Supporting Software Maintenance Activities through a Software Visualization Product Line Infrastructure

Marlon Silva, Marcelo Schots, Cláudia Werner

System Engineering and Computer Science Program
Federal University of Rio de Janeiro
PO Box 15.064 – 91.501-970 – Rio de Janeiro – RJ – Brazil
{marlon,schots,werner}@cos.ufrj.br

Abstract. Different methodologies and tools have been proposed in the software visualization field to improve the understanding about system properties, one of the most difficult and expensive activities during software maintenance. Nevertheless, the construction of these visualization mechanisms is considered of high complexity, given the necessary knowledge, effort and time for their development. In this sense, this paper presents an infrastructure for building software visualizations, which is based on a so-called Software Visualization Product Line (SVPL), aiming at providing greater flexibility in creating and combining appropriate visualizations to support software maintenance activities, with less time and effort in its construction.

1. Introduction

Support for software understanding is needed for several software engineering activities (Schafer & Mezini, 2005). Nowadays, computers have become important tools for creating visualizations, helping users to better understand complex phenomena. Software visualization plays a crucial role in supporting human reasoning. It helps information understanding through visual metaphors and computational techniques.

The visualization process basically involves three main steps (Diehl, 2007): (i) Data Acquisition: there are several sources of information about software, and this leads to different ways to extract it; (ii) Analysis: typically, there is the need for a processing step to reduce the amount of information to be presented to the user, based on the individual focus; (iii) Information Representation: data from the previous steps are mapped into graphical information to be rendered in some output device.

In this context, software visualization aims to develop approaches and tools to provide such support for information comprehension. However, current attempts have generated tools that support very specific tasks, not reaching a wide range of needs, or are too general, not providing enough assistance to more complex activities (Schafer & Mezini, 2005). So, they are used in an isolated way, since their integration and adaptation to attend different scenarios imply more cost and effort.

Thus, there is a lack of mechanisms that offer flexibility to software stakeholders in customizing their visualizations, so one can focus on relevant data and information to improve the understanding of their activities. Since visualization development involves high cost, complexity and time-consuming tasks (Anslow et al., 2008), reuse and automatic generation of these mechanisms are important features to an efficient use. On
the other hand, comprehension is an influencing factor in software maintenance (Mayrhauser & Vans, 1995), which indicates the need for mechanisms for improving understanding in this field.

Thus, in order to allow a faster, easier construction of visualization mechanisms for supporting common activities in the software maintenance field, this work presents a Software Visualization Product Line (SVPL), a SPL which embodies software visualization concepts and techniques, focusing on the software maintenance domain.

This paper is organized in the following sections: Section 2 presents related work; Section 3 describes the overall approach and involved concepts; Section 4 details the infrastructure developed to support this work; Section 5 concludes this paper and discusses future work.

2. Related Work

Many Eyes (Viégas et. al., 2007) is a website that allows users to submit data, create visualizations and share results. Its goal is to support collaboration around large-scale visualizations by fostering a “social style” of data analysis, in which visualization not only serves as a discovery tool, but also as a means to stimulate discussions among users. Despite Many Eyes has a friendly user interface for selecting and sharing data and visualizations, the tool offers little flexibility, keeping a fixed, limited way for data acquisition; also, it does not provide an intermediate step for processing the raw data before mapping it into visual objects.

Mondrian (Meyer et. al., 2006) presents a script-based approach for visualization prototyping. This tool promotes great flexibility in creating visualizations, but it relies on a proprietary declarative language. Thus, there is a “learning overhead” for assimilating the language. This is a considerable drawback, especially when it is considered that users will be stakeholders with different kinds of interests and expertise.

CogZ (Falconer et. al., 2009) is a set of tools that provides a module for the rapid development of specific visualizations for ontologies. This approach facilitates the user interaction by providing drag and drop mechanisms for mapping the concepts (ontology terms) to visual representations. Apart from the benefits and the power of customization with a friendly interface providing great interaction, the tool offers little flexibility, keeping a fixed, limited way for data acquisition; also, it does not provide an intermediate step for processing the raw data before mapping it into visual objects.

Luthier (Campo & Price, 1998) is a framework designed to support the construction of visual tools for dynamic program analysis. As a framework, this work allows the construction of specialized mechanisms based on extensions and interface implementations of its structure. Through this organization, it is possible to build highly-complex visualizations; however, it needs specialized people to operate on the source code level, which may involve time and cost.

