Visualizing Software Dynamicities with Heat Maps

Omar Benomar Houari Sahraoui Pierre Poulin
Dept. I.R.O., Université de Montréal
{benomaro, sahraouh, poulin} @iro.umontreal.ca

Abstract— **Interactive software visualization offers a promising support for program comprehension, including program dynamicity. We present, the extension of an existing visualization tool with heat maps to explore the time and other dimensions of software. To this end, we first propose a framework to unify the two main software dynamicities, execution and evolution. Then, this unified framework is exploited to define a visualization environment based on heat maps. We illustrate our approach on two comprehension tasks: understanding the behavior of programmers during the evolution of an application and understanding class contributions in use cases. The case studies show that the heat-map metaphor contributes to answer, more easily, many of the questions important to program comprehension.**

I. Introduction

There is a consensus today that program comprehension is a major challenge in software maintenance [8]. Many studies show that comprehension consumes the largest part of resources dedicated to maintenance [16]. This fact encourages the maintenance research community to develop tools to support program comprehension. Among these tools, software visualization environments are increasingly popular.

Visualization of multi-dimensional data helps program comprehension by involving human analysts in data exploration without overwhelming them. Unlike automated tools, visualization allows free exploration without a predefined and hard-coded process. Several dimensions may be explored simultaneously, such as structure, quality, and bug tracking.

At the same time, much effort has been dedicated to consider the time dimension in program comprehension. For many tasks, it is essential to understand the dynamicity of a program, for example, from the point of view of execution or of evolution. Nevertheless, it is difficult to represent efficiently the time dimension in a visualization tool. It is even more difficult when the time representation is combined with the representation of other dimensions. Roughly speaking, three types of approaches may be used to represent the dynamicity of a program: (1) different snapshots, corresponding to different time steps, displayed side by side [25], (2) an animated sequence displaying the program's state changes [14], and (3) aggregation of the data into a single view [24].

The existing contributions consider execution and evolution dynamicities as two different problems, irrespective of the approach used to represent them. Consequently, tools proposed in one community are usually not reused in the other. Nevertheless, the two dynamicity problems present similarities in many aspects, which suggests that their unification is possible.

In this paper, we propose a first step towards the unification of execution and evolution software dynamicities. This unified framework is exploited to define a visualization environment based on heat maps. Heat maps are commonly applied on existing representations to display the intensity of a particular phenomenon with respect to the represented entities. We extend an existing visualization environment, *VERSO* [13], in which different dimensions are already represented. In our extension, heat maps are used either to visualize basic properties related to time or combinations of such properties. Our adaptation of the heat map metaphor is not straightforward. Indeed, heat maps are commonly used on concrete representations where the entities' positions are meaningful, such as in meteorology. In our context, software is intangible. It is intended to be understood by humans and computers, and has no concrete reality outside of these purposes.

We illustrate our approach on two comprehension tasks: understanding the behavior of programmers during the evolution of an application and understanding class contributions in use cases. These case studies show that a heat-map based metaphor contributes to answer, more easily, many of the questions important to program comprehension.

The rest of the paper is organized as follows. Section II introduces the necessary background and presents the related work. It gives a brief overview of heat-map visualization, and discusses other software dynamicity visualizations. Finally, it recalls the main features of *VERSO* environment, which is used as a basis to our heat-map technique. The unification of execution and evolution dynamicities is described in Section III. Section IV details our heat-map visualization technique and how it is integrated to *VERSO*. Section V presents two case studies used to evaluate our approach. Section VI summarizes the the paper, discusses encountered challenges and limitations of our approach, and gives future research extensions.

II. Background & Related work

A. Heat maps

Heat maps are 2D graphical elements that employ a heat metaphor to color artifacts and represent the intensity or importance of a particular phenomenon [19]. The latter is represented by a 2D array of data, where each 2D coordinate is associated with a value, bounded by a minimum and a maximum value. Colors are interpolated according to the distribution of these data values. The mapping between data values and colors provides insight about the phenomenon.

There has been much research about heat maps as a visualization technique. Of particular interest are the studies on of eye movements to the visualization [21], [5]. Heat maps have also been used as a visualization technique for

clustering and highlighting the most significant data in high-density information, such as user traffic in an Internet radio station [15]. Pryke et al. [17] show with heat maps the quality of solutions in a multi-objective optimization problem, where the results correspond to a population of solutions consisting of values of parameters and scores on multi-objectives.

Rothlisberger et al. [19] evaluate the use of heat maps within the views of an Integrated Development Environment (IDE). Heat maps are displayed directly on the existing views of the IDE. They assess the capacity of different heat maps to guide developers in exploring software in an IDE. Heat maps are also used to analyze geographical maps and identify trends in a particular data set. Hence, heat maps need to be applied on a 2D surface where each location, defined by a coordinate system, has a representative value. However, the underlying locations are usually natural and meaningful when applying the heat map, and the challenge in our approach is that the represented software entities do not have natural positions, their position being rather determined during the visualization.

B. Software dynamicity visualization

There are several visualization tools designed for understanding software executions or evolution. In the remainder of this section, we present some examples of each category.

