MetricAttitude: A Visualization Tool for the Reverse Engineering of Object Oriented Software

Michele Risi
Dipartimento di Matematica e Informatica
Università di Salerno
Fisciano, Italy
mrisi@unisa.it

Giuseppe Scanniello
Dipartimento di Matematica e Informatica
Università della Basilicata
Potenza, Italy
giuseppe.scanniello@unibas.it

ABSTRACT
In this paper, we present a visualization approach for the reverse engineering of object-oriented (OO) software systems and its implementation in MetricAttitude, an Eclipse Rich Client Platform application. The goal of our proposal is to ease both the comprehension of a subject system and the identification of fault-prone classes. The approach graphically represents a suite of object-oriented design metrics (e.g., Weighted Methods per Class) and “traditional” code-size metrics (e.g., Lines Of Code). To assess the validity of MetricAttitude and its underlying approach, we have conducted a case study on the framework Eclipse 3.5. The study has provided indications about the tool scalability, interactivity, and completeness. The results also suggest that our proposal can be successfully used in the identification of fault-prone classes.

Categories and Subject Descriptors
H.5.2 [Information Interfaces and Presentation]: Graphical User Interface; I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction Techniques

Keywords
Program Comprehension, Reverse Engineering, Software Maintenance, Metrics, Software Visualization

1. INTRODUCTION
Many activities in the software engineering field involve existing software. Software maintenance, testing, quality assurance, reuse, and integration are only few examples of software processes that involve existing systems [1]. A crucial issue of these processes entails the reverse engineering of these systems.

The term reverse engineering has been defined in the context of software engineering as: the process of analyzing a subject system to (i) identify the system’s components and their inter-relationships and (ii) create representations of the system in another form or at a higher level of abstraction [2].

To summarize, this process entails the gathering of information in software artifacts (not only source code). On the basis of that information, a reverse engineering process builds more understandable representations of software artifacts.

In the reverse engineering field, software visualization [3], [4] is extensively and successfully explored and used. Researchers have proposed approaches/techniques and supporting tools based on 2D and 3D environments [5], [6], [7], [8]. These approaches visualize either static information, performing static analysis (e.g., [9]), or dynamic information, relying on the execution of the system (e.g., [10]). Dynamic information is affected by the execution scenarios that should be as much as possible representative of the system’s typical usage [11]. This concern is the weakness of visualization approaches based on dynamic analysis.

Whatever is the information to be visualized, a relevant concern of visualization is the way information is presented. The choice of a proper visualization is straightforward, when a well known and highly adopted representation is used, e.g., the notations of the UML [12]. In other cases, a visualization is the essence of a technique because it has to highlight relevant information at a proper level of detail. The kind of representation adopted impacts on the usefulness of the technique [1].

In this paper, we present an approach for software visualization based on static analysis. The approach provides a mental picture by viewing an object oriented (OO) software system by means of polymeric views [9], i.e., lightweight software visualizations enriched with software metrics. The approach has been implemented in a prototype of a supporting tool, intended as an Eclipse Rich Client Platform (RCP). We named this prototype MetricAttitude. To assess the validity of both the approach and the prototype, we have conducted a case study on the framework Eclipse 3.5. The case study provided indications about the tool scalability, interactivity, and completeness. Further, to assess whether our proposal eases the identification of fault-prone classes, we have studied the visual representation of Eclipse 3.5 with respect to data associated with changes in response to bug reports in a public repository.

The main contributions of the paper are:

- A visualization approach for the reverse engineering of OO software. It is based on OO metrics and traditional code-size metrics;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AVI ’12 May 21-25, 2012, Capri Island, Italy
Copyright 2012 ACM 978-1-4503-1287-5/12/05 ...$10.00.