From these studies, it is observed that a high flexibility in creating visualizations is usually accomplished by means of artifacts closer to the source code, such as scripts and frameworks. These types of features create a dependency of experts in its use, apart
from the time-consuming nature of this activity. In contrast, other researches propose an interaction on a more abstract level, approaching natural human reasoning, which makes it easy to use, but they do not always achieve high flexibility (e.g. Many Eyes), or require specific knowledge (e.g. CogZ and its metamodel background).

The infrastructure presented in this paper aims at the configuration and automatic generation of software visualization mechanisms, allowing the maintenance stakeholders (developers, testers, development managers, systems analysts, etc) to build visualizations faster, with high flexibility (both in data acquisition and visual representation sides), in order to achieve the desired level of comprehension.

3. Software Visualization Product Line

Software Product Lines (SPL) can be defined as a paradigm of software development using a core of common artifacts for mass customization. It is responsible for reuse and cost reduction in product generation over time, by providing product variability and adaptation for specific scenarios (Clements & Northrop, 2002).

These benefits facilitate and encourage the application of visualization on activities that demand software understanding and comprehension, such as those related with software maintenance (e.g. code evolution analysis, metric analysis, software team collaboration analysis, bug’s life exploration, etc). As software maintenance is a very extensive area, this paper will focus in a portion denoted by software evolution analysis and, inside this field, the structural, metric and collaboration analysis (there is ongoing work in our group studying the debugging analysis).

The next section describes the development of a Software Visualization Product Line (SVPL), with the purpose of supporting software maintenance tasks through fast generation, combination and integration of software comprehension mechanisms based on visualization techniques.

4. SVPL Infrastructure

Fig. 1 presents an overview of the approach, which comprises three main activities of an SVPL: domain analysis, domain instantiation (represented by the component development), and application instantiation (denoted by the visualization wizard). Drawing a parallel with the classical stages of SPL creation, the first two activities represent Domain Engineering (DE), while Application Engineering (AE) encompasses the latter. This structure also creates a separation of roles (domain engineer, component producer and software maintenance stakeholder) among SVPL activities, in order to minimize and simplify the steps needed by the software maintenance stakeholders.
4.1. Domain Analysis (DA)

DA is an activity of DE in which SPL features and variability are defined and designed (Clements & Northrop, 2002). DA is generally composed of steps such as the definition of the product line scope and requirements engineering. For the software maintenance domain, requirements engineering was conducted by means of a literature review on studies focused on software maintenance tasks, comprising corrective, adaptive, perfective, and preventive activities (Storey et. al, 2008; Trümper et. al., 2010; D'Ambros et. al., 2007; Voinea & Telea, 2007 etc.). These studies will not be detailed here due to space limitations.

Summarizing, these studies mainly address structural analysis, metric analysis, debugging analysis, behavior analysis and collaboration analysis in the field. This research guides the construction of a Feature Model that is used in the remaining phases of SVPL. Fig. 2 shows a partial view of the feature model, which was developed in Odyssey environment (Software Reuse Team, 2011), a domain engineering environment. Odyssey can also be used for future maintenance and evolution of the feature model. The notation used is denoted Odyssey-FEX and is described by Oliveira (2006). Dotted rectangles represent optional features; stereotypes denote if a feature is conceptual, functional or a parameter for a specific operational environment; letters in the rectangles’ lower right corner show rules in which that feature takes place.

![Figure 2. Partial view of the feature model.](image)

This model is based on three mandatory features: the Data Provider, a conceptual representation of information data source; the Data Extractor, a technical mechanism that extracts data; and the Visual Representation, which is a metaphor used to represent data that elicit information and understanding about a topic. There is an optional feature, named Data Processing, which can be used to process the extracted data before transforming it into visual objects. These features may be interrelated by composition rules (e.g., inclusive relations) defined in the Odyssey environment.
4.2. Component Development

Being an SPL, SVPL needs a core of components that realizes the features designed in the feature model. To this end, it is necessary to select an architecture that can accommodate (and be compatible with) the built components. These requirements are met by EvolTrack (Werner et. al., 2011), an Eclipse-based visualization tool with a flexible architecture, whose components are classified into data source connectors, transformers, and view connectors. These categories map to the concepts presented in the previous section. EvolTrack’s kernel orchestrates and controls the information flow.

The next step was the construction of a component generator, whose main goal is to establish coding and structural patterns for components. Besides, it accomplishes a set of tests, documentation and pre-established implementation points for component producers, allowing adjustments according to the functions to be implemented. This accelerates the process and increases the variability and quality of the SVPL.