Cornelissen et al. [7] are concerned with the dynamic analysis of a program, and propose two views to visualize execution traces: (1) a massive sequence view as an UML-based view, and (2) a circular bundle view that utilizes hierarchical edge bundles to represent dependencies occurring during an execution trace. Renieris and Reiss [18] represent data for temporal trace execution using two views: a spiral view shows the complete trace, and a linear view highlights parts of the execution trace. They also propose ways to coordinate these two views and to take advantage of their respective strengths. Trümper et al. [23] present a visualization method for the comparison of large execution traces. The method uses hierarchical edge bundles to match two traces to be compared and detects similarities as well as execution permutations. Bohnet et al. [4] propose a technique for visualizing pruned execution traces, that supports programmers in comprehension tasks. The technique generates a linear view of a pruned trace which shows call similarities. This view cohabits with other views on traces such as call graphs and sequences, etc.

EvoGraph by Fischer and Gall [9] explores program evolution and produces 2D visual representations of the evolution of structural dependencies extracted from the system's release history. They propose a graph view (stickiness view) to analyze co-change information and a longitudinal view to represent structural dependencies. Wettel et al. [25] give a 3D visualization (*CodeCity*) of an object-oriented program where software elements are represented as buildings and districts by using a city metaphor similar to the one of *VERSO* [13]. In this work, colors represent one time dimension, i.e., the age of methods, classes, and packages. Langelier et al. [14] are interested in the study of large-scale programs' evolution. The authors use an approach based on real-time navigation and animation to investigate programs composed of thousands of classes, over multiple versions. They exploit the intrinsic coherence between subsequent versions along with the human visual abilities, to understand quality aspects of software evolution. Voinea and Telea [24] augment existing visualization techniques by displaying more information simultaneously. They encode up to four attributes using colors and textures. *Gevol* by Collberg et al. [6] uses graph visualization to represent evolutionary aspects of a software, such as authors' contributions in the development process, changes in source code, or other structural changes. The nodes are color-coded depending on the recency of the changes. *Gevol* is intended to be used with other tools, such as code browsers, to facilitate program comprehension by the developers.

All the above-mentioned contributions consider *one* of the two time aspects, and the associated metaphors are not meant to visualize time dimensions concurrently with other properties in an integrated and flexible environment. Moreover, to the best of our knowledge, none of the proposed tools consider at the same time evolution and execution understanding.

C. VERSO

As mentioned earlier, heat maps are applied to a 2D surface. In this work, we associate heat maps to *VERSO* [13], [14], a visualization framework that represents a program as a 3D graphical scene. *VERSO* uses a base rectangular region and a treemap layout to represent the hierarchical structure of the program. The base rectangle is divided into different regions of different sizes to represent software packages. Each region size is proportional to the number of classes in its corresponding packages. The treemap layout algorithm further subdivides these regions to represent the packages' descendants, and so on. Finally, classes are represented as 3D boxes lying on these regions. In summary, *VERSO* gives a 3D visualization of a program by representing packages as regions and classes as 3D boxes within these regions. *VERSO* enables a developer to map software metrics to the graphical properties of 3D boxes representing classes. For instance, classes' coupling can be mapped to the boxes' color, classes' complexity to boxes' height, and classes' cohesion to boxes' rotation angle around the Y-axis. Other metrics could be mapped dynamically to the boxes' attributes such as the number of open bugs associated to a class or simply a developer color identifier. These kinds of information represented in *VERSO* describe static properties of a program during visualization. In contrast to the program static dimension, heat maps intend to represent program changing state over time. Software states can change in time in different manners. In our approach, we study program states during its execution and during its evolution.

III. Unifying time dimensions

Software dynamicity shows itself in two dimensions: execution and evolution, i.e, software changes over time while executing and also during its evolution. This differentiation between software time dimensions reflects in the software visualization communities. We can distinguish between two

visualization communities in the representation of software dynamics. First, execution visualization that is interested in the depiction of the execution states of software and understanding its dynamic behavior (see [7], [18]). Second, evolution visualization that is concerned with the representation of software changes from one version to another (see [14], [25]). These two communities treat software dynamicity visualization in a disconnected manner. There exist similarities in the two visualizations, which suggests their possible unification.

A. Examples of software dynamicity problems

Here are two examples of software dynamicity problems, one for each time dimension stated above.

1) Software execution problem: Consider the task of understanding and analyzing classes' roles in different execution scenarios. We collect data about multiple executions that represent use cases with their main scenario and alternative scenarios. The main scenario describes the *normal* execution of the use case, while alternative scenarios detail the possible extensions and special cases of the main scenario. Each execution scenario brings into play several classes that contribute to its fulfillment. Classes intervene in an execution scenario at different degrees depending on their roles in the software. For instance, core classes are triggered in a majority of scenarios but with low frequencies and generally at the beginning of the execution. On the other hand, specialized classes appear in specific scenarios with high frequencies. When analyzing classes' contributions in program execution using visualization, one must consider the representation of the time aspect with respect to classes interventions in the execution. Also, the aggregation of different execution scenarios in one visualization is key to detect differences or similarities, such as core classes.

2) Software evolution problem: The second example of a software dynamicity concerns the study of developers' collaborations and contributions during software evolution. Usually, software development projects are run by teams of developers. Each developer contributes to the software with a certain degree. Also, several developers may concurrently collaborate to the software or at different periods of time. Developers operate on software classes and perform changes on them from one version to another. The developers' changes differ in importance, size, and frequency. The visualization of developers' contributions using visualization must take into consideration the evolutionary behavior of software as well as the sequence of changes made by developers on software entities. Furthermore, it has to provide a mean of comparing and combining multiple developers' contributions.

