Supporting task-oriented navigation in IDEs with configurable HeatMaps*

David Röthlisberger
Software Composition Group
University of Bern, Switzerland

Oscar Nierstrasz
Software Composition Group
University of Bern, Switzerland

Stéphane Ducasse
INRIA-Lille Nord Europe
France

Damien Pollet
University of Lille 1
France

Romain Robbes
University of Lugano
Switzerland

Abstract

Mainstream IDEs generally rely on the static structure of a software project to support browsing and navigation. We propose HeatMaps, a simple but highly configurable technique to enrich the way an IDE displays the static structure of a software system with additional kinds of information. A HeatMap highlights software artifacts according to various metric values, such as bright red or pale blue, to indicate their potential degree of interest. We present a prototype system that implements HeatMaps, and we describe an initial study that assesses the degree to which different HeatMaps effectively guide developers in navigating software.

Keywords: software analysis, static analysis, development environments, program comprehension

1 Introduction

Gaining an understanding of large object-oriented systems by navigating the source code in a development environment (IDE) is an inherently difficult and time-consuming task. Object-oriented language characteristics such as inheritance and polymorphism make it difficult to understand how an application is implemented purely by navigating and browsing source code [1, 2, 10]. Often conceptually related code is scattered over many different source artifacts, *e.g.*, classes and methods.

IDEs offer little support to navigate efficiently the source space aside from the static system structure. Information about previous navigation, about the system's dynamics or its evolution, is not exploited. Previous research efforts such as NavTracks [9] and Mylyn [5] show that this additional information provides useful insights to a developer exploring a system or relocating previously browsed entities.

*In: *17th International Conference on Program Comprehension*, 2009

The history of navigation and modification of source artifacts can be exploited to provide hints to developers where they may want to navigate to or what to modify in order to perform a development task [9]. One could track:

- Recently browsed artifacts.
- Recently modified artifacts.
- Frequency of browsing.
- Frequency of modification.

Change logs contain a great deal of information that can help the developer to understand how the system has evolved [4, 6, 7, 11], for instance information about the number of different authors or versions.

Given the potential value of these very different kinds of information to help developers quickly navigate to software artifacts relevant to particular task, the challenge is to present this information in the IDE in such a way that does not further overload an already complex and busy user interface.

The Seesoft software visualization system [3] eases software analysis by mapping each line of code into a colored row. The color indicates a statistic of interest, red lines are for instance most recently changed lines and blue lines least recently changed. The main difference to our approach is that Seesoft works on single lines instead of entities and its visualizations are separated from the IDE in a dedicated tool. Thus Seesoft is not able to reason about navigation activities in the IDE, instead it uses version control systems as a source to determine recency of modification.

NavTracks [9] exploits the navigation history to recommend files related to the file the developer is currently looking at. A severe limitation of this approach is that it only takes into account one single data source, namely the recency of browsing in the navigation history, to assess the relatedness of artifacts. Other sources or even combinations of different sources, such as combining frequency and recency of navigation of entities, could lead to much better

results. Furthermore, a recommendation list helps little to obtain an overview over the whole system.

Mylyn [5] computes a degree-of-interest value for each source artifact based on the historical selection or modification of the artifact. The background color of the artifacts highlights their relative degree-of-interest in the context of the current task — interesting entities are assigned a "hot" color. We apply a similar approach to highlighting important artifacts, but the importance is assessed differently. While the degree-of-interest model is fixed in Mylyn[1], the developer can choose between different models in our approach and can even combine various models to obtain better results depending on the exact nature of the task. In our approach we propose to also take into account more complex information than just navigation and modification, including evolutionary information, such as how many different developers worked on the artifact in the history.

