CodeGestalt
IDE Visualization Features for Program Understanding

Christopher Kurtz
Media Computing Group
RWTH Aachen University
christopher.kurtz@rwth-aachen.de

INTRODUCTION

Software maintenance and reverse engineering require a lot of effort, usually more than the original design and programming. One of the primary problems is the time developers spend understanding the existing code base, even if it is their own with which they did not work recently.

To help programmers to easily familiarize themselves with unknown or legacy code bases, the *CodeGestalt* project aims at developing an intuitive interface to effectively search, explore and understand even large projects.

RESEARCH QUESTION

Several studies (e.g. [1], [2]) have shown that the key to fast and effective understanding of source code is an IDE's ability to let the programmer explore the code. Today's most prominent IDEs have certain drawbacks in that regard. Especially the navigation involved in browsing a project consumes a lot of time.

Further, several visualization techniques have been proposed in recent years, but their actual usability is not well understood. Studies like [9] suggest that well integrated IDE plug-ins providing thoughtful visualizations are most capable in supporting source code understanding.

CodeGestalt is targeted at closing these gaps and enhance an IDE via its plug-in infrastructure. It is not finally decided for which of the following IDEs to develop the prototype.

- *Xcode* is the default IDE at the Media Computing Group. This fact would simplify finding participants experienced in the basic IDE functionality for user tests.
- *Eclipse* is the IDE on which the majority of recent visualization tools from the research community and studies are based. Therefore this platform would make comparative performance evaluation easier.

The overall goal is to determine how to improve the IDE in terms of code understanding. This covers primarily how to visualize large code bases and secondarily how to improve IDE navigation.

Initial literature research has suggested that the following properties would be desirable:

- Structural information about the code base
- Detection of semantically related code artifacts
- Ability to search and refine the visualization
- Visualizations with different levels of detail and abstraction
- User controlled visualization process with software support to minimize effort
- Good IDE integration to improve user acceptance and minimize learning effort

RELATED WORK

On the one hand, the problem of program understanding is quite well studied. On the other hand many interesting visualization techniques have been developed, however not all of them are well understood in terms of usability and whether they are suitable to close any gaps or just add another layer of confusion to the program understanding task.

The Shortcomings of Today's IDEs

In [1] Ko et al. demonstrated that developers essentially need to spend a third of their activities on IDE mechanics such as navigation between source files with a lot of time wasted due to overhead.

Similarly Jonathan Sillito showed in [2] that it is hard for programmers to answer high-level questions about a project and that there is essentially not sufficient tool support for those kinds of tasks. Among other findings one of his testers complained about not being able to correlate two or more search results.

In [7] Park and Jensen compared several toolsets and how they enable users to explore a previously unknown code base. The results suggest that good software visualization tools improve the quality but not necessarily the performance of programming.

Relo

To limit the overwhelming size of automatically generated diagrams, Sinha et al. [6] developed the *Eclipse* plug-in *Relo* that allows users to build a class diagram step by step. The software automatically parses the codebase and therefore can offer the user options how to expand the diagram he is working on.

The interface and interaction is heavily software supported, but in contrast the resulting visualization is user driven. Therefore the user has much more control on what implementation aspects are so important to be covered in the diagram and what details to omit compared with fully automated visualization techniques.

Thematic Software Maps

Kuhn et al. developed a "topological" way to visualize large code bases in *Thematic Software Maps* [3]. Using simple heuristics the similarity of classes based on the words used in the corresponding source files is determined. This data was then projected onto a 2D plane and a height map was constructed assuming hills with a normal distribution shape and a size depending on the LOC of the corresponding source file.

The MDS algorithm used to flatten the data to 2D has the interesting property of providing relatively consistent positions for single classes when observed over the evolution of a project.

3D Visualizations

Fronk et al. presented a visualization technique that is aimed at making clear the structure of code bases. Their visualization uses the additional degree of freedom granted by the third dimension to cover class hierarchy, class dependency and package containment relations in a single visualization. In [4] they demonstrate that these *3D Relation Diagrams* can outperform classical UML class diagrams for increasingly large projects.

Another semi-3D visualization is the *CodeCity* introduced by Wettel and Lanza in [5]. Here classes are drawn as buildings organized in blocks representing packages. The size of buildings is determined by their memory requirements and the number of methods they provide. The system also allows highlighting parts of the city to visualize search results.

Young and Munro [8] suggested a visualization called *CallStax* that decomposes a call graph in all distinct paths starting at the root (e.g. `main`) and visualizes them as 3D stacks of colored boxes, where each color is associated with one specific function. This visualization can than be interactively filtered e.g. by showing only those stacks, that contain a selected function.

PROJECT SCHEDULE

Development starts with literature research and an initial exploratory examination. In order to learn about how programmers use visualizations (e.g. pen and paper sketches) in their everyday work, an online survey will be conducted over the period of ten days involving programming communities of different platforms. The survey is supplemented by qualitative interviews with developers (mainly *Xcode* users) from the university's staff. This initial phase is supposed to take four weeks.

From the analysis of these questionnaires a more concise concept for CodeGestalt will be derived. Where necessary, follow up inquiries and interviews are conducted to validate the analysis. This conceptual phase is supplemented by consulting related research and will probably require another month.

The *CodeGestalt* concept will be refined using paper and software prototypes in an iterative DIA cycle. Ultimately this process is aimed at creating the most sophisticated software prototype possible within ten weeks. The prototypes' usability and competitive performance will be evaluated by several user tests.

The remaining time will be used writing out the diploma thesis.

REFERENCES

1. Andrew J. Ko, Brad A Myers, Michael J. Coblenz and Htet Htet Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering 32* (2006), 971-987.
2. Jonathan Sillito. Asking and Answering Questions During A Programming Change Task. 2006.
3. Adrian Kuhn, Peter Loretan and Oscar Nierstrasz. Consistent Layout for Thematic Software Maps. 2008.
4. Alexander Fronk and Dietmar Gude and Gerhard Rinkenauer. Evaluating 3D-visualisation of code structures in the context of reverse engineering. In *Proceedings of the Workshop on Empirical Studies in Reverse Engineering (WESRE)* (2006), IEEE Computer, Society Press.
5. Richard Wettel and Michele Lanza. Visualizing software systems as cities. In *Proc. of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, Society Press (2007), 92-99.
6. Vineet Sinha, David Karger, Rob Miller. Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases. Visual Languages and Human-Centric Computing (VL/HCC 2006).
7. Yunrim Park, Carlos Jensen. Beyond Pretty Pictures: Examining the Benefits of Code Visualization for Open Source Newcomers. 2009.
8. Peter Young and Malcolm Munro. A New View of Call Graphs for Visualising Code Structures. *Computer Science Technical Report* 03/97.
9. Mariam Sensalire, Patrick Ogao: Classifying Desirable Features of Software Visualization Tools for Corrective Maintenance. In *Proceedings of the 4th ACM symposium on Software visualization* (2008), 87-90.