B. Dynamicity representation framework

Software dynamicity as defined earlier can be described using a representation framework that defines its key elements. Here is such framework:

Event: It is an action that occurs periodically during the time dimension. An event is triggered by subjects over time and causes the overall state to change.
In example (1), an execution event occurs when a method in a class is executed. It is triggered by the execution of an instruction and causes the execution state to change. In example (2), an evolution event is defined as a change in the software structure. It is triggered by developers who change the software at a certain time of its life cycle. The event is characterized by the importance of the change, its size, and the time of the change (version).

Entity: There are two types of entities: entities that trigger the event (subjects) and entities that undergo the event (objects). Usually, the entity of interest is the subject as it contributes to the software. Subjects' contributions are evaluated on objects and both entities are central in the representation.
In both examples, the objects are the software classes, while the subjects are the use cases (1) and the developers (2).

State: It is state of the software at a given moment. The subjects cause events that impact on objects. The change incurred by the objects results in an overall state change.
The execution event in example (1) is triggered by the use case execution (entity) that changes the execution state (call stack, objects and variables' values, etc.). In example (2), the developers (subjects) contribute to a software by modifying its classes (objects) code. Hence, the software structure (state) changes due to the event.

Entities' contribution over time: Each entity contributes to a certain degree to the software and it does that at different moments of the time scale considered.
The classes' appearances in the different use cases are an example of entities' contributions in example (1). Developers changes to the software's classes over time constitute the entities' contribution in example (2).

Aggregation of entities' contributions: The entities' contributions give valuable information for the analysis tasks at hand but one often needs to combine several entity contributions to be able to answer analysis questions involving several entities' contributions.
In example (1), the core classes' identification brings into play several classes' contributions (one for each scenario) that have to be aggregated to identify classes appearing in most use cases and early during execution. In example (2), consider the task of studying the developers' collaboration and determining which classes are subject to contributions by several developers. Several developers' contributions must be aggregated to identify classes changed by multiple developers at a given time.

IV. Visualizing dynamicity with heat maps

Our primary goal is to analyze how different subjects (e.g., use-case scenarios or developers) contribute over a time period (e.g., execution or evolution) to the state change (e.g., execution time or code) of a given large-scale software system. In our setting, we are interested in changes made to the states at different levels: classes, packages, and system. In the remainder of this section, we first show how heat maps allow rendering the state changes. Three important aspects are discussed, in particular:

- *Integration with an existing visualization metaphor* to add dynamic information to other displayed software dimensions.
- *Choice of the color schemes* to ease the perception of dynamic information despite the size of the studied system.
- *Package and class placement* to use a heat map to highlight regions of interest rather than the coloration of individual classes.

The second part of this section is dedicated to the combination of multiple heat maps to perform analysis that involves many subject contributions. Finally, the last part details the navigation features that help analysts in their tasks.

A. Representing entities' contributions

Entities' contributions, e.g., classes' participation in an execution scenario or classes' code change made by a developer during the evolution, are represented using heat maps. A heat map offers a convenient technique to visualize the software's time dimension, as it adds a visualization layer on top the actual software visualization.

Integration with VERSO*:* In our work, we use *VERSO* as the visualization basis, and extend it with heat maps to represent the entities' contributions over the time. As mentioned earlier, *VERSO* provides a visualization of a program (packages, classes, relationships between classes, etc.). A heat map augments this representation by applying a color gradient on the software visualization, which adds different information from the one already conveyed by *VERSO*. The heat map visualization is orthogonal to *VERSO*'s visualization. It can be used on other software visualizations with minor modifications when 3D elements are placed on a plane (2D).

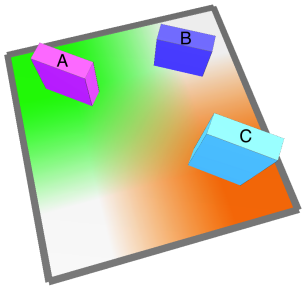
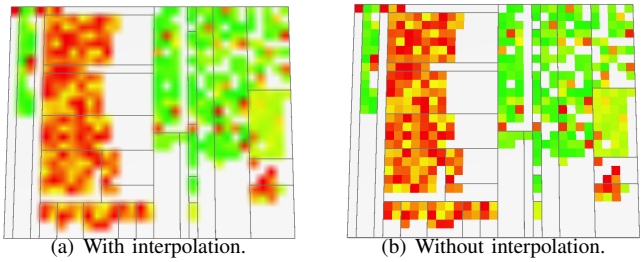


Fig. 1: The heat map is applied on the entire square region representing a package with three classes: A, B, and C. It shows that the state of B does not change, while the state of A changes with a lesser degree than one of C.