1 Please read the paper on-screen or as a color-printed paper version, we make extensive use of color pictures.
• A prototype of a supporting system. To increase the comprehension of a subject system, it implements several views, filtering, and display/editing operations;

• A novel static analysis based approach to recover relationships among classes. Relationships are used in the visualization together with the metrics collected on the system;

• A case study on a large open source software system implemented in Java. The results of this study revealed that the prototype has the following main characteristics: scalability, interactivity, and completeness;

• The bug reports of Eclipse 3.5 has been used to assess whether MetricAttitude and the underlying approach can be used to identify fault-prone classes.

The remainder of the paper is organized as follows: in Section 2, we review previous software visualization approaches and techniques. In Section 3, we describe our approach, while we present MetricAttitude in Section 4. In Section 5, we outline the preliminary results of the conducted empirical evaluation. Finally, Section 6 concludes and discusses possible future directions for our research.

2. RELATED WORK

There are a number of approaches/techniques for software visualization based on Synthetic Natural Environment (SNE) (e.g., [13], [14], [15], [16]). In the first subsection, we discuss these techniques. In the literature, visualization techniques based on graph representations and on polymetric views have been also proposed [9], [18], [19]. We present some of them in the second subsection. Due to the huge number of software visualization techniques, a deep and exhaustive discussion of related work is not possible and it is also out of the scope of the paper. An extensive taxonomy of software visualization, with several examples and tools is available in [20].

2.1 SNE based Techniques

Among the SNE based techniques, the city metaphor is one of the most used (e.g., [13], [14], [15], [16]). Wettel and Lanza [21] propose such a kind of metaphor for the comprehension of object oriented software systems. Classes are represented as buildings and packages as districts. The authors implement the metaphor in the CodeCity tool. In a more recent paper, the same authors [22] present a controlled experiment to assess the validity of the metaphor and CodeCity. The experiment was conducted with software professionals. The results indicate that the tool leads to a statistically significant improvement in terms of task correctness. The results also indicate the task completion time statistically decrease when using the tool.

Graham et al. [23] propose a solar system metaphor. Suns represent a package, planets are classes, and orbits represent the inheritance level within the package. Martínez et al. [24] suggest a metaphor based on landscape. The main objective of that proposal is to visually represent a number of aspects related to software development processes. Ghandar et al. [25] propose a jigsaw puzzle metaphor. Each software component is a piece of a jigsaw puzzle. The system complexity is represented as a pattern on the surface of the pieces.

Erra and Scanniello [8] present an approach based on a forest metaphor to ease the comprehension of OO software. A software is depicted as a forest of trees. Each tree is a class. The trunk, the branches, the leaves, and their color represent characteristic of the class (e.g., a method is a branch of the tree). CodeTrees is the name of the tool implementing the metaphor.

Several are the differences between our proposal and the approaches discussed above. The most remarkable one is that our approach enriches lightweight visualizations with metrics information and relationships among classes.

2.2 Polymetric and Graph Views

Lanza and Ducasse [9] present the polymetric views. These views are lightweight visualizations enriched with software metrics. This approach presents a number of similarities with our approach, e.g., the structure of hierarchies is enriched with software metrics information. Differently, our approach is able to (i) visual represent more metrics at the same time and (ii) show instance level relationships and structure of hierarchies in the same view. In this paper, we also propose a novel static analysis based technique to recover relationships among classes. The validity of our tool and the underlying approach is assessed on a large open source software system.

Reniers et al. [26] present SolidSX, an integrated tool for visualizing large software compound attributed graphs. Our approach is different because it is able to visually represent more metrics in the same view. The software engineer can apply filters to hide useless information for the problem at the hand.

3. THE APPROACH

It is essential to decide which goals have to pursue, when defining a new visualization approach for the reverse engineering of a subject system [9]. We have focused on the following goals of primary importance for our proposal:

• Getting an overview of the system in terms of size, complexity, and structure.

• Locating the most important classes and inheritance hierarchies. This allows comprehending the classes and hierarchies that represent a core part of the system.

• Identifying classes that exchange a huge number of messages and make strong usage of delegations. These classes could require refactoring operations to improve the quality of the subject system.

• Identifying exceptional classes in terms of size and/or complexity compared to all classes. These classes may be candidate for a further inspection or for the application of refactoring.

