominator in class CFNode. The role name alone suggests a relation to a different class than targeted at by the arrow, namely to CFEdge. We will resolve this later on to show the correct relations and determine the couplings for CFNode.

## 4 Using Metrics to Isolate Submodules

Class diagrams are used to model and document the static infrastructure of software systems as a graph. When examining well designed systems, substructures can be recognized in such a diagram. The most common way to partition a system is by putting related parts together in separate modules. Rather than determining boundaries that partition a module (and thus produce disjunct submodules), we take a more inspective approach.

We produce sets of classes, whose contents depend on a *central class*. The latter is specified by the user by means of object-oriented metrics. We then create a class context which includes those classes that are related to the central class through a predefined metric. For CBO for example, this would include all classes related by coupling and for DIT (*depth of inheritance tree* [5]), it would include all classes from the inheritance tree. This way, the user incrementally examines the system in a bottom-up approach. Eventually, overlapping parts appear that belong to more than one class context and function as glue between the different parts.

Our approach includes three steps that are presented below: Finding the central class for the new context from the original diagram, determining the additional classes belonging to the context diagram and finally visualizing it.

In this context, we understand a submodule as a set of classes that are related by a certain characteristic, for example high coupling. Technically, a submodule consists of a central class and its associated class context. These parts are described below.

### 4.1 Finding a Central Class

After having reverse engineered a software package, a set of metrics is calculated for each of its classes. The kind of metric is generally open and depends on the user's needs. The only constraint is that it must be intra class metrics, i.e. measure some characteristic of a single class and allow to compare pairs of classes against each other by their values. Now, we can select a class from the choice proposed by the system. For our current implementation, it consists by default of those classes with above-average metric values. For large packages with an even distribution of metric values, the choice can be constrained further by specifying a threshold value.

Depending on the chosen metric, we could get, for example, classes with very complex operations or with a strong coupling.

### 4.2 Determining the Class Context

Now the class context is determined, beginning at the previously determined central class. First, it has to be decided which metric is used. The resulting context and the information conveyed by it depend on the chosen metric. Thus it is possible to visualize different characteristics of a class, e.g. coupling or inheritance. For example, when using the MPC metric, the class context is established by determining the set of classes to which messages are sent. Also, the number of messages sent to each class is examined. Rather than regarding only the numerical values yielded by metrics, we examine the underlying sets of classes to establish the class context. The context is derived from the set of classes counted by the chosen metric. This approach induces another constraint for the choice of metrics: the metric value has to be based on a set (usually, it is the cardinality of the set). Generally, all counted classes are considered. To further reduce the context size, thresholds can be specified to select only classes within a given range of metric values.

Depending on the chosen metric, different structures of class contexts are produced that are differently well-suited for visualization. The approaches presented here are based on MPC and CBO, as well as fan-in. Others are imaginable and are being worked on.

Our first approach uses the set of coupled classes as context. When determining the coupling for a class we distinguish between the number of coupled classes (CBO) and the tightness of coupling in terms of messages passed (MPC). To get an idea about the average tightness of coupling, we relate the number of classes involved to the number of passed messages. When searching for refactoring candidates, it is furthermore a good idea to examine the individual MPC-values for each target class to determine the classes that are most tightly coupled.

The class context depends on application of and threshold values for the metrics. A common approach for a first run is to use the set of coupled classes derived from CBO. This can be further restricted by limiting the context to those classes with a strong coupling, according to MPC values. The latter is preferable, if CBO produces large numbers of associated classes that have no strong binding to the subject class. By using MPC, we can ensure that the selected classes are strongly coupled.

To examine the package-global usage of a class and show classes with high reuse (i.e. such that are accessed by a large number of other classes), we use the fan-in metric. Classes that belong to the fan-in set do not necessarily have to be directly related. Their most interesting commonness is access of the subject class. By applying fan-in, the number of classes accessing it can be determined to get an idea of the external use, the tightness of coupling and the way other classes use it. This can be visualized in a class diagram (cf. section 4.3.2).

## 4.3 Graphical Representation as UML Class Diagram

Basically, we use UML class diagrams for representation of the metric-based submodules. However, many of the measured class relations are not shown in common class diagrams, e.g. local variables and calls to static methods. For graphical representation of relations derived from metrics and for enabling the user to understand the context, dependencies are added to the diagram. These are distinguished according to their application: First, we describe the use of a separate diagram to show coupling between classes on a package-global level. For visualizing submodules in fine detail, a class diagram is extended by the dependencies shown in section 4.3.2.

