Visualizing Software Evolution with Lagrein

Andrejs Jermakovics
Free University of Bolzano-Bozen, Italy
andrejs.jermakovics@unibz.it

Raimund Moser
Free University of Bolzano-Bozen, Italy
raimund.moser@unibz.it

Alberto Sillitti
Free University of Bolzano-Bozen, Italy
alberto.sillitti@unibz.it

Giancarlo Succi
Free University of Bolzano-Bozen, Italy
giancarlo.succi@unibz.it

Abstract

Lagrein is a tool that allows exploring how a software system has been developed. It supports visualization of multiple metrics, it links requirements to code expected to implement them and couples code with the effort spent in producing it. Moreover, it shows the system's evolution using animation and timing diagrams.

Categories and Subject Descriptors D.2.8 [**Metrics**]: Process metrics, Product metrics; D.2.9 [**Management**]: Life cycle, Time estimation;

General Terms Measurement.

Keywords Software metrics visualization; software maintainability

1. Introduction

Lagrein [2] is a system that supports managers and developers in exploring how a software system has been developed. It is able to identify which components required more effort or were more complex to develop. Following the path laid out by tools like Code Crawler [3], it supports the visualization of multiple metrics at the same time supplemented by three major features: 1) it links individual requirements to portions of the source code expected to implement them; 2) it couples the source code with the effort spent in producing it; 3) it visualizes the overall evolution of a software system. Such features allow the manager to measure, with a certain level of approximation, the effort required to implement single requirements and to understand the state of health of a software system.

2. Functionality

Lagrein provides the following main functionalities: code visualization and navigation, requirements navigation, and effort visualization. We visualize code using various polymetric views [3] supplemented by our extensions. Moreover, Lagrein enables the visualization of the evolution of a software project both using animations and timing diagrams.

Requirements Navigation - Lagrein imports the requirements from requirement management tools like RequisitePro or from text documents where the requirements are properly separated, either using special paragraph tags or using special characters (in textual files).

Requirements are linked to the relevant sections of the source code and to the associated effort by using algorithms for Latent Semantic Indexing (LSI) as revised by De Lucia *et al.* [1]. When one or more requirements are selected, the classes that are estimated to implement such requirements are highlighted. The opposite navigation is possible as well.

Development Effort Visualization - Lagrein provides four different types of views on how effort is distributed among class methods, developers, requirements, and over a period of time.

The Method Effort View represents the distribution of development effort among the methods of a class. It uses a tree map to represent each method as a rectangle, whose size is proportional to the effort and color to LOC. An analogous view shows how much time each developer spent on that class. The Requirements Effort view shows how much time has been spent on implementing each requirement using a bar chart and the Requirement timeline shows periods of each requirement's implementation thus allowing to recover the development process and product evolution. When effort for a new requirement is estimated, its development period is visualized together with the already implemented requirements.

Animation of Software Evolution - Apart from the visualizations of code structure, the tool offers a diagram of relationships. It uses a graph of entities (classes, packages, methods) and shows relationships among them as edges (Figure 2). The types of relationship range from direct ones (method calls, attribute access, inheritance) to more sophisticated (co-changed, implement one requirement). Since different kinds of relationships have a different kind of strength, we want the stronger related entities to be shown closer together and vice versa. To achieve this, we use a spring layout algorithm, which is a force-directed algorithm that models all edges as springs having physical forces of attraction and repulsion.

Having the related entities together makes it easy to spot highly coupled parts or clusters. Groups of highly coupled classes could indicate modules of the system, while a pair of such classes could suggest Inappropriate Intimacy code smell. A single class that is coupled to many other classes (high CBO) can suggest a

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-220-7/08/10.

candidate for refactoring, since it can be a possible source of bugs.

Animations are well suited to demonstrate the overall time evolution of a software project; however, they are not suitable to understand evolution patterns and their impact on software quality attributes such as maintainability. We propose a gradient color technique for displaying the time dimension. The approach is very simple: for example, considering single classes, represented as cubes such as in Figure 1, we would like to display the evolution of their cyclomatic complexity, CC, over 3 releases of the software. To do so we compute for each class the minimum and maximum of the CC within the requested time period and use a color gradient to indicate the change of CC from one release to the next. Small CC values are displayed light-colored, while high CC values are dark-colored. We can observe that the class in the middle in Figure 1 has a stronger increase of the cyclomatic complexity from version 2 to version 3 than the other 2 classes. Such visualization helps to spot classes that tend to be difficult to maintain and deteriorate the overall maintainability of a software system. Moreover, we can understand at a glance the time evolution of a given maintainability metric: does it fluctuate and is it unstable? Does it grow continuously? Or does it converge to an upper limit or even decrease? Such patterns are important to assess not only the current state of a software system, but also to predict its future evolution. They allow identifying problematic areas of a software system and helping to decide what maintenance actions should be envisioned: refactor the code, let it unchanged, or eventually rewrite it.