We propose a simple and uniform mechanism, called *HeatMaps*, to represent complex information in an easily understandable way in any IDE. A HeatMap maps all source artifacts presented in the IDE to colors ranging from red ("hot") to blue ("cold"). Hot entities contribute heavily to a given property while cold ones contribute little or nothing. HeatMaps represent a simple and uniform mechanism as we can apply them to very different properties of software, such as how recently a source artifact has been navigated or modified or how many versions or authors an entity has. Different HeatMaps may be more suitable than others for a given task-at-hand. A HeatMap can also be defined as a combination of existing HeatMaps, to simultaneously display different kinds of information.

In Section 2 we present the *HeatMaps* mechanism in detail. We assess the efficiency and accuracy of various HeatMaps for several case studies using a data set spanning 20 months of IDE navigation in Section 3. Section 4 concludes the paper with some remarks on future work.

2 HeatMaps

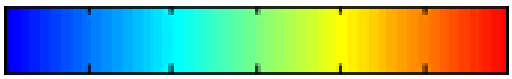

Figure 1. A color gradient from light blue to light red representing heat.

A HeatMap[2] employs the metaphor of heat to color artifacts: colors range from blue (cold) to red (hot) as Figure 1 illustrates. The "hotter" an artifact is colored, the more relevant it is meant to be for the task-at-hand. A HeatMap thus guides the developer and provides additional information about the relative importance of different source artifacts. In a large unknown system consisting of thousands of classes and methods, the *hot* artifacts are readily visible and can serve as a starting point to explore the system further. Figure 2 illustrates two examples where source artifacts are highlighted (i) based on the number of versions and (ii) how recently they have been browsed.

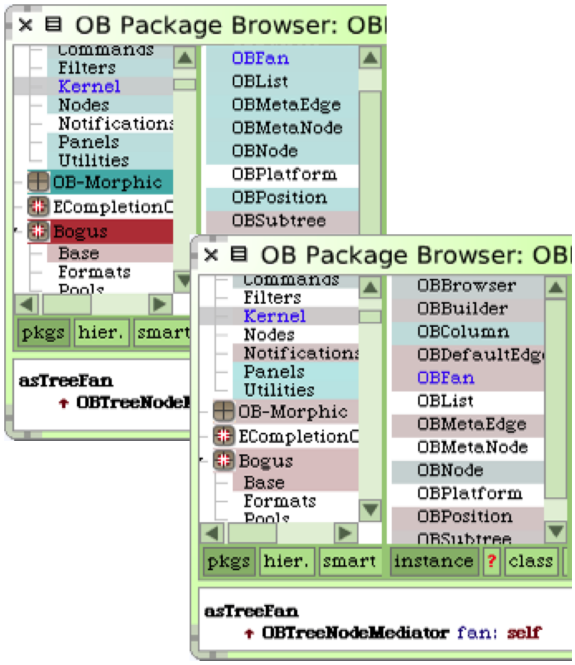

Figure 2. Two HeatMaps highlighting number of versions of source artifacts, top left, and recently browsed artifacts, bottom right.

HeatMaps can be seamlessly integrated in all traditional tools of an IDE. With the help of a dedicated interface, developers choose the kind of information that the HeatMap displays, and they can also configure how different HeatMaps are combined. The HeatMap for the chosen information then appears in all views and tools in the IDE, for example, in the package browser hierarchically presenting all system entities, as well as in the hierarchy browser focusing on the class hierarchy of a selected class. Source artifacts that appear in the data history for the selected HeatMap, such as artifacts that have been browsed or modified while correcting a defect, are assigned a background color representing their heat; artifacts not in the history are still displayed but not colored. Our prototype runs in Squeak Smalltalk[3] but could easily be ported to other IDEs such as Eclipse, as the technique does not depend on any Smalltalk-specific idiom.

[1] `http://www.eclipse.org/mylyn/`

[2] NB: "*HeatMaps*" (in italics) refers to the prototype tool, while "HeatMap" (unemphasized) refers to an individual map.

[3] `http://squeak.org/`

Typically, the navigation history, indicating how frequently entities have been browsed in the past, is a good guide to the importance of source artifacts. For a specific maintenance task other, more task-related information might lead to a better assessment of the relative importance of different artifacts.