• Suggesting the presence of design patterns or occasions where design patterns could be introduced.

The approach provides a large-scale understanding of a software system visualizing all classes together and handles class instance level relationships and structure of hierarchies. We also provide a fine-grained representation of individual classes (e.g., the name of the class and its methods).
In the following, we describe the selected metrics and the static analysis based approach we defined to get relationships among classes. The section concludes presenting the mapping proposed between the graphical properties of classes (or more generally software entities) and the selected metrics as well as the representations used for class instance level relationships and structure of hierarchies.

3.1 Metrics

A reverse engineering process should not only present a list of problematic classes or subsystems [9], but its result should also facilitate the identification of valuable piece of information for the quality attributes (i.e., complexity, efficiency, re-usability, testability, and maintainability) of the subject system [27]. To this end, software metrics have been proposed [28], [29], [30], [31].

The concept of software metrics is well established in the software engineering community [32]. Recently, an increasing interest in OO metrics [33] has been showed thanks to the increasing popularity of OO analysis and design methodologies. We considered the most popular OO metrics firstly presented by Chidamber and Kemerer [34] and “more traditional” code-size metrics:

- **Weighted Methods per Class (WMC).** It is defined as being the number of all member functions and operators defined in each class. WMC measures the complexity of individual classes.
- **Depth of Inheritance Tree (DIT).** It measures the number of ancestors of a class. DIT primarily evaluates efficiency and re-usability.
- **Number Of Children (NOC).** It gives the number of direct descendants for a class. NOC is used to assess efficiency, re-usability, and testability.
- **Coupling Between Object classes (CBO).** It measures the number of classes to which a given class is coupled. CBO evaluates efficiency and re-usability.
- **Response For a Class (RFC).** It measures the number of methods that can be potentially executed in response to a message received by an object of that class. This metric evaluates understandability, maintainability, and testability.
- **Flow Info (FI).** It is computed on fan-in and fan-out as follows: \( L = (fan-in + fan-out)^2 \). \( L \) is a code-size metric. We used the LOC metric. Fan-in measures the number of methods called in a software component. We considered here a class as a component. Conversely, Fan-out is defined as the number of local flows out of a component. As for Fan-in, we considered a class as a component. FI is employed to measure testability.
- **Number of Message Sends (NMS).** It measures the number of messages from a method. This metric is designed to be a unbiased measure of the size of a method. The value for a class is achieved by summing the NMS values of all the methods in the class. This metric is used to evaluate re-usability and testability.
- **Number of Methods (NM).** It represents the number of method of a class and is used to determine the complexity.

**Lines Of Code (LOC).** It indicates the number of all non-empty and non-comment lines of the class and its methods. LOC is useful to estimate maintainability.

**Number of Comments (NC).** It represents the number of comment lines of the body of a class and its methods. NC can be used to evaluate understandability, re-usability, and maintainability.

As shown in [27] and recalled above, these design metrics provide useful information on software quality attributes (e.g., complexity and maintainability).

3.2 Reverse Engineering the Subject System

We highlight here the new static analysis approach proposed to identify relationships among abstract classes, interfaces, concrete classes, and inner classes. Our proposal has been inspired from the work presented by Sundaresan et al. in [35]. Several are the differences with respect to that approach. The most remarkable one is that we work on Java source code, while byte-code is used in [35]. Our choice makes our approach more flexible and easily modifiable for subject systems implemented with programming languages different from Java.

We first build a call graph, where each node represents an abstract class, an interface, a concrete class, or an inner class. For simplicity reasons, we will refer to them as class or software entity. We will distinguish among abstract class, interface, concrete class, and inner class only if needed. Directed edges in the call graph are invocations from a method in a class to another method in the same class or in a different class.

