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. Previous research has shown that other forms of information, such as evolution of the software artifacts, historical information capturing how previous developers have navigated the code, or runtime behavior of the application, can be much more useful than the static structure of the software to direct developers to parts of the code relevant to a task-at-hand. These different kinds of information, however, are not uniformly accessible from the IDE, so the developer may have to struggle with heterogeneous tools and visualizations to find the relevant artifacts. 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. HeatMaps can be dynamically configured to reflect different kinds of information relevant to a given task-at-hand. 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, dynamic analysis, static analysis, development environments, visualizations, program comprehension

1 Introduction

Conventional IDEs enable the exploration of a software system principally by providing views and mechanisms based on the static structure of the source code. Object-oriented language characteristics such as inheritance and polymorphism can lead to conceptually related code being scattered over many different source artifacts [2, 19]. This can lead to an unfocused, undirected navigation of the source space, resulting in the same entities being browsed several times in the same working session. Empirical experiments have shown that during a one day coding session, developers browsed 95% of all visited methods more than once [9].

IDEs offer little support to efficiently navigate 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 [16] and Mylar [6] show that this additional information provides useful insights to a developer exploring a system or relocating previously browsed entities. If a developer has for instance access to historical information about her own navigation or that of other developers, she will be able to locate previously navigated entities more quickly [16, 6].

Although gathering additional information about navigation, history, or even the dynamics of a system may not be particularly challenging in itself, representing and displaying the vast amount of information gathered in the constrained IDE space is inherently difficult. In this paper we tackle the following research question: "Is there a unifying mechanism to represent the complex information that developers face in the context of a constrained IDE space while working on a development task?" This question is then further divided in the following issues:

- How can a uniform mechanism represent various kinds of complex information in an IDE?
- What different kinds of information are of use to a developer while performing various tasks on a large software system?

In this paper we introduce 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, how many versions or authors an entity has, or even how much space is allocated to a method invocation. Different HeatMaps may be more suitable than others for a given task-at-hand, so the developer can configure the IDE dynamically to apply a selected HeatMap. A HeatMap can also be defined as a combination of existing HeatMaps, to simultaneously display different kinds of information.

In Section 2 we motivate the need for a uniform approach to represent various kinds of information in the IDE. In Section 3 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 4. Section 5 discusses the strengths and weaknesses of this approach while Section 6 concludes the paper with some remarks on future work.

2 Information overflow in IDEs

Development environments present vast amounts of information to developers, usually reflecting the static structure of the code. For example, in the Eclipse IDE[1] there are more than ten different types of projects, there is a huge icon set with more than a thousand different icons, and a large number of source code entities are distinguished, including packages, classes, interfaces, class hierarchies, methods, attributes, inner classes, and aspects. This vast range of information can easily overwhelm developers, making it difficult for them to focus on the particular entities relevant to a task, such as classes working together at runtime, or methods changing in tandem in every new version.

We therefore argue that there is a need for a configurable mechanism to ease navigation by highlighting software artifacts of special relevance to a given task.

Next we present a typical use case that could clearly benefit from such a mechanism.

2.1 Motivating Use Case

As developers we face the task of correcting a defect in a large, unfamiliar web application written in an object-oriented language. This defect occurs in a feature that has been previously implemented by another developer who has left the team. Due to our lack of knowledge about this system, we cannot easily identify the entities responsible for the broken feature. Our IDE has recorded the development actions performed by the developer while building this particular feature, so we can exploit this information. However, it is not clear how this historical information can be presented in the IDE to help us with this task. In addition to the historical navigation data, we also have access to the change logs, containing data about previous versions, commits, and authors. It is known that this kind of information can also be useful to direct the developers to software artifacts likely to contain defects [3, 4, 17, 21]. Besides correcting the mentioned defect, we are required to also boost the performance of this feature in general. We hence want to see directly in the IDE hints about the execution behavior in terms of execution time.

In our case we could benefit from the availability of three very different kinds of information directly in the IDE: (i) information about previous navigation and possibly modifications performed by developers in the past, (ii) information about the system's evolution, and (iii) information about the runtime behavior of the subject system.

2.2 Development Driven Information

There is a large range of information that is orthogonal to the static structure of a software system, but which may be of use for various development tasks. We consider several program comprehension tasks that developers are typically faced with, and we list, in each case, several of the kinds of non-structural information that can be helpful for the task-at-hand.