Given an object-oriented program composed of a hierarchy of packages, where each package contains classes, the root package represents the rectangle encompassing the entire scene and determines the size of the heat map. A heat map covers the entire rectangle of the root package and conveys the proper dynamic information about each class contribution by associating the class' data to a corresponding color as shown in Figure 1. Hence, we consider the implementation of heat maps as color textures (2D array of color pixels, known as "texels" for texture elements). This implementation choice comes naturally considering the graphical properties that we want to visualize. A data array containing the information to be visualized is computed and used to generate a color texture with the appropriate colors at the corresponding coordinates. As stated earlier, heat maps are meant to represent the time dimension of a software, and as such, we introduce two ways of representing this dimension. Firstly, the colors of the heat map may represent the time or age of a certain information, e.g., time since the last change made by developer Joe on the classes. On a color scheme representing the heat map, the color intensity represents the age of the last changes. Secondly, heat maps may represent accumulation events' effects triggered by the same subject. For example, the color intensity represents the total execution time used by the methods of each class. In some cases, we want the heat map colors to be "plain" colors, for instance, to compare two specific colored regions. In other cases, the visualization requires color interpolation to give a visual impression of the phenomenon as a whole. We use bilinear color interpolation between the regions (see Figure 2). The color interpolation helps highlight regions of interest by blurring the edges of the individual colored cells.



(a) With interpolation. (b) Without interpolation.

Fig. 2: Interpolation of heat map colors.

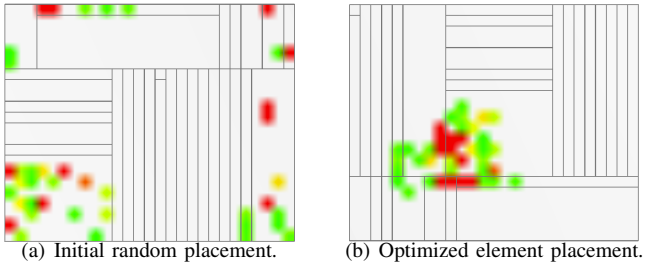
Colors: A color texture is represented by an array containing the different color values computed. These colors must reflect the data distribution that they represent. For this matter we considered several color systems that have different visualization characteristics.

Color scales for uni-variate data, as used in heat map visualization, should respect some desired visual properties [20], such as colors chosen to visualize ordered data values must be perceived as following the same order. We use the analogy of heat to produce this perception of order. Colder colors (green) are lower than warmer colors (red). The distance between two colors should also be representative of the distance between the two corresponding data values. Furthermore, clearly separated data values should be represented by distinguishable colors, and closer data values should correspond to variations of the "same" color. The color visualization should not introduce perception side effects, and not create fake boundaries, i.e., if data values do not have boundaries, the colors should not suggest that there are some. Also, the color visualization must not convey a certain organization of data if it is not present in the data itself. For example, color arrangement should not suggest clusters if the corresponding data do not aggregate.

Element placement: Usually heat maps are used on representations in which concerned elements have meaningful positions. In the case of software systems, the positions of classes and packages are not absolute and are determined to show a given property such as the system architecture. The treemap layout algorithm used in *VERSO* arranges software entities on a rectangular region. The algorithm operates on a tree structure to subdivide the rectangle into smaller regions representing software elements (packages and classes). It processes the software elements starting from the tree root and recursively traverses the tree, but the processing order of the node's children is random. Therefore, we take advantage of the two degrees of freedom (subpackages and classes) to augment our heat map visualization by enforcing particular element placements according to the visualization needs. In order to take full advantage of the heat map visualization, we group classes that are part of the heat map, i.e., classes that have a data value to be displayed in the heat map visualization. This element placement allows for a better analysis of the heat map because interesting elements that have similar values and therefore similar colors in the heat map are put closer together. This was motivated by the fact that when analyzing a heat map, it is easier to have the interesting software elements closer to each other rather than scattered over the entire base rectangle. Hence, we consider element placements that minimize the distances between all the interesting software elements.

The search space covers all possible permutations of the subpackages, and for each package, all possible classes positions within their parent packages. The size of this search space is prohibitively large and results in a combinatorial explosion. Therefore, a brute force or an exhaustive search would be very inefficient. Moreover, we do not necessarily need the optimal solution to element placement, as our goal is to improve the heat map visualization. Therefore a near-optimal solution should prove satisfactory for our goal. For this matter, we view element placement as an optimization problem that could be solved using a meta-heuristic algorithm.

In order to search for a layout that minimizes distances between classes with similar colors in the heat map, we use a simulated annealing (SA) algorithm [12]. SA is a local search meta-heuristic inspired by the metal annealing process of metallurgy, where a crystalline solid is heated and then cooled down according to a cooling schedule until it reaches its optimal energy state, and thus is free of defects. For the layout optimization, we start by an initial layout, and using a pseudo temperature with a cooling scheme, we simulate iteratively the state change by exploring neighboring solutions. The generation of a neighboring solution from a current one combines two strategies. First, we randomly choose a level in the software package tree structure, from which we select two sibling packages. We swap those two sibling packages' position, which does not alter the software structure (first degree of freedom). Then, we randomly choose two packages, not necessarily the ones chosen in the first stage. We select two classes from each chosen package, and we swap their relative positions within their parent package (second degree of freedom). Each candidate solution is evaluated using an objective function that computes the sum of the relative euclidean distances between the classes involved in the heat map, and compared with the current solution. Solutions that improve the fitness are automatically accepted. Those with a fitness deterioration could be accepted with a probability that decreases along with the cooling process and the level of deterioration.