Special attention is needed when adding an edge representing a call to a virtual method (i.e., a method without a concrete implementation). We did that by using an approximation of run-time types of receivers. In particular, we find all the possible targets of a method call of a class by looking down the class hierarchy of the receiver. Directed edges are added from the caller to each possible target receiver. We called this kind of edges as Virtual Delegation. At run-time only a call to a target method is present. This is the difference between the type resolution at run-time and an approximation of run-time types. We observed on the system used in the case study (i.e., Eclipse 3.5) that only in a few cases the number of possible target receivers is greater than one. For example, the class FileHandle through the method handleNotification calls the virtual method exists of the interface ChainedHandle. The found target methods are those implemented in the FileHandle and LinkedResourceHandle classes, that implement the ChainedHandle interface. In the case study, similar conditions happened in 132 cases out of 682.

There could be also the case that no target receivers are found using an approximation of run-time types. Then, we add a directed edge from the caller to the class containing the virtual method. This kind of directed edge is named Abstract Delegation. An example of this edge and how it can be used in the identification of a design pattern is shown in Figure 1.

Directed edges in the call graph are also added on the basis of the access to public fields. Accesses to public fields are managed similarly to the invocations from a method in a class to another method either in the same class or in another class. For space limits, details are not provided.
3.3 Graphical Representation of Software Entities and their Relationships

The chosen mapping and graphical representations are inspired by well-known software visualization techniques (e.g., [9],[18]) and the experience the authors gained in the program comprehension field and software visualization and development. We also tried as much as possible to find intuitive mappings to ease the comprehension of both high and low-experienced software engineers.

Each software entity is represented with a node, which is graphically represented with an ellipse and a rectangle placed just below. As shown in Figure 2 and Table 1, the properties of ellipse summarize OO metrics, while the properties of the rectangle traditional code-size metrics. Table 1 also shows how the kind of relationships among classes are graphically represented.

The dimension of ellipse denotes WMC. The convention is that the larger the ellipse, the higher the metric value is. We used the flattening of ellipse to represent DIT. The more the flattening, the higher the metric value is. The color of the border of ellipse is used to denote NOC. The color is between light red and dark red. The higher the metric value, the darker the border is. We used the thickness of the border of ellipse to represent CBO. The larger the thickness, the higher the metric value is. To avoid losing information about NOC, a thin border is drawn even if CBO is 0. The color of ellipse denotes RFC. The color ranges between black to white. Dark gray represents a smaller metric measurement than light gray.

The color of rectangle denotes FI. Similar to RFC, the color ranges between black and white. The lower the metric value, the darker the color is. We used the thickness of the border of rectangle to represent NMS. The larger the thickness, the higher the metric value is. We used the color of the border of rectangle for NM. The color ranges between light and dark red. Light red represents a smaller metric measurement than dark red. The height of rectangle denotes LOC. The convention is that higher the metric value, the more the height of the rectangle is. The width of rectangle is used for NC. The higher the metric value, the more the width is.

Relationships between classes are showed as lines: simple, arrowed, and hollow. These lines can assume different colors (see Table 1). With regard to the instance level relationships, we also considered the thickness of the lines associated. More the thickness, the larger the number of links between a source class to a target one is.

2A sample video of the execution of the approach and the prototype is available at http://www.dmi.unisa.it/people/risi/www/MetricAttitude/. The prototype is also available for downloading.
MetricAttitude also implements various graphical layouts for disposing the graphical elements of the subject system. The implemented layout algorithms are those shown in Figure 4. For each entity, the tool also shows: the name, the fields, and the methods. The values of all the metrics are shown as a tooltip when the mouse pointer is over the associated graphical object.

The technologies used to implement MetricAttitude are:

- **JDT (Java Development Tools) Core**. It is included in the Eclipse SDK and offers a built-in incremental Java compiler to run and debug code which still contains unresolved errors. JDT Core also provides a model for Java source code. The model offers API for navigating elements like package fragments, compilation units, binary classes, types, methods, fields, and so on. Although JDT Core implements a number of features, our system exploits the incremental Java compiler to make independent the subject system from the libraries it uses. Our prototype also uses the Java source code model to identify class instance level relationships (as shown in Section 3.2) and structure of hierarchies.

