Code Gestalt: A Software Visualization Tool for Human Beings

Christopher Kurtz
Media Computing Group
RWTH Aachen University
52056 Aachen, Germany
christopher.kurtz@rwth-aachen.de

Copyright is held by the author/owner(s).
CHI 2011, May 7–12, 2011, Vancouver, BC, Canada.
ACM 978-1-4503-0268-5/11/05.

Abstract
Programmers are often faced with the necessity to visualize source code and grasp its structure. In a survey we studied how developers deal with this task. Based on our findings, we present the software visualization tool *Code Gestalt*, which assists programmers in quickly creating class diagrams. We evaluated and refined our concept using two prototypes. As a result, Code Gestalt introduces the *tag overlay* and *thematic relations*. These augmentations to class diagrams display similarities in the vocabulary used in the underlying source code. This simple, yet effective toolset empowers the user to explore and visualize software systems. The preliminary results of a user study investigating Code Gestalt indicate good usability.

Keywords
Software visualization, tag cloud, class diagram, survey, prototype, evaluation, user study

ACM Classification Keywords
H5.2 Information Interfaces and Presentation (e.g., HCI): User Interfaces. Graphical User Interfaces.

General Terms
Design, Human Factors

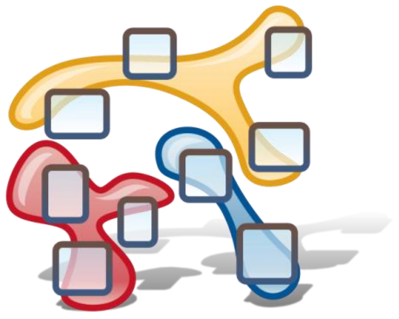

Figure 1. Code Gestalt builds upon the widely used visualization of representing types as boxes. We augment these class diagrams by allowing users to search for and visualize regions in the diagram that deal with common themes or concepts such as "network", "undo", or "update".

Introduction

Roughly outlining the structure of a code base or a software feature is frequently achieved using *sketches*, such as pen and paper or whiteboard drawings (see *Online Survey*). Although there are many tools dedicated to the task of *software visualization* (SV), they are not widely used in everyday development [9]. We investigated why the computer does not play a more pivotal role in creating quick visualizations, and how a software tool has to be designed to become more attractive for programmers.

Over the course of our exploration, we found that most SV tools fall into one of two categories: One type is represented by graph-based applications (like [13]), which focus on the syntax and structural properties of source code, but lack the ability to emphasize what a human developer would deem important. The other type of tools visualizes metrics (like [5]) to identify those parts of a code base that may be of interest to a programmer. These tools usually create rather static SVs allowing for little or no user customization.

Code Gestalt (CG) is our concept to bridge these two approaches. The user is able to create and edit graph-based SVs (we use class diagrams as familiar base-line) and augment them with more semantic information that emphasizes important regions according to the user's interests (see **Figure 1**).

To achieve the latter, we focus on the vocabulary of the source code, since this allows us to exploit the human understanding existent in the naming of variables, methods, and type names [12]. Making this source of information easily accessible for other developers helps them to carve out the overall gestalt of a code base.

Related Work

Several surveys and evaluations have analyzed and compared the capabilities of SV tools in the past. A user study by Park and Jensen [6] suggests that SV tools help newcomers to an open source project to get started. Bassil and Keller [1] conducted a survey among 107 participants to determine which aspects of SV tools are important to users, and what disparities exist between user needs and features in available SV applications. They identified that the interface and usability did not match the users' expectations. In 2007 Sensalire and Ogao [9] asked five professional programmers to evaluate three representative SV tools. Their study revealed a gap between what the expert users desired and what was offered by the tools in the areas of IDE integration, search functions, simplicity, and flexibility.

Several attempts have been made to incorporate "human insight" in SVs and to simplify diagram generation. Sinha et al. [10] presented *Relo*, an editor that allows the creation of partial class diagrams. The user adds types and members to a diagram by expanding call and inheritance relations of existing elements. That way, only code artifacts selected by the user are visible and the user controls layout and scope of the diagram as it grows. A completely different approach is taken by the *thematic software map* by Kuhn et al. [5]. In this SV types are represented as hills on a map, where those with similar vocabulary are placed close to each other. Several overlays can be displayed on this map, e.g. call relations and search results.

Our work introduces new visualization and interaction techniques to integrate the advantages of a

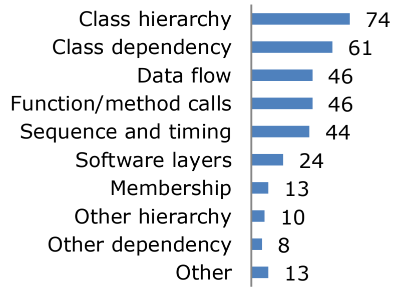

Figure 2. The responses from 112 participants to the question "What aspects of your software project or code artifacts do you usually cover in sketches?" Participants were allowed to give multiple answers. 16 participants stated to never sketch.

customizable graph-based SV with those of metrics-based SVs, namely finding thematic and conceptual similarities in types using tag cloud algorithms.