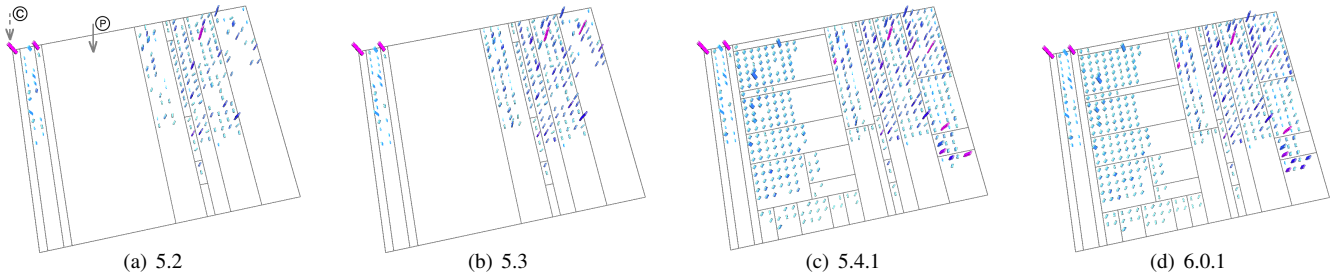


(a) Initial random placement. (b) Optimized element placement.

Fig. 3: Element placement optimization starts with an initial solution that is then optimized by swapping sibling packages and sibling classes.

Figure 3(a) displays a heat map of the method executions per class when using *Pooka* [2] system for a use-case scenario, where element placement is done with the treemap and random order of the sub-packages and classes. Figure 3(b) illustrates the same data with the optimization of element placement. The heat map visualization is easier to analyze when classes involved, with the same degree, in the execution scenario are closer to each other. For instance, to compare two classes with similar colors, we should minimize the distance between them, thus reducing the scanning effort and potential visual perturbations. In addition, it becomes easier to perform an action of zooming on features if the area of interest is smaller.

Another placement issue is when a heat map displays data related to evolution. Indeed, software structure changes over time, as from one version to another, classes and packages may be removed, added, or modified. The representation of these changes should not alter the visual coherence of the overall navigation and interaction with the scene. For instance, a class is removed and another is added at a certain version, the added class should not be positioned at the location of the removed class to avoid confusion. The same is true for entire packages. To have consistent heat maps, we used a fixed positions layout [14] where all software elements remain at the same positions during the visualization of each version. The elements' positions are computed for all versions at the beginning of the visualization by constructing a virtual tree representing all elements that existed at any version. The virtual tree contains the hierarchy level of the elements that is used for the treemap layout explained above. Figure 4 illustrates the fixed positions layout computed for four versions of *JHotDraw* [1]. The package Ⓟ, which contains tests, is added only in version 5.4.1 and hence appears only in the last two versions, but its space is present since the initial version. The class Ⓒ remains in the same position throughout the



(a) 5.2 (b) 5.3 (c) 5.4.1 (d) 6.0.1

Fig. 4: Fixed positions layout for four versions of *JHotDraw* system.

entire system evolution.

B. Aggregation entities' contributions

For some comprehension tasks, it is necessary to analyze two or more heat maps corresponding to the contributions of different subjects. For example, one could execute different scenarios of the same use case and study the involvement of the classes for all the variations of the use case. To this end, we utilize three different strategies:

Multiple windows: Each subject's contribution is represented by a heat map and rendered on a separate scene and window. The analysis is done by visually comparing the different scenes where the different contributions are depicted. Several interaction features are provided to facilitate the navigation between the different windows (see Section IV-C).

Flipping: The heat maps representing the different contributions are all displayed in the same scene. However, only one heat map is rendered at any given time. The comparison of heat maps is done by switching the rendered heat map, only the heat map that is rendered on the surface plane is changed, the scene remains the same.

Color weaving: To represent multi-variate data (multiple heat maps), we tested color blending and color weaving. Both techniques use multiple color scales, one for each variable (heat map) to be visualized. Color blending consists of mixing the color values of the represented variables, thus resulting in one computed color. However, issues with color blending are the identification of individual variables and the resulting colors might not be very meaningful. Color weaving is performed by representing the individual colors side by side in a higher-frequency color texture. We explored the use of color weaving to represent multi-variate data as it performs better than color blending when the dimensionality of the data increases [11]. Figure 5 illustrates a close-up view of two different heat maps combined in the same view. The resulting texture is obtained by subdividing each individual region of a software element into a number of texels (100×100 texels, Figure 5 right) and by introducing noise with the location of the two colors of the same region in the heat maps to be combined.

C. Navigating in views

VERSO provides an interactive visualization environment. Most software maintenance tasks are too complex to be completely automatic. Hence, human intervention is often needed during the analysis tasks. Heat map visualization follows the same principle and allows the analyst to navigate within the 3D scene. Some navigation and interaction features are:

Camera: The 3D scene camera allows the user to change the point of view from anywhere in the scene, as well as zooming in and out of it. In the multiple windows view, a camera synchronisation feature helps maintaining a visual coherence between the windows for comparison sake.

Color-scale manager: This feature permits to filter a heat map within a subset of interesting values. It also allows the user to re-map a range of values to a wider color range in order to better distinguish between closer values. Figure 6 illustrates these features.

Histogram: The color-scale manager interface provides a histogram of the values displayed in the heat map. This histogram gives an intuition about the distribution of the heat map data. It also gives the user extra information about high concentrations of values that may need to be filtered and re-mapped in order to be analyzed separately.

