On the Design of a Multi-Perspective Visualization Environment to Enhance Software Comprehension Activities

Glauco de Figueiredo Carneiro, Claudio Sant'Anna and Manoel Mendonça

Departamento de Ciência da Computação – Universidade Federal da Bahia (UFBA)
Av. Adhemar de Barros, S/N, Campus de Ondina, 40170110 - Salvador – BA – Brazil
{glauco.carneiro,santanna,mgmendonca}@dcc.ufba.br

Abstract. In spite of all views available in Modern Integrated Development Environments (IDE), programmers still struggle to obtain all the information they need from the source code. Software visualization tools have been proposed to solve this problem. However, most of them focus on a single visual metaphor that represents the software from a specific perspective and therefore addresses a specific software comprehension issue. Real problems frequently require looking at the software from many different perspectives to address a certain maintenance task. These perspectives cannot be fully provided by only one view. It is thus desirable that an IDE provides several resources to explore the software and to configure them to the task at hand. This paper describes steps followed to design an extensible multi-perspective Eclipse plug-in for static software visualization. It offers an initial set of three perspectives and their respective views that are intended to provide a fairly broad set of software visualization resources to the IDE user.

1. Introduction

The maintenance of software systems is a challenging task. Current integrated development environments (IDEs) already offer several resources to support software comprehension. In spite of the available resources, program understanding remains a very difficult task, especially on large and complex software systems [1][2]. Consider the case where programmers want to detect occurrences of God Class code smell and therefore identify candidate classes with high complexity [9], low cohesion [9] and that use more than few attributes from other classes [9]. This is not a trivial task considering available resources in current IDEs.

In order to address this issue, this paper presents an approach that combines the use of multiple perspectives with well established interaction and navigation mechanisms from the information visualization domain such as range sliders and conceptual zooming. The main goal is to support programmers in characterizing (exploring and investigating) source code. Each perspective helps the analysis of an important object-oriented concept that used altogether and in a coordinated fashion enables programmers to examine different representations of source code. The result is an environment integrated with the IDE that enables programmers to perceive insightful relationships among software modules and their most relevant information.
2. The Motivation for a Multi-Perspective Software Visualization Environment

Important information frequently needed by programmers is not easily available in current IDEs. The software is intrinsically complex and intangible. Depending on the case, different types of information are required for achieving programmers’ objectives, such as fixing errors, changing or adding new features, or improving the code and design [1]. For example, the information about the code structure presented by the Eclipse’s package explorer (package-file-class-methods and attributes hierarchy) is useful but limited. If used alone, the package explorer may certainly be insufficient to support certain development or maintenance tasks. It does not present, for example, information about the inheritance hierarchy or coupling among the software modules. In fact, most of the modern IDEs do not effectively convey this information. When information related to inheritance, coupling or any other issue that is not explicitly supported by the IDE is required, programmers usually end up addressing this need through direct analysis of the source code, using the code editor as their main knowledge acquisition tool.

Selecting and integrating visual paradigms among themselves and with the IDE to efficiently convey relevant information. During the recent years, the software visualization community has proposed a plenty of non-traditional views to depict modules inheritance hierarchy [3], modules dependency [5], code evolution [6], and high level views of the code structure [7]. However, the majority of these views are standalone and for this reason they are not appropriate to convey all information needed to detect, for example, code smells such as God Class [9]. Moreover, most of these views are not integrated to IDEs neither provides access from the views to source code and vice versa.

Browsing, navigating, searching and filtering relevant information. Based on preliminary experimental results we have obtained on characterizing programmers detecting code smells as those presented in [1] and [2], multi-perspective visualization resources appear to be effective to bring software visualization closer to the programmer’s needs. This is intensified when providing mechanisms for browsing, navigating, searching and filtering the information provided as presented in Section 4. These mechanisms provides interaction with source code representation and hence leverage the understanding of current properties that otherwise would remain hidden.