### 4.3.1 Visualizing Class Coupling

To get an understanding of the coupling between classes within a package, we employ what we call a coupling diagram. This is actually a class diagram that shows class coupling by means of dependencies. In addition to the CBO metric that can be used to identify the set of class couplings for a given class, we also use MPC to determine the intensity of coupling between class pairs.

The coupling diagram bases on the classes from the original class diagram for the examined package, with dependencies added between coupled classes. These can be extended with an optional stereotype to show the kind of coupling, e.g. message passing. Also, each dependency is labeled with the metric value for the connected classes, thus yielding a weighted graph showing the coupling distribution over the diagram. The information about metric types is especially interesting, if different metrics are used to determine coupling for a single diagram. However, stereotypes should be used sparsely, as the diagrams can quickly become hard to read otherwise. In practical use with larger diagrams, it has proven sensible to use only one metric at a time and show only the metric values for labeling.

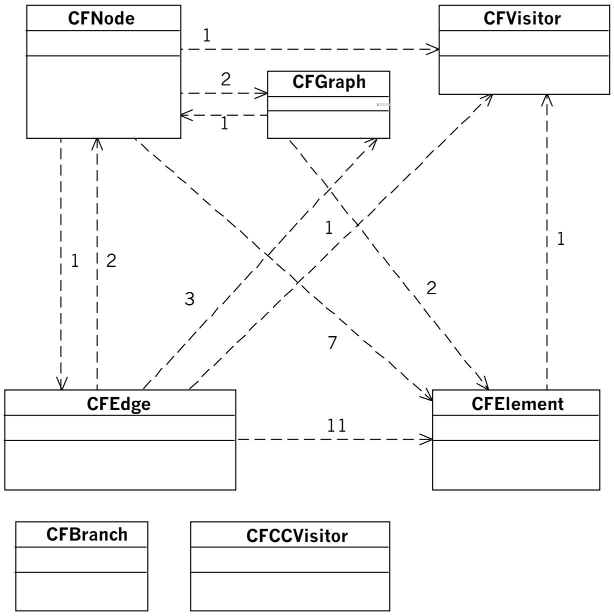


**Figure 3. Coupling Diagram for CF Package**

By default, the coupling diagram shows all couplings between classes. To show regions of strong coupling, a threshold value is applied to the metric and only dependencies with values lying above it are shown. As a default, the average metric value for the package is used. This can then be increased gradually to further isolate selections.

The diagram in figure 3 shows coupling calculated with the MPC metric. Notable are the high coupling values between CFEdge and CFElement, as well as CFNode and CFElement. As can be seen in the class diagram in figure 2, both are subclasses of CFElement. The high numbers therefore relate to strong inheritance coupling, which indicates a high level of reuse. This observation could be verified by checking the source code. On the other hand, the classes CFCCVisitor and CFBranch show no coupling at all. This is due to the fact that both are children in an inheritance hierarchy and are not referenced directly by other classes. CFBranch is identical to CFNode modulo type and only used to distinguish branches from ordinary nodes by means of polymorphism. Classes from CFElement's inheritance tree can be passed a CFVisitor to perform actions on them. In its place, every subclass of CFVisitor can be passed, but these do not have to be known by the accepting classes and are therefore not coupled, as the case for CFCCVisitor. The coupling between the classes from CFElement's inheritance tree and the visitor class can be verified by checking the dependencies targeting the visitor: all have an MPC value of 1, which – as we will see later – refers to calls of the visit() method of the visitor, which can be accessed via the passed parameter of the accept() method from the element classes.

An interesting question that arises when examining the coupling diagram is whether to consider messages sent to container classes. We can see that CFNode sends only a single message to CFEdge. As it also receives only two messages in return, this numbers seem to be quite consistent. Still, by checking the source code, it can be seen that several messages are not regarded. As these address the handling of object groups and are not related directly to the contained class (CFEdge in this case), we do not reflect them in the coupling diagram.

The difference between the common use of metrics and the coupling diagram is the view on the metrics. Instead of calculating the value for a given class and printing it, we split the metrics up and examine the partial values. A separate local MPC value is calculated for each class with coupling, which yields a large number of values. Showing them as labels on dependencies creates a weighted graph that gives a package-global overview on the coupling between classes on a quantitative level. To get a qualitative, more detailed impression of the software architecture, the following approach can be used.