Navigation: Switching between several heat maps must be as simple as possible, because it is one of the most frequent operation used during exploration and analysis of the data. This is associated with the up/down arrow keys and its result is instantaneous. We also use left/right arrow keys to switch between versions of the software.

Scene clearing: Despite our efforts to make the scene less cluttered, we sometimes need to clear the scene where all the attributes are rendered (see Figure 7(a)) in order to better visualize specific graphical elements. For this purpose, the user may hide the 3D boxes and display only the heat map (see Figure 7(b)) and vice versa (see Figure 7(c)). The user can also keep some contextual information by given the 3D boxes desaturated colors and fixed heights, as shown in Figure 7(d).

V. Illustrative Case Studies

To illustrate the use of our heat map visualization, we discuss two case studies involving two time-based software comprehension tasks. The first case study concerns the evolution of the *JHotDraw* [1] software and the second one targets the analysis of the *Pooka* [2] software's features.

A. Software evolution

Objective: To understand some aspects of software evolution, an analyst needs to combine time-related information

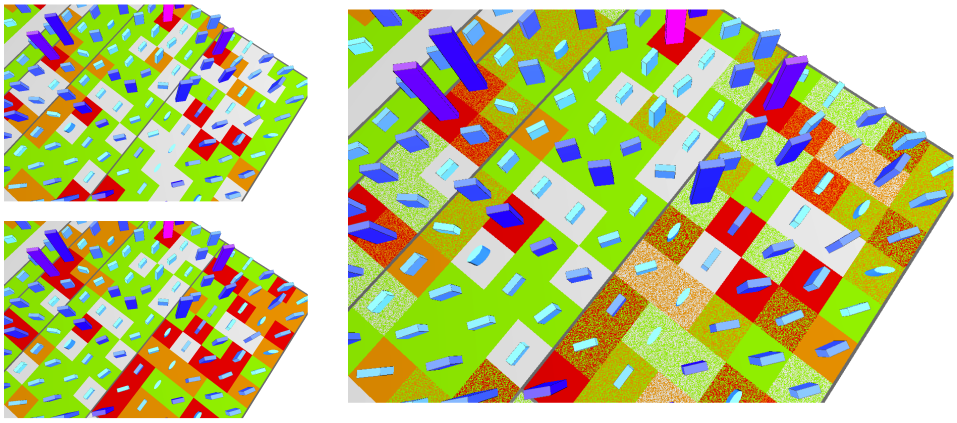
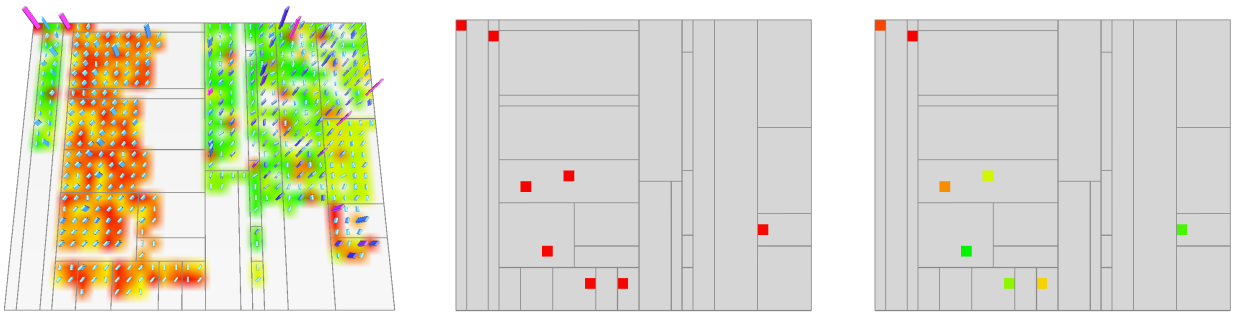
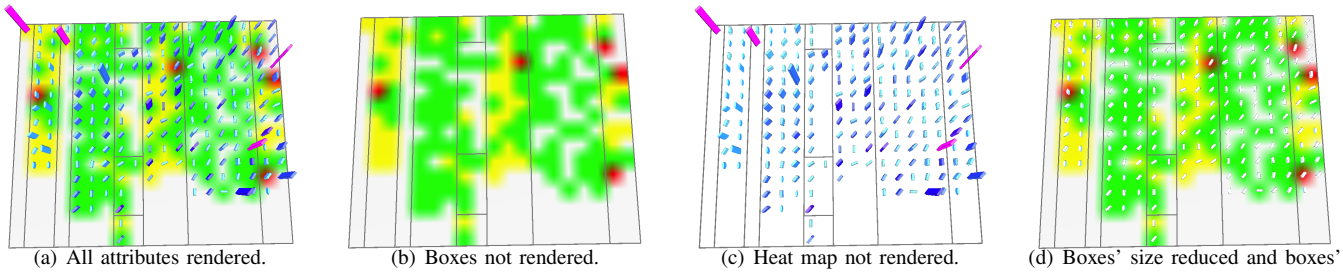


Fig. 5: The two heat maps on the left are combined on the right using color weaving with a high-frequency color texture.



(a) No filtering. (b) Filtering. (c) Remapping on a wider color gradient.

Fig. 6: The color gradient on the rectangle represents the time/age of the changes accomplished by developer *mrfloppy* in *JHotDraw* 6.0.1.



(a) All attributes rendered. (b) Boxes not rendered. (c) Heat map not rendered. (d) Boxes' size reduced and boxes' colors desaturated.