- **JGraphT**. It is a Java library for graph visualization. It supports drag and drop operations, selection modes, and display/editing options. The graphical appearance of the nodes can be easily modified and used. To automatically position nodes, JGraphT implements a number of layout algorithms. Examples of available layouts are: hierarchical, organic, and tree. JGraphT easily allows defining new layouts. This was one of the reasons because we chose that library. The prototype uses that technology to visualize the subject system.

- **Eclipse RCP**. This technology allows developers to use the Eclipse platform to create flexible and extensible desktop applications. The Eclipse RCP applications are build upon a plug-in architecture. We used this technology because it easily allows integrating Java-based technologies. The use of the Eclipse RCP technology made the prototype platform independent.

5. **EVALUATION**

To validate MetricAttitude, we conducted a case study on the core of the framework Eclipse 3.5. It is an open source multi-language software development environment. The Eclipse framework is implemented in Java and has an extensible plug-in architecture. It has been also used in the past to assess the validity of various software visualization tools (e.g., [21]). The core of the framework has: 331 classes, 2313 methods, and 1580 attributes. The number of lines of code is 73809, while the number of line comments is 31427.

Figure 5 shows Eclipse 3.5 as it appears in MetricAttitude. There is a software entity that represents the core part of the subject system. This entity is that on the top of the class hierarchy. The height and width of the rectangle associated to this entity indicates high values for LOC and NC. Further, the dimension of this rectangle is greater than the other rectangles. As suggested by the thickness of the rectangle border the represented software entity is an abstract class or an interface. We inspected the source code to confirm that assumption. It is an interface, whose name is IResources.

Figure 1 shows an excerpt of the class diagram of Eclipse 3.5. It represents a command design pattern of Eclipse 3.5. There is an Abstract Delegation from the class BuildManager to the ICommand interface. MetricAttitude adds a Virtual Delegations to the concrete class BuildClass. This result gives credit to the fact that our approach could suggest the presence of design patterns or occasions where design patterns could be introduced.

The use of MetricAttitude on Eclipse 3.5 leads to the following considerations:

- **Scalability**. The tool scales up well in terms of the size of the software systems to be visualized. On a very large software systems (i.e., Eclipse) the tool did not slow down. The good scalability is due to the use of both JDT Core for the reverse engineering of Eclipse and JGraphT for its visualization.
• **Navigation and Interactivity.** The tool enables the visualization of the name of each class through a label associated to the corresponding node. For each node, the tool also shows attributes and methods. Furthermore, double-clicking on the node the source code can be inspected or modified. The modified source code is automatically compiled and the graphical representation of the subject system is modified according to the modifications performed. This is possible due to the use of JDT Core. Several layouts and filtering operations are also available to improve interactivity. The tool provides moving and selection modes as well as display/editing options.

• **Completeness.** A number of features for analyzing single nodes are available. For example, the tool visualizes the values of all metrics, the kind of node (abstract class, interface, concrete class, and inner class), the fields, and the signatures. The approach and MetricAttitude also provide a fair amount of information for an overview of the subject system. In fact, nodes can be graphically compared each others to get indications about the relevance of abstract class, interface, concrete class, and inner class.

### 5.1 Identification of Fault-Prone Classes

The goal of this part of the evaluation is to analyze whether MetricAttitude provides support to the identification of fault-prone classes. The perspective of the study is both from the point of view of the researcher, evaluating whether the visualization approach is able to identify fault-prone classes, and from the point of view of the project manager, who wants to evaluate the possibility of adopting the proposed approach within his/her own organization.

Mappings between 43 bug reports and the changes that implemented the fixes were extracted from their repository and bug tracking system (available at bugs.eclipse.org). Past changes in Eclipse provide us with actual changes in the code done (i.e., classes and methods) in response to a bug. The total number of (non unique) methods changed for the 43 bugs was 144. Changed methods for a bug could be in more than one class. In particular, the classes involved in the change methods were 54.