Time- and metrics-based HeatMaps. Heat is computed by two different means, namely in *time-based* HeatMaps or *metrics-based* manner. To highlight recently browsed or modified entities, we use a time-based approach. The interest in an entity usually steadily decreases after it has been navigated or edited, that is, the entity gradually "cools down" as time passes by. A threshold determines after which time an entity should not taken into account anymore in a HeatMap. To denote for instance frequency of browsing or modification of an artifact, we use metrics-based HeatMaps. The higher the metric value the more important the artifact becomes; this importance is linearly mapped to heat colors. Again there is a threshold to filter out very low values of importance.

Combined HeatMaps. We assume that combining different kinds of information leads to a more accurate estimation for the source artifacts' importance than just exploiting one kind of information. Combining for example recently with frequently browsed HeatMaps is supposed to better assess the developer's interest in an artifact. We offer two different means to combine several HeatMaps: (i) weighted linear combination of the color values of different HeatMaps and (ii) exponential decay when combining one time-based with one metrics-based HeatMap.

How to gather the information for the HeatMaps. For many time-based HeatMaps we instrument the IDE itself to gather information about the navigation, modification, or deletion of source entities. Most metrics-based HeatMaps initially obtain their information by executing a batch process that analyzes all system artifacts to extract information such as number of versions or authors of specific artifacts.

Storing, caching, updating, and exchanging the information. We store the data used by HeatMaps in a simple file format. With some HeatMaps the underlying data sets quickly grow in size, so we cache the results of color computations. Usually HeatMaps are based not on an imported data set but on the data generated by the current developer in the current development session; in such cases we update the caches whenever an event occurs that is relevant to the currently selected HeatMap. The HeatMap data is easily exchangeable (*e.g.*, to append it to a bug report) as it is stored in files.

3 Validation

HeatMaps are intended to help developers to more quickly navigate to software artifacts relevant to the task-at-hand. To be successful, HeatMaps have to be *accurate*, that is, they should assess entities' importance properly, actually highlighting what is relevant for developers. We performed initial experiments to validate this requirements by testing HeatMaps against an available navigation and modification history spanning nearly two years to verify whether the various HeatMaps would haven given accurate hints to the developer.

Procedure. In a nutshell, the benchmarking procedure we implemented replays a recorded sequence of interactions, and measures the color of each element that was interacted with (in sequence) according to the HeatMap. The warmer the element is, the more accurate the map is. The sequence of interactions we replay consists of nearly 90'000 navigation and modification events recorded in an IDE while developing and maintaining a medium-sized system (consisting of 7000 methods in 700 classes) used to analyze software evolution over the course of 20 months. Benchmarks have the advantage of being easily replicable, ease the comparison of results, and can be used to test a restricted functionality, such as the effect of the weight used in the combination of different HeatMaps. The same approach has been used by other researchers to evaluate similar works such as code completion engines [8].

We implemented two variants of the benchmark, corresponding to two distinct use cases for *HeatMaps*:

1. In the *Monitoring Use Case* the developer uses HeatMaps in her daily work. Information used in a HeatMap is continuously gathered and displayed in the IDE, so when she navigates to a new artifact, the recently browsed HeatMap immediately takes this event into account.

2. In the *Historical Use Case* the developer does not record events about her own development but imports a recorded history of another development session, for example, a session recorded by another developer while implementing a feature. This historical data is assumed to be read-only, that is, newly created events are not added to the *HeatMaps* database.

Evaluation. To simulate this scenario we create an initial database with the first 500 records in the history, test for all following elements the color value they would be assigned in a particular HeatMap, and add the tested element itself to the *HeatMaps* database. The second use case is similarly simulated; here we vary the records added from the history to the *HeatMaps* database starting at the beginning of the history with a database size of 500. We then test the 100 elements following next in the history. Afterwards we create a new database with the next 500 elements after the 100 tested elements, test the 100 subsequent elements, and so on.