3. A Multi-Perspective Software Visualization Model

The identification of code smells is the adopted example of a maintenance task to motivate the use of the implemented environment. Results of an characterization study of programmers identifying code smells are available at [4]. In the context of the proposed environment, a perspective is a scenario that supports programmers addressing problems related to specific object-oriented concepts. For example, two different perspectives should be designed and implemented to deal with coupling and inheritance respectively. In each of these perspectives, one or more views can be implemented to represent the concept addressed by the perspective. In the case of the perspective that represents coupling, one view may show a representation of the classes and relationships between them to represent an object-centric or data-centric view of the program using graphs while other view can represent these same relationship using
matrix. The perspectives and their respective views, if used in a coordinated fashion, should facilitate comprehension, especially if effectively combined and cross-referenced.

A prototype application was developed to implement the desired views to compose the perspectives. Based on the feedback gathered from programmers, and evaluation by means of experimental studies, we arrive at a set of recommendations regarding the design and usage of the proposed infrastructure. Before going any further, it is important to remark that this study adopts Eclipse configured for Java Language Programming as its reference IDE for discussions and examples. This choice is motivated by the fact that Eclipse is largely used in the industry and has been widely adopted in software engineering experiments reported in the literature.

### 3.1. Deriving the Perspectives in the Model

According to [9], the characterization of an object-oriented system includes at least the following concepts: (i) size and complexity, (ii) inheritance and (iii) coupling. These three concepts provide useful information for deriving three perspectives for this work: the software package-class-method structure, the inheritance-wise structure, and the software dependency perspective. The next paragraphs explain the rationale for deriving each perspective.

**The first perspective proposed is the software package-class-method.** It is related to structural representation and deals with module hierarchy and how they are organized in packages, classes and methods. Important in this perspective is the representation of metrics such as size and cyclomatic complexity of software modules. These two metrics are suitable to enrich the high-level organization of Java package-class-method hierarchies. In our example, this perspective is decisive to help programmers to spot the God Classes code smell [9].

**The second perspective is the software inheritance hierarchy.** The inheritance in object-oriented programming uses concepts of generalizations and specialization to organize abstraction and better distribute functionalities. This perspective has the goal to provide information to characterize the software in terms of inheritance hierarchy and to identify opportunities to better redistribute functionalities to other classes.

**The third perspective is the software dependency.** It addresses the issue of module coupling. The SE community has presented coupling as one of the essential reasons for software complexity. One important issue of the object-oriented paradigm is related to objects that encapsulate data to reduce coupling. This perspective aims at addressing module coupling from the perspective of the existence of relationships and its overall structure. This same perspective also addresses the issue of module coupling with information regarding the degree of coupling among modules. The excessive inter-module dependencies have long been recognized as an indicator of poor software design. Highly coupled systems, those in which modules have unnecessary dependencies, are hard to work with. They are not easy to understand, change and extend in isolation. This perspective aims at helping programmers to identify occurrences of highly coupled modules as well as providing information to understand the reasons that motivated this situation.
3.2. Mapping Software Attributes to Visual Attributes

In the case of the package-class-method perspective, the infrastructure adopted treemaps [17] as its current view (Figure 1(a)). The main reason for this choice was that treemaps is a space-filling visualization method capable of representing large hierarchical collections of quantitative data. It is a 2D visualization that maps a tree structure into rectangles with each rectangle representing a node. Treemaps are very effective in showing node attributes using size (area) and color and they enable users to compare nodes and sub-trees at varying depths in the tree. Treemaps furthermore facilitate the discovery of patterns and outliers [17].

The visual attributes of the treemaps visual paradigm are especially useful to represent the package-class-method structure perspective. The attributes that are relevant to this view are: the structure of packages, classes and methods itself, and their name and size. The elements are packages, classes and methods. They are represented as nested rectangles, where the innermost rectangles are methods and the outermost rectangles are packages. Using this metaphor, methods that are together in the same class are represented in the same rectangle. Likewise, classes that are in a specific package are represented in the rectangle, and so on so forth. A single screen shot can represent all methods, classes and interfaces in accordance with its position in the structure. Besides the nested sequence of rectangles, other visual attributes such as the rectangles’ area and color are also used to represent software attributes. The size and color of each rectangle can represent, for example, the size and complexity of each method. As a result, the size of a class will be the sum of its methods’ area whereas the size of a package will be the sum of its classes’ area.