- *Exploiting navigation and modification activities.* 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 [16].
  - Recently browsed.
  - Recently modified.
  - Frequency of browsing.
  - Frequency of modification.
  - Modified by me, *i.e.*, the degree to which an artifact has been modified by the current author, measured by number of methods or versions contributed.
  - Extent of modification, *i.e.*, how many lines or methods changed in a method or class.
  - Inclusion in search results, *i.e.*, how often an entity appears in the results of submitted searches.
- *Exploiting evolution history.* Change logs contain a great deal of information that can help the developer to understand how the system has evolved [3, 7, 10, 21].

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

  - Number of different authors or versions.
  - Age.

- *Exploiting execution.* Dynamic information helps developers to reason about issues occurring at runtime, such as performance bottlenecks [1, 5, 18, 14].
  - Memory consumption.
  - Execution time.

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. With this question in mind, we now briefly review related work in the field of representing information orthogonal to the system's static structure in order to highlight entities potentially relevant to a given developer task.

2.3 Related Work

In the context of development environments, in particular FEAT [13], NavTracks [16] and Mylar [6] aim at a similar goal as our proposal. However, there are several limitations to these approaches and differences to our proposal.

FEAT [13] applies a concern graph to visualize scattered but conceptually related code elements together in order to navigate concerns. However, in the original FEAT tool, developers had to manually create this concern graph. Robillard *et al.* [12] enhanced FEAT to automatically infer concerns, however, users still have to accept or decline the inferred concerns; our approach does not require any explicit user action.

NavTracks [16] exploits the navigation history to recommend files related to the file the developer is currently looking at. This approach works at the granularity of files, hence does not take into account specific methods or classes. 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 modification and navigation of entities, could lead to much better results. Furthermore, a recommendation list helps little to obtain an overview over the whole system; the developer just sees a list of artifacts possibly related to a specific artifact, but does not see all interesting entities in a "big picture" view. These recommendations are always relative to a selected artifact, that is, dependent on what the developer has currently selected, while our goal is to generally guide the developer through the source space, independent of a concrete selection. The exchange of data sources recorded by different developers is also not supported with NavTracks as its model is built on the client side in this specific environment.

Mylar [6] 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 Mylar, the developer can choose between different models in our approach and can even combine various models with each other to obtain better results depending on the exact nature of the task. In Mylar the information used to compute the interest value is relatively simple: selecting and editing an artifact increases the interest; if no further event occurs the interest decreases over time. In our approach we propose to also take into account more complex information, including runtime information, and evolutionary information, such as how many different developers worked on the artifact in the history. For many tasks, information about previous navigation or modification is not sufficient to accurately determine the degree-of-interest, while dynamic or evolutionary information is likely to give better results. For example, when addressing a software regression, taking into account evolutionary information about who changed what in the system gives good hints to developers about what they should navigate. For this reason we provide in our approach suggestions what information, including combinations of different kinds of information, gives best results for which tasks.

HeatMaps also visualize evolutionary information about software systems, that is, information obtained by mining software repositories. Other researchers also exploited software repositories to improve task-oriented navigation, in particular by studying how source artifacts have frequently changed together. Shirabad *et al.* [15] or Moonen [8] use information about artifacts with common change patterns to recommend developers to also change the related entities when working on an artifact. Our approach differs as it applies historical information as a measure for general importance of source artifacts with respect to the task-at-hand.

Xie *et al.* [20] also show a complete picture of evolutionary data extracted from CVS repositories, but these visualizations are outside of the IDE and thus of limited use while working with the static system structure.

3 HeatMaps

We now introduce *HeatMaps* and explain our approach in detail. In particular we explore how IDEs can use HeatMaps to display the different kinds of information seen in Section 2.2 with a uniform mechanism.

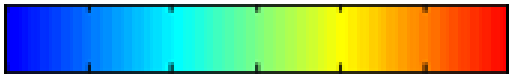

Figure 1. A color gradient from light blue to light red representing heat.

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 thousand 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.