Online Survey

Former surveys and evaluations describe advantages and deficiencies of existing SV tools. They do however not answer the question how programmers accomplish visualization tasks in lack of a suitable SV tool.

We performed an online survey that contained 31 questions, asking about the participant's background, impression of common and experimental visualizations, and working experience with SV tools, sketching, and visualizations in source code documentation. Links to the survey were distributed through international programming forums and mailing lists. We recorded a total of 128 participants: 67 students, 42 professional programmers, 11 researchers, 2 teachers, and 4 persons from other areas of software development. 35% of the participants sketch at least once a week, while only 20% do use an SV tool that often. Class diagrams were by far the most popular SVs, rated "useful" by 80% of the participants. **Figure 2** illustrates what aspects of a code base users sketch, when they do not use an SV tool.

We found that SV tools must allow for fast SV generation, as time consumption was a primary reason for many users to completely avoid SV tools. Qualitative feedback suggests that an adequate SV tool should give the user control over the visualization and not be a "one-click-solution": In open-ended questions, participants commented that some tools "tend to become unreadable", and automatic SV tools "put too much detail in the diagram". For details refer to [6].

These findings guided our development of CG. The tool should support the user in creating an SV step-by-step based on a class diagram and offer features to discover and highlight important features.

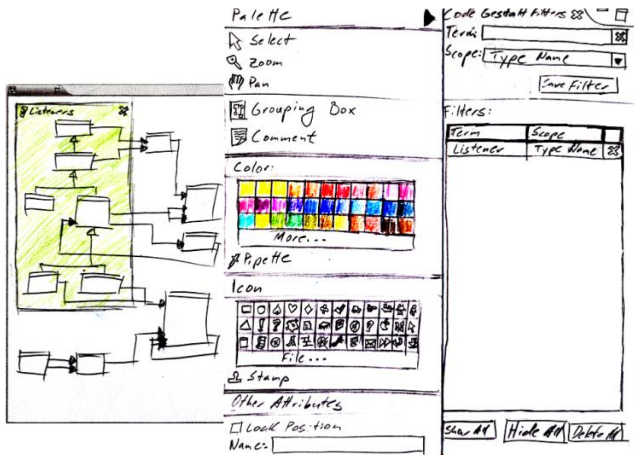

Figure 3. A mockup UI from the paper prototype. The editor (left) is inspired by Relo [10]. The user can perform searches and define filters (right) that impact what elements are visible in the editor. Using the tagging interface (center), the user can assign colors and icons to elements, thus grouping related code artifacts.

The user has identified a group of listeners and tagged them with the icon of an ear and a shade of green.

Paper Prototype

For the first iteration of the CG concept we built a paper prototype (see **Figure 3**). We looked at some code bases and found that naming conventions could be exploited to gain a certain degree of "human insight" [12], if the vocabulary used in types could be visualized. E.g., terms like "message", "server", and "port" have a high probability of appearing in the implementation of network features.

For this purpose we added a visual filter feature to an editor inspired by Relo [10], and provided additional IDE integration. In this concept a user confronted with an unknown code base can visually filter the diagram using one or more search results to find types that share common themes. That way, the programmer is

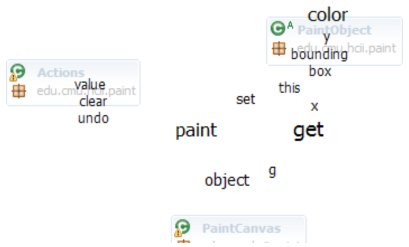

Figure 4. A small tag overlay for three classes: *Actions*, *PaintObject*, and *PaintCanvas*. The user can identify the "center of gravity" for concepts, such as *undo*, which is an important term in *Actions*. The font size of each tag is determined by the mean of the weights assigned by each type (term frequency in type).

This naïve algorithm causes an overexposure of terms like *get* and *object*. In future work we want to examine different tag cloud metrics and filters to reduce clutter.

able to identify related elements, which can be grouped and tagged using colors and icons to create landmarks.

We tested the prototype with five students using twelve use-cases. The testers had some conceptual problems with the visual search/filter feature and asked for a simplified interaction.

Tag Overlay Design

To replace the visual search feature we conceived the *tag overlay* (see **Figure 4**). It is designed to help the user build a spatial model of the source code ("the network code is in the top-right") and emphasize important aspects.