#### 4.3.2 Fine Grained Information for Subdiagrams

The subdiagrams described here document the infrastructure of software systems on a higher detail level than usual for class diagrams. This is used to show all connections between classes that are necessary to comprehend the coupling between classes, i.e. the basis for the metrics applied to them. In addition to the common relations association, generalization and realization, we must also consider local variables, parameters, etc. as they are not reflected in the basic class diagram. To represent these additional elements, we use dependencies that are extended by the following UML stereotypes:

- $\ll call \gg$ is used to mark invocations of static methods from another class.
- $\ll parameter \gg$ marks a relation which is established by passing an object reference as a method parameter to the subject class.
- $\ll local \gg$ denotes a relation to another class which is caused by creation of a local variable in the subject class.

The keywords $parameter$ and $local$ are defined in the metamodel as stereotypes for a LinkEnd and refer to the instance level. The meaning in combination with a dependency arrow in a class diagram is that the source class of the dependency is related to the target class because of a relation of the kind and level given by the respective stereotype. The keyword $call$ defines a usage dependency whose target is a static method call (a class operation, in UML terminology). It means that an operation of the source class invokes a class operation of the target class. Ordinary method calls and constructor calls do not have to be marked by a dependency, as to call a method of another class requires the subject class to have an attribute of the other class. A constructor call marks the establishment of the respective association (note that these are only unidirectional associations).

There are many ways to capture the coupling between classes by means of metrics. For detailed class diagrams as the one shown in figure 4, we present all available information about class relations, independently of the applied metric. Not all of these are used for each metric, though. For example, CBO considers method calls and attribute accesses only, while MPC measures nothing else by method calls.

UML dependencies are very useful to show finer grained information about a system's architecture. However, as they apply not only to the static class structure but also to the static expression traces from methods, showing all available dependencies would create a substantially larger amount of information than is possible to render in a concise manner or necessary to understand the diagram. Therefore, we only use dependencies to show the relation of subject and target classes that are not related in terms of static structure. They are not added, if subject and target class are already related by inheritance, realization or association, including resolved container types. Additionally, we introduce a precedence order in which dependencies are added: $call$, $parameter$, $local$. While hiding some information, this step ensures that the diagrams are not overloaded with details. The user can show the hidden parts, too, if desired. Within the order, $call$ takes a special role as it refers to invocation of class methods and therefore cannot interfere with the other dependencies, which apply to instances only.

Apart from attributes, parameters and local variables provide a way to reference other classes. $Parameter$ is preferred over $local$, as it describes a part of the static structure (more exactly of a method's signature) and thus can be understood more easily when examining the class diagram.

The diagram in figure 4 shows the class context for class CFEdge, consisting of the classes CFNode, CFElement, CFGraph and CFVisitor. We can see that only two of the classes are associated to CFEdge by classical relations: CFNode is connected by association and CFElement by inheritance. In contrast to the original class diagram in figure 2, the following additional relations are now visible: A dependency to CFVisitor is caused by parameter vis of method CFEdge.accept(). When comparing the class context with the coupling diagram in figure 3, we can observe that the same classes that pass a message to CFVisitor are linked to it by a parameter to their accept(...) method. We can conclude that the parameter is used to call the visitor's visit(...) method. The second dependency of CFEdge points at CFGraph and is caused by an invocation of the class method println(...). Finally, it can now be seen that the resolved association edges from CFNode actually points to CFEdge, which is the class contained in the former container class. We have described the process of container class resolution and recognition of aggregations is in our previous work [14, 10].

The additional information which is necessary to understand the class context adds of course to the overall complexity of the shown diagram. To avoid increased complexity and bad readability in the whole diagram, we only show this additional information in the subdiagrams. This can be considered increasing the level of detail when taking a closer look at a part of the diagram, comparable with zooming into a map.

# 5 Implementation and Tool Support

Our prototypical implementation, Idea, supports static reverse engineering and analysis of Java programs. The program information is extracted from source and byte code, using parsers that capture static structure and method traces in an object-oriented data repository. We employ an implementation of our Java metamodel here that is extended by tool functionality and implementations of different metric suites.

As the source code is held in tree structures, it is easily possible to examine and search it for software artifacts by using visitors. The application of search algorithms and transformations makes it possible to convert the implementation level data model to a more abstract design model [14, 10], which represents programming language-specific constructs on a more general level, using UML features.