HeatMap	Accuracy
Recently browsed	74.48%
Frequently browsed	21.08%
Recently modified	34.52%
Frequently modified	4.01%
Artifacts' age	43.12%
Number of versions	$< 1\%$
Recently and frequently browsed *combined*	73.24%
Recently and frequently modified *combined*	39.17%
Recently browsed, recently modified *combined*	74.48%
Recently browsed and age *combined*	48.56%
"Best of everything"	75.91%

Table 1. Accuracy rates of different HeatMaps in the Monitoring Use Case

HeatMap	Accuracy
Recently browsed	68.27%
Frequently browsed	18.14%
Recently modified	39.02%
Frequently modified	3.62%
Artifacts' age	21.93%
Number of versions	$< 1\%$
Recently and frequently browsed *combined*	63.81%
Recently and frequently modified *combined*	39.02%
Recently browsed, recently modified *combined*	65.48%
Recently browsed and age *combined*	37.41%
"Best of everything"	70.36%

Table 2. Accuracy rates of different HeatMaps in the Historical Use Case

Testing a single artifact means computing its color value for the currently active HeatMap, then computing the distance to red as a percentage value, so "red" is a 100% fit, "blue" and not colored a 0% fit, and values in-between are interpolated. This procedure assumes that if the developer in the history selected an artifact and a HeatMap colored it red, then the HeatMap would have successfully guided the developer to the right artifact. The percentage values are aggregated for all tested elements to form an average result for the whole HeatMap using the given history.

Evaluated HeatMaps. In this experiment we test six different HeatMaps: *recently browsed*, *frequently browsed* (how often the artifact has been visited), *recently modified* (created, update, moved, renamed, or deleted), *frequently modified*, *age of artifact*, and *number of versions* (how often the artifact has been committed). Furthermore, we combine different HeatMaps to test whether combined information yields better results. We combined these maps using the weighted linear combination approach and weighted the second map with a factor of 2. As stated in Section 2 we can give different weights to the individual HeatMaps when combining them; in this validation each HeatMap is assigned the same weight in the combinations we tested. Finally, we did a *best of everything* experiment, that is, we computed for each tested artifact the maximum accuracy achieved under all tested HeatMaps. This final experiment thus leads to the maximum accuracy rate we possibly obtain with our approach and this data set. Table 1 shows the various accuracy rates for different HeatMaps we tested using the recorded developer activities.

Discussion of the results. From these results we conclude that HeatMaps perform similarly well for both use cases, that is, when continuously used in a development session, or when imported from a recorded history and used without taking into account events generated thereafter. The *recently browsed* HeatMap is the best performing single metric, which comes as no surprise since the past navigation actions are most likely to be a good basis to predict future navigation actions.

Modification actions lead to significantly less accurate results compared to navigation actions, as do frequency-based HeatMaps compared to recency-based HeatMaps. This is intriguing as other researchers reported higher accuracy rates for models based on modification activities [5, 9]. We explain our contradicting results by the fact that the used data set contains much fewer modification than navigation activities (84000 navigation events compared to 4000 modification events); thus many browsed entities have never been modified, which means that those entities are not colored by modification-based maps. We performed another experiment which tests modification-based maps only with those entities that indeed have been modified. In this experiment we obtain accuracies of 67.49% for recently modified and 31.08% for frequently modified. The really low accuracy for the number of versions map is explained by the fact that just a very small percentage of methods have more than one version. For systems with more evolutionary information available we expect much better results for maps based on such data.

Combining different HeatMaps does not in general increase the accuracy, although in some cases the combination of recently with frequently browsed HeatMaps does. Studying the "best of everything" test reveals that in three quarters of all cases the recently browsed HeatMap gives best results, but for one quarter of all elements the combination of recently and frequently browsed yields better results.