The polymetric [10] was the visual paradigm used to represent the software inheritance-wise perspective. Originally proposed as a lightweight software visualization technique enriched with software metrics information, the polymetric view helps to understand the structure and detect problems of a software system in the initial phases of a reverse engineering process [10]. Polymetric is particularly efficient to represent inheritance trees that comprise the software system. As seen in Figure 1(a), it is a two-dimensional display that uses rectangles to represent software entities, such as classes and interfaces, and edges to represent inheritance relationships between them. The rectangles’ dimensions are used to represent properties of the entities. In our implementation, the width corresponds to the number of methods while the height to the number of lines of code of a class or interface. The color indicates the element type, i.e., whether the element is an external class, an abstract class, an internal class, a common class, an enumeration or an interface.

Graphs were one of the visual paradigms to represent the software dependency perspective. Graphs are composed of objects (nodes) and relations (links). As software visualization mostly deals with software modules and their interrelations, graph-based representations play a major role in this context. Moreover, they can be used to configure appropriated visual scenarios to spot coupling relationships that otherwise may remain hidden. For this reason, it is a suitable structure to represent dependency among software modules. However, as soon as the size of the graph and the link density increases, node-link graphs face occlusion problems due to link overlapping. Thus, it becomes difficult for programmers to visually explore the graph and interact with its elements. To tackle this problem, the visual metaphor should provide interactive
resources to paint the relationships in accordance with the criteria configured by the user. This will limit the number of nodes and links to be visualized, thus reducing the occlusion occurrences. Programmers may choose to present the graph based on the type of dependency (object, method, field, inheritance, interface implementation, interface inheritance) as depicted in Figure 1(b). Moreover, programmers can select and deselect specific nodes and visualize their coupling based on the dependency directions such as affereent and efferent coupling of selected nodes.

Candidate visual paradigms to represent the dependency strength among modules should convey them in decreasing order of dependency, i.e., programmers should be able to easily identify modules with high and low dependency values. Depending on the dependency direction, selected by the programmer, the value of the dependency strength can refer to the affereent coupling value (how many nodes depend on the presented node), efferent coupling of selected nodes (how many nodes the presented node depends on) or both. To complement this scenario, it is also advisable that another view convey the dependency strength of each module to other in the software system. This information is useful to identify classes that are more demanded or that demand more from others. This is the central point of important code smells such as God Class (GC) that is characterized by non-cohesiveness of behavior and the tendency of a class to attract more and more features [9].

A visual paradigm based on grid was the selected candidate for the view of the dependency strength perspective. The goal was to present software modules such as classes and interfaces in decreasing order, i.e., the module with the highest dependency value is in the upper left corner of the canvas whereas the lowest dependency value is in the bottom right corner as presented in Figure 2(a) and also available at [4]. Depending on the coupling option selected in the menu by the programmer, the value of the dependency strength can refer to the affereent coupling value (how many nodes depend on the presented node), efferent coupling of selected nodes (how many nodes the presented node depends on) or both. The dependency strength layout has two main goals: the first is to present an overview about the coupling degree value of all the modules that comprise a software system. The first goal is presented in a grid layout where the cells represent the modules and their position and the label in each cell represent their coupling degree with other system modules (Figure 2(a)). The second goal is to present the coupling degree of the selected node with others (Figure 2(b)). This is portrayed as a spiral egocentric graph where the central node is the module under analysis while the other peripheral nodes are positioned in accordance to its dependency strength to the central node.

The combined use of these perspectives is intended to provide a broad range of information to programmers to support them to characterize source code. The previously listed perspectives are the basis for the views to be implemented in our multi-perspective visualization environment. A wide range of visual paradigms from the information and software visualization domains were analyzed to appropriately convey the information in the views of each perspective.