For the third step in our approach, the visualization, it is necessary to render UML diagrams graphically in an integrated software system. As we are building our diagrams from a data model, we also need support for automatic diagram layout. These are obvious, yet common issues for many research projects that deal with the UML. Instead of writing support for such features anew, it would be desirable to have a generic tool that is able to read UML data models in a standardized form (e.g. XMI), layout created diagrams and render them to the screen.

For our visualization of class diagrams, we have interconnected Idea with the CASE tool Together [26], which provides access through an included programming interface. This allows us to layout and visualize structure models of software systems generated by Idea as class and package diagrams.

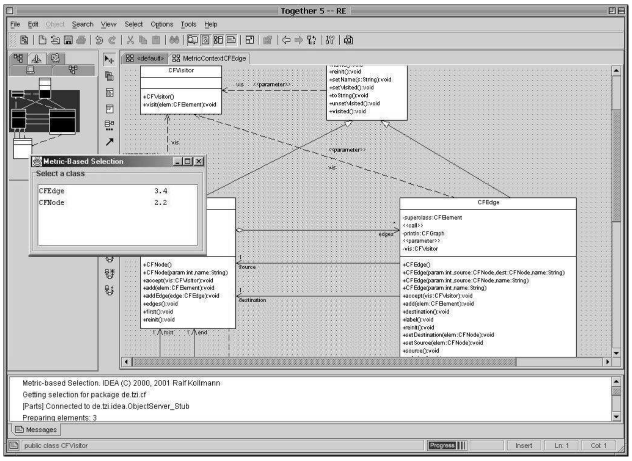


**Figure 5. The Idea tool and its graphical output in Together**

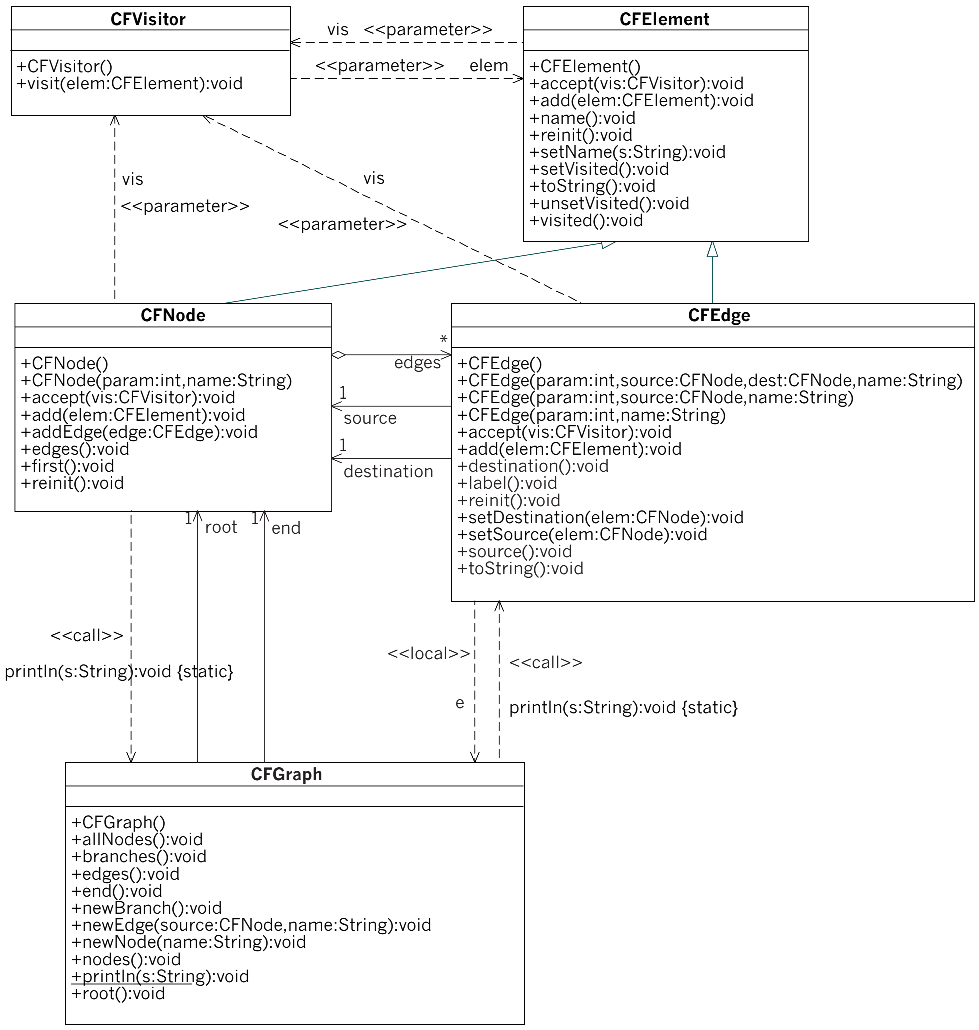


**Figure 4. Class Context Diagram for Class CFEdge**

## 6 Experiences with a Large System

We tested our implementation with several software systems of different sizes between eight and 450 classes. Generally, we tried to cover the reverse engineered class diagram with as little submodules as possible, to keep redundancy low. For the larger systems, we also had to consider external classes that caused inter-package dependencies.

The average submodule size remained in manageable magnitudes: five classes for submodules of the smallest package (CFGraph) and up to 15 for some of the USE packages. It is notable however that the majority of submodules remained within sizes below ten classes. The degree of overlapping between submodules depended strongly on the interconnectedness of the whole diagram. Therefore, packages with a clean design that used loose coupling were easier to divide into submodules. The class diagram for package org.tzi.use.uml.sys (shown in figure 1) required five submodules to cover all classes. The sum of the classes shown in all submodules was 56, compared to 43 classes in the original diagram, resulting in an average redundancy of about two and a half classes per submodule.

The bottom line of our experiences was that is possible to divide large diagrams into mainly distinct submodules. The remaining amount of redundancy was generally considered useful: This is a similar issue as encountered when examining programs with several packages. Most commonly, large programs contain classes that have relations to package-external classes, so there is a comparable situation with some classes being shown in several diagrams. These overlapping classes often functioned as a glue between modules and helped holding the parts together.

The size of the submodules was small enough to visualize the largest part of the diagram on screen while still being able to recognize details. If navigating through the diagram was necessary, it could be limited to a reasonable amount.