Based on our choices and assumptions, the evaluation consists of a case study where our tool and the underlying approach are used to ease the localization of classes associated with past changes. We have studied the classes that include these changes looking at the metrics empirical validated in [28], [29], [30], and [31]. The results of these studies are summarized in Table 2. The symbol + denotes that the effect of the metric was significant in the prediction of fault-proneness class in direct way, while − in inverse way. The symbol ++ means that the metric was a more useful predictor. The symbol −− indicates that the metric was more useful in the built model, but in inverse way. The symbol 0 denotes that the effect of the metric was not significant. A blank entry means that the effect of the metric was not investigated. As shown in Table 2, only a subset of the metrics used in our approach has been empirical validated on commercial and open source software systems to determine whether they can be used as fault predictors.

To identify the classes involved in the change methods, we implemented a new feature in MetricAttitude. This feature highlights these classes in yellow. Figure 6 shows the framework Eclipse 3.5 (by applying the radial layout) and its
Table 2: Fault predictor metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>[28]</th>
<th>[29]</th>
<th>[30]</th>
<th>[31]</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMC (Weighted Methods per Class)</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>DIT (Depth of Inheritance Tree)</td>
<td>+++</td>
<td>+</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>NOC (Number Of Children)</td>
<td>++</td>
<td>0</td>
<td>-</td>
<td>++</td>
</tr>
<tr>
<td>CBO (Coupling Between Object)</td>
<td>+</td>
<td>++</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>RFC (Response For a Class)</td>
<td>+++</td>
<td>++</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>LOC (Lines Of Code)</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Figure 6: A view of the Framework Eclipse 3.5 to identify fault-prone classes

5.2 Threats to Validity

To comprehend the strengths and limitations of the empirical evaluation of our approach, the threats that could affect the validity of the results and their generalization are presented and discussed here. Despite our efforts to mitigate as many threats to validity as possible, some are unavoidable.

The second part of our evaluation is based on reenactment with user actions being simulated. Using actual developers could result to different results. We plan to conduct studies with developers in the future.

The software systems used in the study represent another threat. It was chose primarily because of the availability of data and previous studies. We cannot estimate the effect of attributes, such as, programming language, domain, and so on. With regard to the system studied, other possible threats are related to the fact that it is open source. This kind of software is usually developed by volunteers and the development process used is different from those applied in the industry.

6. CONCLUSION AND FUTURE WORK

Software visualization allows studying multiple aspects of complex problems like in reverse engineering, when comprehension tasks are performed. The available techniques and approaches are often too simplistic and lack visual cues to correctly interpret software systems [9]. In other cases, visualizations are too complex to be useful in practice.

In this paper, we have presented a visualization approach based on static analysis that easily shows complex information in OO software. Our proposal provides a mental picture by viewing a software through several views based on OO metrics and traditional code-size metrics (e.g., LOC) and relationships among classes (e.g., delegation and inheritance). To ease the comprehension of a subject system, we have chosen an intuitive mappings between the visual representation of classes and the ten metrics selected.

We implemented the approach in MetricAttitude. The current implementation of this tool only supports software written Java. However, the prototype can be extended to software written in different programming languages. For example, the use of ADT and CDT with respect to JDT will make our prototype suitable for ADA and C/C++, respectively.

To assess the validity of the approach and the supporting tool, we have conducted a preliminary empirical investigation as a case study. This investigation was performed in two steps on the framework Eclipse 3.5. In the first step, we assessed the supporting tool with respect to: scalability, navigation and interactivity, and completeness. In the second step, we verified whether our approach is useful to identify fault-prone classes. To this end, data associated with changes in response to bug reports were used. For both the steps of the evaluation, we achieved promising results.

The main future direction for our research consists in conducting users’ studies to evaluate the effectiveness of our proposal in the execution of maintenance or software comprehension tasks. These studies will also enable us to identify possible concerns in the actual use of the tool. Finally, future work will be devoted to replicate the study presented here with commercial software systems.

7. REFERENCES