3. Architecture

Lagrein enables the visualization of data regarding the software development process, the software product, and its evolution. Data are extracted from a non-invasive measurement tool, PROM (PRO Metrics) [4]. It collects process metrics by using plug-ins for mass-market applications such as office automation suites, software development tools, etc. Product metrics are collected from version control systems (the system supports CVS, Subversion, and Microsoft Team System). Lagrein is based on the Eclipse RCP (Rich Client Platform), which consists of a minimal set of plug-ins needed to build a rich client application based on the Eclipse runtime.

4. Demonstration

We want to demonstrate how Lagrein can be used to analyze and monitor software maintainability using visualizations quickly and with ease.

We start off with visualizations involving several maintainability metrics such as coupling and complexity metrics. We will try to answer the following questions: "Which classes are most coupled with other classes?" and "Are those classes difficult to test and/or modify due to their high complexity?". We will show a view with all classes represented as cubes and having coupling as the width metric, complexity as the depth metric, and the color as size metric (LOC). Such view shows us the classes, which are likely to require a high maintenance effort. However, by experience we know that complex classes do not necessarily mean high maintenance effort if they are designed well. We need additional information in order to evaluate maintainability of a software, namely its past evolution.

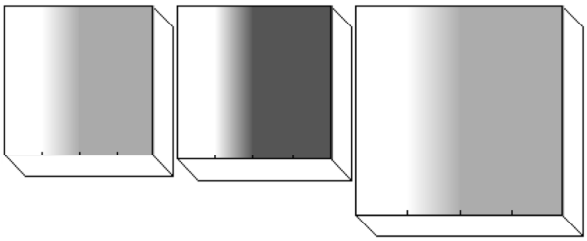

Figure 1: Hot spot view with gradient color visualizing the cyclomatic complexity.

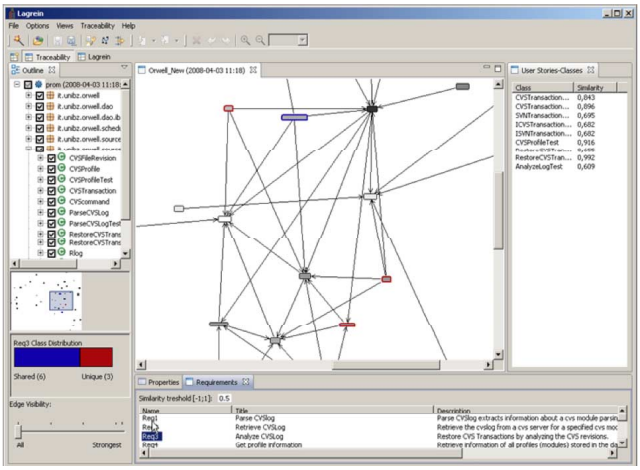

Figure 2: Screenshot of Lagrein.

To visualize the evolution of maintainability indicators we use the gradient color technique. We augment our previous visualization with a gradient color that indicates the evolution, i.e., increase or decrease, of the respective maintainability metric for each version of the software. In this way we can easily spot patterns that indicate bad maintainability: continuous and fast increase of a specific metric, spikes and instability. On the other hand, if a maintainability metric decreases over time and eventually converges to some stable value – even if it is relatively high – we may assume that the corresponding classes are easy to modify and to maintain.

References

[1] De Lucia A., Fasano F., Rocco O., and Tortora G. "ADAMS Re-Trace: a Traceability Recovery Tool", *9th Europ. Conf. on SW Maintanance and Reeingineering (CSMR 2005)*, March 2005.

[2] Jermakovics, A., Scotto, M., Succi, G. "Lagrein: tracking the software development process", OOPSLA Companion 2007: 882-883, 2007.

[3] Lanza M., Ducasse S. "Polymetric Views—A Lightweight Visual Approach to Reverse Engineering", *IEEE Trans. on SW Eng.*, 29(9):782-795, 2003.

[4] Scotto M., Sillitti A., Succi G., Vernazza T. "A non-invasive approach to product metrics collection", *Journal of Systems Architecture*, 52(11): 668-675, 2006.