In practical use, we have experienced that interactivity is one of our tool's key factors and leads to an exploring manner of examining software. After reconstructing the abstract design of a software system, we consider the metric based approach the next more advanced step. Its successful application requires understanding of the metric suite that is used. Then, it is relatively easy to get further insights into the system architecture by using selected metrics as a means for emphasizing the desired aspects. Both diagram extensions presented here have to be used in combination to take the best advantage. The additional structural information provided by class context diagrams helps to understand the relations indicated by the metric values. We have found that especially during the elaboration of a software design or the reengineering of an existing system, it is possible to recognize inelegant implementations or redundancies, which can be repaired for example using refactorings.

## 7 Related Work

In the research project presented in [25], reverse engineered Java software systems are represented as graphs and analyzed using object-oriented metrics. The Chidamber&Kemerer suite of metrics[5] is presented and integrated into a reverse engineering tool. The graphs are extended by annotation with metric values, which is quite similar to the coupling diagrams presented in this paper.

Another approach proposes visual representation of metrics as visual attributes of nodes in a graph [6]. Up to five different metrics are thus "encoded" into a single node and provide visual information about characteristics of the analyzed program. The used metrics are divided into class, method and attribute metrics.

While most research projects in the field of metrics and visualization make use of 2-dimensional graphs, the work presented in [24] takes a new approach by presenting techniques to explore and analyze visualizations of software systems in 3-dimensional space. Metrics are applied here to recognize "bad smells" known from refactoring [8] and geometrical distance is used as medium to convey cohesion of classes, methods and attributes.

Currently, research on lifting UML diagrams to 3-dimensional space is in progress in our work group, so we can imagine that future research may involve reasoning on how to adopt and extend the aforementioned techniques for use with the UML.

The work in [13] applies metrics to direct software reviewers to critical parts of software systems. The static structure of programs is analyzed and its properties are measured. To achieve flexibility and interactivity while examining the software, the information about the static structure is stored in a data base, with measurements being defined as SQL-like queries on it.