In CG the user creates a diagram along the lines of structural relations such as inheritance and method calls. The tag overlay is an optional visual layer that can be toggled by the user. It displays a tag cloud on top of the existing diagram. The terms used for the tag cloud are parsed from the identifiers of each type. First, we construct tag clouds for individual types by assigning each term[1] a weight based on its frequency. The font size of a tag in the overlay represents the mean of these weights. Similarly, the position of a tag is the "center of gravity" between the type boxes in the diagram with respect to these weights. The overlay is dynamic and updates as the user moves, creates or deletes types from the diagram.

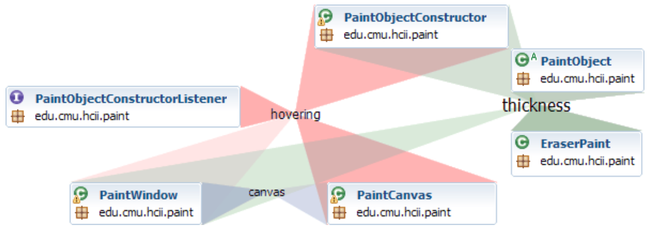

Figure 5. Three thematic relations: The tags *canvas*, *hovering* and *thickness* connect those types that use these terms in their source code vocabulary. The intensity of the fan segments indicates the relative frequency of the term in each type.

[1] We differentiate terms (non-weighted) and tags (weighted).

This can be seen as an inverse of the approach taken by the thematic software map [5]. We augment our SV with spatial and thematic information extracted from the vocabulary of the source code without sacrificing the flexibility of the underlying graph editor. Using this technique, CG gains some of the semantic expressiveness desired by the participants from our survey, without becoming a static "one-click-SV".

When the user selects a tag all types are highlighted that use the corresponding term. The intensity of the highlight color represents the weight assigned to the term by each type. Vice versa, the user can select a type to highlight all tags which terms are used in the selected type. Again, the color intensity of the highlight visualizes the corresponding weight. A programmer can explore the code base and determine what types deal with what themes by selecting corresponding tags. Highlighting tags by selecting terms on the other hand gives the user a better idea of its thematic scope and in what parts of the diagram related types can be found.

While this overlay already aids in the analysis of the code base, it is transient and does not interact with the underlying class diagram. To include a tag in the diagram, the user can create a *thematic relation*. Thematic relations connect all types using the same term with the corresponding tag. This relation is visualized as a fan (see **Figure 5**) and persistent with respect to the tag overlay toggle. We assign different transparency levels to the fan segments representing different weights. Thematic relations are recognizable landmarks that add structure to the diagram.

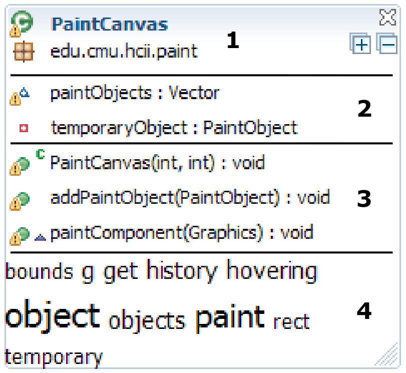
PaintCanvas
edu.cmu.hci.paint
1
paintObjects : Vector
temporaryObject : PaintObject
2
PaintCanvas(int, int) : void
addPaintObject(PaintObject) : void
paintComponent(Graphics) : void
3
bounds g get history hovering
object objects paint rect
temporary
4

Figure 6. A type box with a header containing the name and package of the type (1), fields (2), methods (3) and a tag cloud (4). UI controls like the close button in the top-right corner or the resize handle in the lower-right corner are only displayed, when the user selects an element.

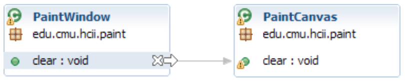

Figure 7. The preview of a call relation between methods. Clicking the semi-transparent arrow makes it persistent.

This interface is almost as powerful as the features introduced in the paper prototype, but offers many advantages. We can still perform searches (using the highlighting) and group types (using thematic relations). Since the vocabulary is visible, searches do not run the risk of returning no results due to ambiguous keywords (e.g., "listener"/"observer"). The overlay additionally helps the user to quickly identify key terms and build a spatial model of the code.

We created a demo of this concept with the prototyping system *SketchFlow* [11]. The demo features animations and an interactive mockup of the tag overlay. This prototype was evaluated by discussing it with members from our group and the *Computer Science Department III* of the *University of Bonn*. The interactions offered by the tag overlay and thematic relations were well received. This feedback led to our decision to implement a working system for further study.

Eclipse Plug-In

We implemented CG as editor plug-in for *Eclipse* [3]. We use the Eclipse API (*JDT*) for general *Java* queries, such as inheritance and call relations, and *Cultivate* [12] to obtain tag metrics. CG diagrams are integrated as a new document type with Eclipse. They can be placed everywhere in the project structure a source code file can, and are compatible with versioning systems.