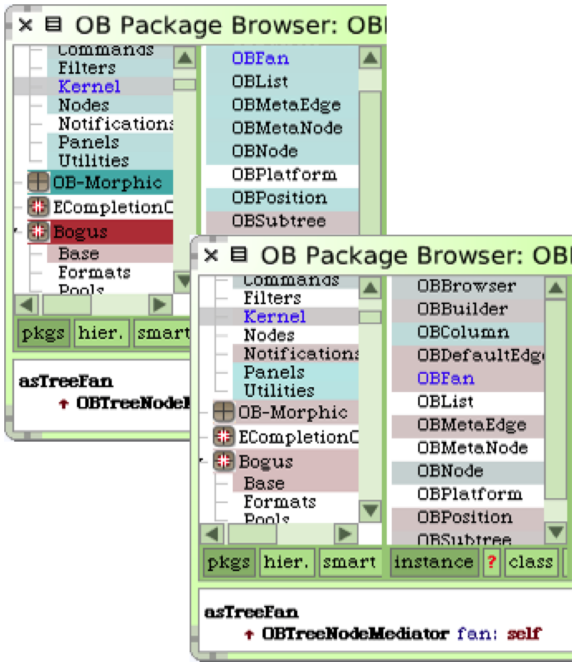

Figure 2. Two HeatMaps highlighting number of versions of source artifacts, top left, and recently browsed artifacts, bottom right.

HeatMaps can be seamlessly integrated in all traditional tools of the 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. HeatMaps do not replace or alter the display of the system's source code in any tool of the IDE, except by adding a background color to the display of source artifacts such as packages, classes, or methods. 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.

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. HeatMaps are freely configurable by developers. Depending on the exact nature of the task, the system's evolutionary information might give better results than, say, information about historical navigation.

As the different HeatMaps to visualize the heat of an entity are based on very different kinds of information, we briefly describe the way in which heat is computed for two classes of HeatMaps, namely *Time-based* HeatMaps and *Metrics-based* HeatMaps.

Time-based HeatMaps. HeatMaps highlighting recently browsed or modified entities are used to reason about the time at which the navigation or modification of entities occurred. The interest in an entity usually steadily decreases after it has been navigated or edited. In Figure 3 we can see how with a time-based HeatMap a cold entity is associated with an early time while a hot entity is close to the current time. We assume that the interest in an entity decreases steadily as time passes by, thus an entity's color constantly "cools down". We experimented with several mechanisms to cool down an entity (cf. Section 4) and got best results when gradually cooling the entity as time passes by. There is a lower bound of entities' time values to take into account, determined by the size of the available history and the time passed by between now and the recorded time for an artifact. This means that if artifacts have not been covered by a relevant event for a long time, they drop out and will not be colored in this particular HeatMap. When reusing old navigation data, as in the use case described in Section 2.1), HeatMaps take the highest time value in the recorded data set as the current time to color the most recent items red.

Metrics-based HeatMaps. Frequency of browsing or modification of an artifact, and the number of developers having altered it are two examples of metrics-based

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

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

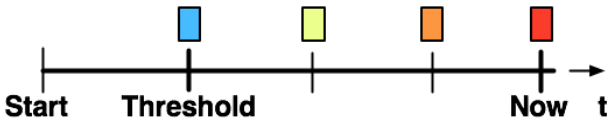

Figure 3. Time-based color gradient

HeatMaps. Such HeatMaps are used to reason about metrics associated with each artifact in the system. The higher the metric value the more important the artifact becomes. Metric values are linearly mapped to heat colors in metrics-based HeatMaps, as illustrated in Figure 4. To make sure that HeatMaps meaningfully highlight particularly important source artifacts, we introduce a threshold if the data set contains a wide range of different ordinal metric values. Hence we often associate *cold* not with the minimum value in the data set but with the threshold value (cf. Figure 4). We determine the threshold based on the system size, the size of the data set, and the distribution of the data.

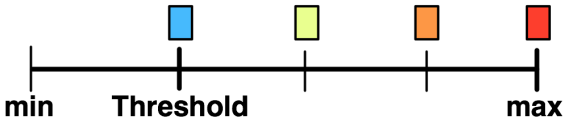

Figure 4. Metrics-based color gradient

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. Combining two HeatMaps linearly means that an entity once colored in blue and once in red is assigned an in-between color, if the two HeatMaps are equally weighted. It often makes sense to weight one HeatMap more than the other(s). For instance, if we combine recently browsed with recently modified, we weight the color value from the recently modified map with a weight of 2 (or even higher), as modification is rare and thus most likely increases the interest in an entity more than its navigation does. In the exponential decay combination we assume that the interest in an entity exponentially decreases over time. Obviously we are most interested in an artifact at the moment when we browse it. This event is additionally weighted with the number of times we previously browsed the same entity. From this point on, the interest in that artifact decays exponentially, similar to radioactive decay. Such a combination has the advantage that entities not having experienced any action for a long time are still colored if they once had been very important.

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. HeatMaps used to visualize behavioral information require the developer to instrument and exercise the application to gather execution time or memory usage data [14].