Task-dependent HeatMaps. The data set with which we performed this validation also contains information about the nature of the task that has been performed at the moment in which the navigation data has been recorded. In another experiment, we use this information to compare the performance of different HeatMaps for different specific tasks, to reveal whether some HeatMaps are better suited for one kind of development task than for another. We extracted four types of major development tasks from the data set: *defect correction*, *new feature implementation*, *refactoring*, and *navigation tasks* (tasks which do not change the system, probably performed purely to gain understand-

ing). This extraction of these four development task from the data set was performed manually. The data set was already separated in different development session. We analyzed these sessions and considered as defect correction sessions that only modified a small number of code entities (a few methods for instance), a feature implementation is a session which adds several novel entities to the system, and a pure navigation session does not modify the system. Most difficult to identify were refactoring tasks; as a refactoring we considered sessions that either contained refactoring actions provided by the IDE or that changed several entities in tandem without adding new logic. We left out sessions that we could not clearly associate to one of these tasks.

In Table 3 we report on how often a particular HeatMap most accurately directed the developer to the desired entities. For these four types of tasks, the recently browsed map, for defect correction and feature implementation combined with the recently modified map, performs best. We attribute this to the fact that in particular bug correcting activities often occur after a system has been frequently modified, thus the frequently modified combined with the recently browsed map gives best results. Refactoring and in particular navigation tasks often occur after navigation activities in which developers have spotted issues or interesting code segments to be investigated further. Hence visualizing previous navigation efforts helps developers to find the entities to refactor or analyze in more detail. The results in Table 3 serve as a guideline: when working on a task in one of these four areas, developers obtain best results when using the suggested HeatMap. We make use of this knowledge in *HeatMaps* to suggest well-performing HeatMaps to the developer based on the task-at-hand.

HeatMap	Defect	Feature	Refactor.	Navig.
Recently browsed	49.48%	50.90%	**64.27%**	**75.19%**
Frequently browsed	19.07%	20.28%	22.99%	24.82%
Recently modified	45.20%	31.73%	38.03%	28.39%
Frequently mod.	32.98%	9.64%	17.62%	11.88%
Rec. brow. & rec. mod.	**54.31%**	**51.14%**	63.00%	72.04%
Freq. brow. & freq. mod.	32.78%	44.01%	29.22%	61.76%

Table 3. Performance of different HeatMaps in specific tasks

4 Future Work

We plan to further explore three main directions: (i) which kinds of information provide the most accurate support for which kinds of tasks, (ii) which combinations of HeatMaps perform better than individual HeatMaps for a given class of tasks, and (iii) more detailed case studies to validate the performance of HeatMaps for different kinds of applications and tasks. Future validation will also include formal user-based experiments to evaluate how developers benefit of *HeatMaps* in their daily work.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010) and the INRIA support for the REMOOSE Associated team. We thank Orla Greevy, Lukas Renggli and the anonymous reviewers for their feedback.

References

[1] S. Demeyer, S. Ducasse, K. Mens, A. Trifu, and R. Vasa. Report of the ECOOP'03 workshop on object-oriented reengineering, 2003.

[2] A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[3] S. G. Eick, J. L. Steffen, and S. Eric E., Jr. SeeSoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, Nov. 1992. Depth.

[4] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 40–49, Los Alamitos CA, Sept. 2004. IEEE Computer Society.

[5] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, New York, NY, USA, 2005. ACM Press.

[6] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution)*, pages 37–42, 2001.

[7] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization)*, pages 67–75, St. Louis, Missouri, USA, May 2005.

[8] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of ASE 2008 (23rd International Conference on Automated Software Engineering*, pages 317–326, 2008.

[9] J. Singer, R. Elves, and M.-A. Storey. NavTracks: Supporting navigation in software maintenance. In *International Conference on Software Maintenance (ICSM'05)*, pages 325–335, Washington, DC, USA, sep 2005. IEEE Computer Society.

[10] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.

[11] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings 1st International Workshop on Mining Software Repositories (MSR 2004)*, pages 2–6, Los Alamitos CA, 2004. IEEE Computer Society Press.