The implementation of all the views required adapting and redesigning activities. In the case of the polymetric [10], the visual paradigm was already being used in software visualization, requiring only its reimplementation. In the case of treemaps [17], it has being used mostly in the information visualization domain.
For this reason, we proposed an adaptation for the software visualization domain and include it in the proposed environment. In the case of the Grid view and the Spiral Egocentric Graph, they had to be designed for the task at hand.

4. Exploring the Multi-Perspective Environment
Currently, all the views present the name of the project, the number of packages, classes and methods of the project under analysis. Programmers may need to reduce the information presented in the canvas to facilitate their understanding. Simplifications must focus on the entities of interest to the programmer. This can be done by filtering the visual scenario using some entity attributes values of interest. Filtered out software entities are usually erased from the visual scene. All the views are integrated to the same filtering engine in order to maintain consistent visual scenarios under any filtering operation.

Using Colors to Represent Module Features and Concerns: visual elements can be filled with colors representing size, complexity and element types (abstract class, external class, and interfaces). It is up to programmers to decide what information will be represented by the color that fills the visual elements. Concerns of software modules
can also be represented using colors and labels as visual cues. The term concern has been used to describe anything a developer might want to consider as a conceptual unit in a program. Ideally, concerns should be tidily encapsulated within modules with well-defined interfaces. Much of the complexity of software design can be derived from the poor modularization of concerns.

To identify occurrences of God Class candidates, the package-class-method structure is indicated to present classes and interfaces that are candidate outliers in terms of size and the realization of many concerns. Optionally, the inheritance-wise structure could also be used to identify outliers. Figure 5 portrays a possible scenario of MM version 3 where BaseController and ImageAccessor clearly stand out as God Class candidates. In Figure 1(a), the class BaseController is the largest rectangle as indicated by the arrows in Treemap and Polymetric View. Moreover, it contains methods with different concerns (colors). In the same Figure, the class ImageAccessor is also indicated by arrows in Treemap and Polymetric.

5. Related Works

Despite the relevant information provided by software visualization tools, they do not use multiple perspectives. Researchers have pointed out some of the challenges with using and bridging between multiple perspectives [11]. On the one hand, multiple perspectives can provide utility in terms of minimizing the cognitive overhead engendered by a single, complex view of data. On the other hand, multiple views can decrease utility when added to a system in terms of higher cognitive overhead needed for context switching, the time and effort required to learn the system, the load on the user’s working memory and the effort required for comparison [12]. To overcome these shortcomings, researches have proposed guidelines for designing views organized in multi-perspectives as presented as follows: (a) use multiple views when there is a diversity of attributes, models, user profiles, and levels of abstraction [13]; (b) highlight how the combined use of multiple views can bring out correlations and/or disparities to show otherwise hidden relations [14]; (c) partition complex data into multiple views to create manageable chunks and to provide insight into the interaction among different perspectives [15]; (d) make the interfaces for multiple views consistent, and make the states of multiple views consistent [13]; (e) use roundtrip visualizations so that changes to the underlying data updates the visualization and changes made through the visualization itself are reflected in the underlying data [16]. This is not an exhaustive guideline list, but it points out key issues that serve as references for us while designing and evolving the proof of concept of this work.

6. Conclusions and Future Work

This paper describes the design of a multi-perspective environment aimed at enhancing software comprehension activities. The example that motivates the use of the proposed environment is the detection of the God Class code smell. Due to the different type of information required to spot this code smell, a single visual perspective is not enough. For this reason we propose the use of multiple perspectives to give programmers the chance to configure visual scenarios that provide information useful to the task at hand.

Currently, we are conducting empirical studies to evaluate to which extent the multi-perspective visualization environment can enhance software comprehension
activities. Despite the many resources provided by the infrastructure, there still are several improvement opportunities to the proposed environment. We plan, for example, to design and implement a matrix-based representation of dependencies among modules. It may provide better insight about how modules depends one another for large projects, avoiding element occlusion in the views.

References