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. This is particularly important for aggregate entities such as packages or classes as they aggregate the color value from their child elements (*e.g.*, single methods), rendering their color computation more time-consuming. 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. These caching mechanisms make sure that HeatMaps are displayed efficiently even when their underlying data grows with the ongoing development session. The HeatMap data is easily exchangeable (*e.g.*, to append it to a bug report) as it is stored in files. Thus we can easily import the navigation data generated by the developer in our use case (Section 2.1) to correct this defect.

4 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 fulfill at least two requirements: They need to be (i) *efficient*, so updating and displaying should not slow down the IDE, and (ii) *accurate*, that is, they should assess entities' importance properly, actually highlighting what is relevant for developers. We performed initial experiments to validate these two requirements by (i) benchmarking the efficiency of updating and rendering HeatMaps, and (ii) 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. Finally, we report on an informal user experiment we conducted with developers using *HeatMaps*.

4.1 Efficiency of HeatMaps

We tested the performance of the *recently browsed* HeatMap by observing the time it takes to add new elements to the database, including updating all dependent HeatMaps, refreshing all involved caches and updating the visualization, and we measured the time to actually color all artifacts for a particular HeatMap in the whole system. The system we used is the Squeak Smalltalk system itself, consisting of 3180 classes and 57400 methods. We measured the display of HeatMaps in the system browser that shows all system entities. Updating the *HeatMaps* database upon navigation activities causes a non-measurable slowdown in the range of some milliseconds. Coloring the whole system with a new map affecting more than half of all entities took less than a second. Thus we consider *HeatMaps* as a an efficient means to visualize information in IDEs.

4.2 Accuracy of HeatMaps

In this section we evalute the accuracy of various HeatMaps and their combinations using a benchmark.

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 [11].

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 the first use case 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.

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. We compute accuracy for the Monitoring Use Case as follows:

$$Accuracy = \frac{\sum_{i=d}^{n} dist(CV(x_i), RED)}{n - d}$$

where d represents the size of the initial database, n is the final size of the database, x_i is the ith element, $CV(x)$ is the color value assigned to element x, and $dist(cv_i, cv_j)$ is the distance between two colors.

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 3 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 (Monitoring Use Case) and Table 2 (Historical Use Case) show the various accuracy rates for different HeatMaps we tested using the recorded developer activities.

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

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; thus our motivating use case (Section 2.1) should be easier to support by a HeatMap that gives us hints about what the developer browsed while originally developing the broken feature.

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 [6, 16]. 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 contains 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 nearly three quarters of all cases the recently browsed HeatMap gives the best value, but for one quarter of all elements the combination of recently and frequently browsed yields a better result.

To assess the fitness of this experiment we also constantly studied the ratio of entities colored by the evaluated HeatMap and all system entities. This ratio was varying between 5% and 38% throughout all experiments with an average at 17%, hence colored entities clearly stand out.

Threats to validity. There are several threats to validity in the experiment we performed. Firstly, the data set we used contains all navigations and modifications occurring in one single application, thus we cannot generalize our results to other systems as well (threat to external validity). However, we consider this data set as being fairly typically for other applications of similar (medium) size. The system experienced several extensions, changes, and refactorings. It also contained several defects that had to be addressed, thus the recorded development history covers all the typical tasks we want to support with *HeatMaps*.

Secondly, the observed navigation and modification patterns have not necessarily been effective or even optimal, for instance, if developers didn't navigate themselves directly to the right artifacts. Since HeatMaps should guide developers effectively to the entities they have to understand or modify in the context of a specific task, the HeatMaps should make close to optimal suggestions. The recorded data set most likely does not represent an optimal navigation in all cases, thus it is likely that HeatMaps performing well in the simulated study are not necessarily optimal for the task-at-hand (threat to external validity). However, as we know that the developers generating this data set have been involved in the system's development from the start and have thus been very familiar with it, we assume that their navigation patterns are generally very directed to what they were actually looking for. The results of the experiment would have been different if we had assumed that not the navigation but the actual modifications performed indicate an optimal pattern. In that case an optimal navigation directly opens the entities to modify in order, for instance, to correct a defect. Under this assumption, the recently and frequently modified maps give much better results, namely

72.25% and 47.81%, respectively. Neither assuming that the navigation nor the modification patterns are optimal in the available data set, is fully correct. We opted for the former assumption because navigation activities are, of course, much more frequent while working in an IDE [9] than modification activities. The reliability of test results is usually higher when based on larger data sets.