Fig. 7: Options for scene clearing and color interpolation.

with other software properties. In *VERSO*, we used heat maps to represent time information of software evolution in addition to static software information mapped to the 3D boxes. For the sake of illustration, tasks are defined as questions to answer for program comprehension.

Tasks and Data: During maintenance tasks related to software evolution, developers often ask recurrent questions. Fritz and Murphy [10] determined several such questions by interviewing professional developers. The developers' questions are organized by domains of information from which answers can be found, such as source code, bug reports, test cases, etc. Because we visualize program changes over time, we are interested in two domains: source code and change sets. Here are some of these developers' questions, related to these two domains, that we answered using our visualization technique:

A) Who is working on what?
B) What are coworkers working on?
C) Who changed this class?
D) What is the most popular class?
E) Who is working on the same classes as I am?
F) What classes have been changed?
G) Which class has been changed most?

We considered four versions of *JHotDraw* (5.2, 5.3, 5.4.1, and 6.0.1), a Java GUI framework for technical and structured graphics. The numbers range from 14 to 36 packages, and from 171 to 498 classes. The SVN logs of each version were extracted and parsed to retrieve information about the contribution of the different developers in each version. There are eight developers whom contributed to the *JHotDraw* project over the four considered versions. The collected data have been organized into a matrix, where the first dimension represents developers, and the second represents versions. Such an organization permits to visualize a developer's contribution over several versions of a program with the possibility to compare multiple contributions of the developers. The visualization techniques offer a way to represent the developer's contribution under different facets. For example, we can visualize the importance of the contribution as the proportion of changes made to different classes. We can also visualize the recency of the contributions. The modifications can be filtered by type: all changes, additions, modifications, and removals.

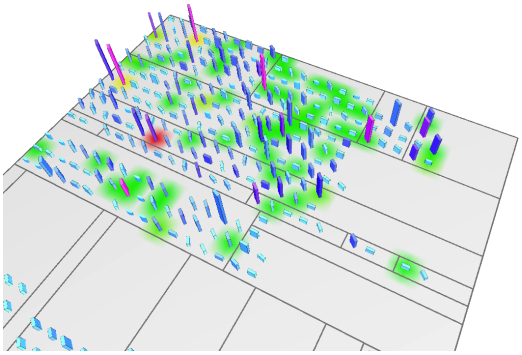


Fig. 8: *ricardo_padilha*'s contributions to *JHotDraw* 6.0.1.

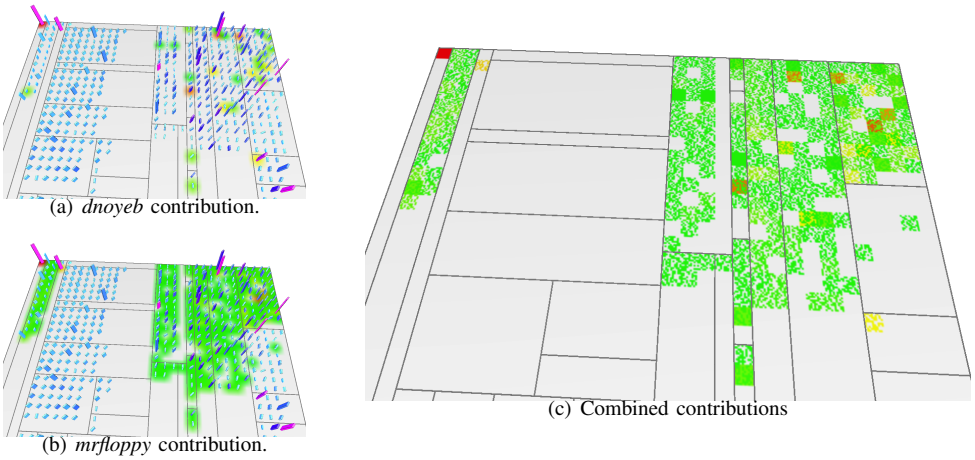
Analysis: We used our heat map visualization within the *VERSO* framework to answer the selected questions by generating heat maps representing the system's degree of changes, which corresponds to the extent to which the software differs between two time stages. The information about change is organized by developer and by version of the system, hence, a heat map corresponds to one developer who worked on a particular version of *JHotDraw*. Figure 8 shows the changes made by developer *ricardo_padilha* in *JHotDraw* 6.0.1. The changes include additions, modifications, and removals of classes. The colors represent the importance of the changes made. For instance, *ricardo_pardilha* made the most changes to class *org.jhotdraw.figures.TextFigure*, which appears in red in the heat map of Figure 8. To answer most of the questions with respect to a developer, we generate heat maps representing the developer's contributions. These heat maps answer question A. Question B can be answered by aggregating all these heat maps and visualizing the resulting heat map, which would include the contributions of all the developers and their importance. For question E, we compare the heat maps one by one with the heat map representing the developer asking the question in order to identify classes in common. For instance, developers *mrfloppy* and *dnoyeb* have 26 classes in common in version 5.4.1. Figure 9(c) illustrates the contributions of these two developers to *JHotDraw* 5.4.1. The analysis is done by comparing the two heat maps. The comparison can be achieved either by generating two scenes side by side displaying each of the heat maps, as shown by Figures 9(a) and 9(b), or by flipping interactively between the two heat maps on the same scene, and noticing the differences, or by combining the heat maps in the same view by weaving their colors (see Figure 9(c)). We used a coarse-grained texture in this case (compared to Figure 5 (right)) to highlight classes in common, where there are two colors in the class region, in comparison to classes that do not appear on both heat maps, where the background color is weaved with the one color. Questions D, F, and G can be answered in the same manner with the generation of a heat map that represents the aggregation of all changes made by all developers. This resulting heat map would show the most popular classes in red (for question D). Filtering this heat map to show only modifications will answer question F (all classes with a heat map color) and question G (classes with a color red). Finally, to answer question C, one can generate a heat map for each developer with the age rather than the amount of changes. Then, the heat maps are compared side by side or by flipping. The responsible is the developer with the heat map exhibiting the reddest color for this class.