The user generates diagrams using drag-and-drop. Project files and all kinds of code artifacts can be dragged to the editor from any Eclipse view and are visualized in a way similar to class diagrams. Types are represented as boxes (see **Figure 6**), where fields and methods can be added to a list of members. At the bottom there is a tag cloud, displaying the ten most frequent terms. The tag overlay and thematic relations are implemented as described in the previous section.

We display relations as context sensitive previews when diagram elements are selected (see **Figure 7**). The user decides on a case-by-case basis, if a relation should be included in the diagram. The informed decisions of the human user avoid uncontrolled growth of the SV and distracting clutter.

Evaluation

We evaluated CG by means of a user study with 16 computer science students and postgraduates from the *RWTH Aachen University* and the *University of Bonn*. We prepared four tasks, which asked the participants to look at a specific feature of the *Paint* source code from [4] and draw a diagram to explain it. Two diagrams had to be created using CG and pen & paper each. Afterwards, the testers were given a questionnaire to evaluate CG and rate the usefulness of individual features on a five-point Likert scale.

Code Gestalt scored a mean of 79.53 on the *System Usability Scale* [2] (standard deviation 8.37, median 77.5). The participants agreed with the statement, that CG is a practical alternative to pen and paper (median 4 on a (0..4) scale). Similarly, the thematic relation (median 4) was rated very useful. The highlighting features for types (median 3) and tags (median 3) were rated useful, as was the tag overlay (median 3).

The comments made by the participants during the study were compiled to a list of feature changes and improvements. The most desirable features are more customization and a number of new relation types (dependency, override, etc.). The users would also like to get previews for elements not yet included in the diagram to make the tool a substitute for Eclipse's call and type hierarchy views and more useful for exploratory tasks. Another feature in high demand is the selection of multiple elements in the tag overlay, using the cross-product of the respective weights for highlighting. This indicates that users expect multiple tags to capture a given concept better than one.

Summary

We developed Code Gestalt with the information obtained from an online survey and the evaluation of three prototypes. Through this process we introduce the tag overlay and thematic relations to allow users to find related code artifacts and take advantage of the human intelligence that went into the naming of identifiers. A user study supports these new concepts.

Future Work

We are in the process of evaluating the results from a second survey among the 16 study participants, who were asked to rate clarity, comprehensibility, and other aspects of other testers' CG diagrams and sketches to find out how CG performs as means of communication. Areas for improvement are the mathematical model behind the tag overlay, which is currently putting too much emphasis on trivial terms like "get", and the scalability of Code Gestalt. An interesting long-term perspective is the integration of hand-drawn sketches with the editor as in CodeGraffiti [7].

Acknowledgements

We thank Leonhard Lichtschlag, Jan Borchers, Günter Kniesel, Daniel Speicher, Jan Nonnen, and Johanna Nellen for many fruitful discussions and their support.

References

[1] Bassil, S. and Keller, R.K. Software Visualization Tools: Survey and Analysis. In *Proc. IWPC 2001*, IEEE Society Press (2001), 7-17.

[2] Brooke, J. SUS – A Quick and Dirty Usability Scale. *Usability Evaluation in Industry*. Taylor & Francis (1996).

[3] Eclipse, http://www.eclipse.org

[4] Ko, A. J., Myers, B.A., Coblenz, M.J. and Aung, H.H. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering 12*, 32 (2006), 971-987

[5] Kuhn, A., Loretan, P. and Nierstrasz, O. Consistent Layout for Thematic Software Maps. In *Proc. WCRE 2008*, IEEE Society Press (2006), 209-218.

[6] Kurtz, C. Code Gestalt: From Class Diagrams to Code Landscapes. *Diploma thesis*. To appear in 2011.

[7] Lichtschlag, L. and Borchers, J. CodeGraffiti: Communication by Sketching for Pair Programming. In *Ext. Abstr. UIST 2010*.

[8] Park, Y. and Jensen, C. Beyond Pretty Pictures: Examining the Benefits of Code Visualization for Open Source Newcomers. In *Proc. VISSOFT 2009*, 3-10.

[9] Sensalire, M. and Ogao, P. Visualizing Object Oriented Software: Towards a Point of Reference for Developing Tools for Industry. In *Proc. VISSOFT 2007*, 26-29.

[10] Sinha, V., Karger, D. and Miller, R. Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases. In *Proc. VL/HCC 2006*, 187-194.

[11] SketchFlow, http://www.microsoft.com/expression

[12] Speicher, D. and Nonnen, J. Consistent Consideration of Naming Consistency. In *Proc. WSR 2010*. 51-52.

[13] Umbrello. http://uml.sourceforge.net/