Thirdly, in this experiment we did not yet distinguish between different tasks. We are going to analyze the performance of *HeatMaps* with respect to the task-at-hand in the subsequent experiment. We performed this experiment under the assumption that the recorded data represents one large task during which developers navigated optimally (threat to construct validity). A separation by different development sessions, however, would make sure that a history of one session, for example, in which a bug was fixed, does not influence the suggestions for navigation in a completely different session dedicated, say, to refactoring. However, this implicit knowledge would have rather increased the accuracy, as using only the history of a similar session to generate the HeatMaps is very likely to give 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 understanding).

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 (cf. Section 5). We did not test how the HeatMaps visualizing dynamic information would have performed as there is no recorded runtime data about this system available. We expect such maps to outperform others for specific bug corrections.

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

4.3 User feedback

In addition to the benchmark validation we also gathered feedback from developers using HeatMaps in practice. Four developers used HeatMaps over a period of several hours up to a week while performing various kinds of tasks such as maintaining a known system. We also asked one developer to gain an initial understanding for a unfamiliar system we developed; we provided him with HeatMaps visualizing our navigation history in this system. The developers using HeatMaps generally appreciated their presence during their work. They considered this navigational aid to be useful; in particular they liked that fact that HeatMaps are easy to understand and that the maps apply to a wide range of different kinds of information. The colors we have chosen as background color for the source artifacts are considered to be non-intrusive (we opted to use a color gradient from light blue to light red to obtain soft colors). All participants stressed the importance of suggesting task-dependent HeatMaps; although the IDE should suggest, based on the developer's characterization of the task-at-hand, the best suited HeatMap, the engineers still want to be able to customize the automatic suggestion.

After using *HeatMaps* for a while, one developer considered the *frequency* and *recently browsed* HeatMaps to be most useful when he was interested in understanding the system in general; for addressing a specific maintenance task, he opted for HeatMaps focusing more on that task, such as HeatMaps showing evolutionary information about a specific part of the system or information involving frequency of modification, or the number of versions, not just navigation.

These early user comments offer a promising feedback about how useful *HeatMaps* can be in practice; performing a full-fledged user experiment we leave as future work.

5 Discussion

Next we discuss several important aspects of our proposal: (i) combining or aggregating different information from single HeatMaps, (ii) task-dependent or goal-oriented usage of HeatMaps, and (iii) studying its limitations.

Information aggregation. From the validation in Section 4.2 we learn that combining HeatMaps does not appear to have a significant positive effect on the accuracy of a HeatMap. The accuracy of combinations heavily depends on the HeatMaps used, on their weighting, and on the data set of recorded activities. As the experiment studying the task-dependency of HeatMaps reveals, combined HeatMaps can outperform single maps for specific goals or tasks, as it was the case for defect correction and implementation of new features (cf. Section 4.2), where a combination of the recently browsed with the recently modified map performed best, also better than the recently browsed map alone. Instead of combining HeatMaps we could also already take into account the different actions performed by the user when initially building a single HeatMap. Mylar [16] for instance creates a degree-of-interest model in which not only navigation but also each keystroke performed during the modification of an artifact directly increases the interest value.

As one of our primary goals with this approach is to freely combine, exchange, and distribute the underlying data for *HeatMaps*, as well as to have a uniform approach to display very different information efficiently in the IDE, we deliberately keep the information used in single HeatMaps as simple as possible, even though gathering this information is often complex or time-consuming, as is the case for HeatMaps presenting dynamic information. This enables the developer to select from a wide range of information that which best fits the specific task-at-hand; the IDE supports the developer hereby by offering suggestions for proven combinations.

Task-dependency, Goal-orientation. A navigational aid such as *HeatMaps* should ultimately guide the developer towards her goal, for example, the entity she actually needs to modify to correct a defect. In software maintenance the goals can be very diverse — gaining an understanding for the software is usually a prerequisite to attain any goal when maintaining software. *HeatMaps* contribute to program comprehension by highlighting entities according to their importance. As an artifact's importance depends highly on the programmer's task and on the concrete goal she is pursuing, *HeatMaps* can visualize a wide range of information and propose suggestions what configuration or combination of different HeatMaps are most useful for a specific task. Developers can further refine the suggestions given by the IDE.