## 8 Summary and Conclusion

We presented approaches to use object-oriented metrics for analyzing programs and isolating coherent submodules from existing class diagrams representing non-trivial software systems. The shown techniques make it possible to understand programs in a bottom-up manner, by examining smaller subparts that eventually merge to a whole. Our motivation was to find solutions for the "haziness" often encountered when viewing large UML diagrams: because of their inherent characteristic to contain graphical as well as textual information, it is hard to see both fine detail and an overview over the complete structure at the same time. We have presented an exemplary application of our approach showing two different visualizations in addition to the plain class diagram: coupling diagrams show coupling between classes as dependencies labeled with metric values and optional stereotypes specifying the used metric. A high level detail class diagram is used to show submodules isolated by means of metrics. It extends the conventional class diagram notation by dependencies that document all class relations that have been derived from the metrics and that are necessary to understand the connectivity of the submodule. The approaches presented here have been implemented in our reverse engineering tool, Idea. Current and near-future research focuses on a wider-spread evaluation of metrics concerning their applicability in combination with the isolation algorithms.

## Acknowledgments

We thank the anonymous referees for their comments, which helped to improve this paper.

## References

[1] J. M. Bieman and B.-K. Kang. Cohesion and Reuse in an Object-Oriented System. In *ACM SIGSOFT Symposium on Software Reusability*, pages 259–262, 1995.

[2] N. Boyd. Building Object-Oriented Frameworks. *The Smalltalk Report*, 3(1):1–6, 14–16, September 1993.

[3] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering: An International Journal*, 3(1):65–117, 1998.

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

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

[6] S. Demeyer, S. Ducasse, and M. Lanza. A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualization. In *Proc. 6th Working Conference on Reverse Engineering (WCRE)*, pages 175–186, 1999.

[7] J. Eder, G. Kappel, and M. Schrefl. Coupling and Cohesion in Object-Oriented Systems. In *Proc. Conference on Information and Knowledge Management, Baltimore, USA*, 1992.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[10] M. Gogolla and R. Kollmann. Re-Documentation of Java with UML Class Diagrams. In E. Chikofsky, editor, *Proc. 7th Reengineering Forum, Reengineering Week 2000 Zürich*, pages REF 41–REF 48. Reengineering Forum, Burlington, Massachusetts, 2000.

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

[12] S. Henry and D. Kafura. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, September 1981.

[13] G. Köhler, H. Rust, and F. Simon. An Assessment of Large Object Oriented Software Systems. In W. Melo, S. Morasca, and H. A. Sahraoui, editors, *Proceedings of Object-Oriented Product Metrics for Software Quality Assessment at 12th European Conference on Object-Oriented Programming, CRIM Montreal*, 1998.

[14] R. Kollmann and M. Gogolla. Application of UML Associations and Their Adornments in Design Recovery. In P. Aiken and E. Burd, editors, *Proc. 8th Working Conference on Reverse Engineering (WCRE)*. IEEE, Los Alamitos, 2001.

[15] R. Kollmann and M. Gogolla. Selektive Darstellung von Programmstrukturen mit UML. In J. Ebert and F. Lehner, editors, *Proc. 3rd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany*. Universität Koblenz, Technischer Bericht, 2001.

[16] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Fakultät Informatik der Universtät Stuttgart, Deutschland, 1999.

[17] W. Li and S. Henry. Object-Oriented Metrics that Predict Maintainability. *The Journal of Systems and Software*, 23(2):111–122, November 1993.

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

[19] Mark Richters. UML-based Specification Environment (USE), 2001. `http://www.db.informatik.uni-bremen.de/projects/USE/`.

[20] U. A. Nickel, J. Niere, J. P. Wadsack, and A. Zündorf. Roundtrip engineering with FUJABA. In *Proc. of 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany*, Technical Report 8/2000, Fachberichte Informatik. Universität Koblenz-Landau, 2000.

[21] OMG, editor. *OMG Unified Modeling Language Specification, Version 1.3, June 1999*. Object Management Group, Inc., Framingham, Mass., Internet: `http://www.omg.org`, 1999.

[22] OMG. UML Notation Guide. In *OMG Unified Modeling Language Specification, Version 1.3, June 1999* [21], chapter 3.

[23] OMG. UML Semantics. In *OMG Unified Modeling Language Specification, Version 1.3, June 1999* [21], chapter 2.

[24] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics Based Refactoring. In P. Sousa and J. Ebert, editors, *Proc. 5th European Conference on Software Maintenance and Reengineering*, pages 30–38. IEEE, Los Alamitos, 2001.

[25] T. Systä, P. Yu, and H. A. Müller. Analyzing Java Software by Combining Metrics and Program Visualization. In *Proc. 4th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 199–208, 2000.

[26] TogetherSoft Corporation. Together 5, 2001. `http://www.togethersoft.com`.