The component generator works through a UML-based XMI model, where the components are designed as classes in which attributes and associations match pre-configuration points. This model can be built with any modeling tool which is able to export models in XMI format (versions 1.1, 1.2 for UML 1.4; 2.0 for UML 2.0). Fig. 3 shows two types of components in the same model: one for visual representation (labeled ViewConnector), and the other for data acquisition (labeled DatasourceConnector). The former is associated to two view modules, i.e., visualization modules of a connector that can act alone or together to present information. This XMI model serves as the input for the component generator, which processes it with a template-oriented engine. After that, it generates the source code with unit tests, documentation and implementation points for those connectors, respecting the established patterns and good practices.

![Figure 3. Component generator XMI model example](image)

When the connectors are ready for use, they are stored in a repository, like dependency repositories\(^1\), for the use of the Visualization Wizard (which will be detailed in the next subsection). Some of the components used by this work were developed by

---

\(^1\) more details in [http://maven.apache.org](http://maven.apache.org)
previous approaches of our group, such as PREViA (for co-evolution and static analysis through metric collection and model comparison), EvoITrack-SocialNetwork (for social network analysis) and EvoITrack-VCS (for Version Control Systems extraction) (Schots et al., 2010; Werner et al., 2011). Other components for reuse analysis through metrics and debugging analysis through bug databases data extraction are under development by our research group.

4.3. Visualization Wizard

This module deals with some technical aspects of AE, seeking to provide a mechanism that provides a way of selecting and configuring features from the feature model arising from DA, transparently to the stakeholder. By means of a user interface, the visualization wizard allows the features selection, taking into account the constraints and consistency rules (which are mapped in the feature model), to prevent the generation of inconsistent products. Due to its high level of abstraction, the wizard replaces the need for stakeholders to know the components responsible for the features implementation. Thus, this layer integrates the feature model and the component repository, i.e., it retrieves the necessary components based on the selected features, makes their composition and delivers a visualization package.

This package encompasses the needed components (adapted and configured) that run into an Integrated Development Environment (IDE). This is due to software maintenance activities being usually in contact with software development artifacts. In this approach's context, the Eclipse IDE was chosen as the proof of concept to run the visualization package. The wizard workflow is as follows: it receives the feature model as input for controlling the features’ constraints, compositions and configurations and allows users to make their visualization features choice; based on the stakeholder’s selections and options, it verifies whether the composition rules of the selected features match the original design; if so, the wizard retrieves the components (the information about which components implement a feature is also provided in the feature model designed by Odyssey), configures them and packs them into a visualization package ready for use in Eclipse IDE (i.e., an Eclipse plug-in).

In particular, the selection made by the stakeholder can be done in 2 steps: the first one is a boolean selection made with checkboxes (and the wizard interaction monitoring the composition rules), which is simpler for the stakeholder to understand and use; the second uses a form to allow the stakeholder to fill in parameters that specializes the operation of his/her visualization. These parameters are those defined as operational environment features in the feature model (it means that the second step is not mandatory) and allows for a more detailed visualization configuration. Fig 4 illustrates the 2 mentioned steps.

In a hypothetical scenario (scenario 1 - metric analysis), a software engineer (SE) would want to analyze a structural software metric instantly during the software development. So, he would choose the Instant Metric Speedometer, Process Structural Metrics and VCS features in the model. When he/she selects the VCS feature, the wizard warns (through the composition rules) that the source code extractor is necessary. The resulting visualization is presented in the left side of Fig 5.
Optionally (scenario 2), if the SE wants to check the overall dimension of a huge system structure, he/she can select the VCS and 3D City Metaphor features to obtain a 3D city, in which packages are areas (rectangles), classes are buildings, and interfaces are pyramids, with zoom, drag and rotate interaction resources. Possibly, he/she can increment this visualization with some kind of metric analysis, represented by color and size of the city construction. Fig 5 (right side) shows the visualization for this scenario.

5. Conclusions and Future Work

This paper presented an SPL-based infrastructure that uses an SVPL approach to generate flexible software visualizations, which is expected to meet software maintenance stakeholder's comprehension needs. Although other work tried to generate flexible visualizations, they usually require expertise knowledge for stakeholders to operate them. More components are being built to meet other requirements (like reuse metric analysis, bug tracker analysis, etc.) and expand the product line. Further steps include the planning and execution of an observation study for evaluating if the proposed approach achieves the goal of building visualizations in an easy way, in less time, and with enough flexibility to meet stakeholders’ needs.

References