For many tasks and goals it may be obvious what information is likely to be most useful for identifying the important entities. For example, to optimize performance, a HeatMap highlighting heavy computations is a natural choice. Not only the HeatMap itself but also the nature or age of the data it visualizes influences the accuracy. From our experience we know that when starting a new task, the data for the HeatMaps should either be freshly recorded from scratch or originate from a similar task, otherwise the assessment of the entities' relevance is not accurate enough to properly guide the developer. For this reason *HeatMaps* provide the means to easily store the used data for later reuse in another, similar task (as done for the running use case in Section 2.1). Particularly useful is saving the *HeatMaps* data used while correcting a bug together with the bug report, thus giving other developers in the team the opportunity to view the HeatMaps of the original developer who addressed this particular bug [16].

As mentioned in Section 4.2 we also provide best practice guidelines to suggest which HeatMaps are most useful for which kind of task. For developers correcting defects different entities may be important than for new team members trying to gain an initial understanding for a large software system. In the future, we want to perform empirical user studies with different HeatMaps with respect to how well they perform for various tasks. Of particular interest is the impact of HeatMaps on initial program understanding, such as when a new developer joins a team.

Limitations. *HeatMaps* are limited in their expressiveness, since they can, by definition, only display one ordered set of values at a time. We can circumvent this limitation to some degree by combining different HeatMaps. However, it is unclear whether we could display more than one HeatMap at the same time in the IDE by, say, coloring artifacts in several colors, or whether we could use discrete colors to visualize discrete values, such as authors, but still show the degree of realization for a variable, for example, how much an author contributed to an artifact, by displaying a gradient around each discrete color. Abusing *HeatMaps* to visualize too complex information is likely to erode their main advantage, that is, being easily understandable.

6 Conclusions and Future Work

In this paper we addressed the research question whether there is a uniform means to guide developers working on various development tasks through a large software space directly in the IDE. We proposed *HeatMaps*, a uniform approach to visualize various kinds of information orthogonal to the static system structure in the IDE. Evolutionary information, information about the historical navigation, modifications performed in the IDE, or dynamic information about a large software system, can direct developers to software entities important for a specific goal. As software develop-

ers and maintainers face very diverse tasks, *HeatMaps* offer a flexible and configurable means to visualize different kinds of information relevant to these tasks. In particular, we provide predefined configurations of HeatMaps that are, according to our evaluations, best suited to direct developers when solving problems such as navigating a system to gain an initial or deeper understanding, correcting defects, implementing new features, or refactoring a system.

In the future 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.

References

[1] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, pages 326–337, Oct. 1993.

[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] 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.

[4] A. Hassan and R. Holt. Predicting change propagation in software systems. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 284–293, Los Alamitos CA, Sept. 2004. IEEE Computer Society Press.

[5] D. Jerding, J. Stasko, and T. Ball. Visualizing message patterns in object-oriented program executions. Technical Report, Georgia Institute of Technology, 1996.

[6] 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.

[7] 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.

[8] L. Moonen. Exploring software systems. In *Proceedings International Conference on Software Maintenance (ICSM 2003)*, pages 276–280. IEEE Computer Society, 2003.

[9] C. Parnin and C. Görg. Building usage contexts during program comprehension. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 13–22, Los Alamitos CA, 2006. IEEE Computer Society.

[10] 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.

[11] 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.

[12] M. P. Robillard and G. C. Murphy. Automatically inferring concern code from program investigation activities. In *Proceedings of the 18th International Conference on Automated Software Engineering*, pages 225–234, Oct. 2003.

[13] M. P. Robillard and G. C. Murphy. Feat: A tool for locating, describing, and analyzing concerns in source code. In *Proceedings of 25th International Conference on Software Engineering*, pages 822–823, May 2003.

[14] D. Röthlisberger, O. Greevy, and O. Nierstrasz. Exploiting runtime information in the IDE. In *Proceedings of the 16th International Conference on Program Comprehension*, pages 63–72, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[15] J. S. Shirabad, T. C. Lethbridge, and S. Matwin. Mining the maintenance history of a legacy software system. In *International Conference on Software Maintenance (ICSM 2003)*, pages 95–104, 2003.

[16] 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.

[17] A. Tarvo. Mining software history to improve software maintenance quality: A case study. *IEEE Software*, 26(1):34–40, Jan. 2009.

[18] R. J. Walker, G. C. Murphy, J. Steinbok, and M. P. Robillard. Efficient mapping of software system traces to architectural views. In *CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 12. IBM Press, 2000.

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

[20] X. Xie, D. Poshyvanyk, and A. Marcus. Visualization of CVS repository information. In *WCRE'06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 231–242, Washington, DC, USA, 2006. IEEE Computer Society.

[21] 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.