B. Software execution

Objective: We consider program execution from the perspective of time, and visualize it using heat maps. These are used as an exploratory technique as well as for specific tasks, such as feature location or identification of core classes in alternative executions. Our goal is to help the user gain insight into a system without up-front knowledge, and also get into the system's features by performing alternative use cases.

Tasks and Data: We used our visualization technique to analyze features from *Pooka*. We performed a visual analysis of execution traces to understand classes' participation in different use cases. The tasks taken for the case study are inspired by the case study reported by Cornelissen et al. [7].

We collected the execution traces of the software *Pooka*, an email client written in Java, using the Javamail API. There are 301 classes organized in 32 packages. We used a custom extraction agent written in C that utilizes the jvmti API to listen to events triggered at the method's entry or exit. Each time a method of our considered software is called during the execution, the agent captures the entry event and returns information about the method, such as its parent class, the thread within which it is executed, its signature, a time stamp, etc. The same information is collected when the method exit event is captured. This information is then processed to be aligned for each method to produce a call graph with the execution time of methods. The execution time is relative, in the sense that nested methods' execution times are included in the outer method. The call graph generated includes multiple execution threads and can be traversed in the same order as the events were triggered during the execution. It can also be traversed by following a particular execution thread. System



(a) *dnoyeb* contribution.

(b) *mrfloppy* contribution.

(c) Combined contributions

Fig. 9: The combination of heat maps representing the contributions of developers *dnoyeb* and *mrfloppy* to *JHotDraw* 5.4.1.

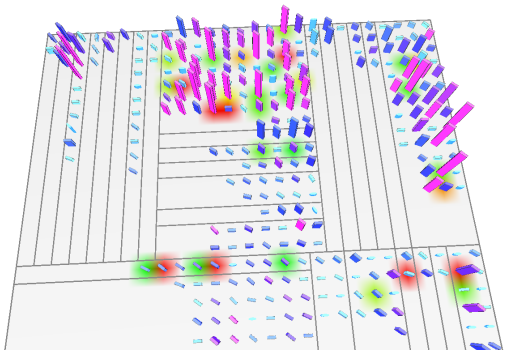
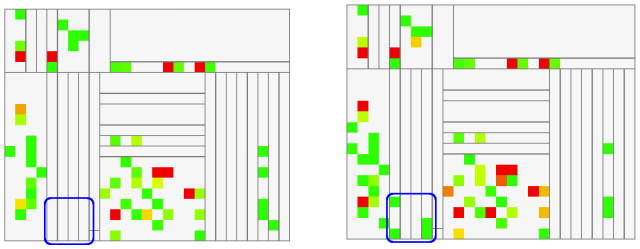


Fig. 10: Class activity in the use case *open email, add sender to address book, and close email*.



(a) *open email and close it.*

(b) *open email, add sender to address book, and close email.*

Fig. 11: Comparison of two alternative executions of use case. Classes in the blue rounded square are active in Figure 11(b), but have no activity in Figure 11(a).

method calls are filtered out as well as unnecessary events, such as mouse hovers, panels repainting, etc. After filtering, we organized the traces by use case; each use case regroups several alternative execution traces. We recorded 37 traces of user interactions with the system and organized them in three main use cases: "*read mail*", "*inbox actions*", and "*search mail*". For example, use case "*read mail*" includes an execution trace of opening an email without attachments, one of opening an email with attachments, and one of opening an email and searching for a word within the email, etc.

Several information types can be visualized using the techniques presented in this paper. For instance, we can represent the age of each class in the execution stack. We can visualize the activity (cumulative execution time) of each class in an execution trace. Finally, we can visualize the occurrence of classes over several execution traces. These visualizations are then used to compare different use cases and execution traces.

Analysis: We generate a heat map for each execution trace representing an alternative use case. The resulting color gradient reflects the appearance of the class in that particular use case. Hence, every heat map shows the classes participating in the use case and its contribution to its execution. A color red indicates high activity within the use case, while a color green indicates low activity. For instance, "*open an email, add the sender to the address book, and close the email*" triggers 31 classes in seven different packages. Each class has a different level of activity when this use case is executed. Figure 10 shows class activity during the execution of the use case. We can see the 31 classes contributing to the use case. There are eight classes with high activity (red) and the class *net.suberic.pooka.gui.dnd.DndUtils* is the one with the least activity in this particular use case. Another important task that our technique helps the user to perform is the identification of core classes that appear in several alternative use cases. This task is achieved by aggregating multiple alternative use cases and computing a heat map representing the number of class occurrences. The heat map indicates the number of alternative use cases where the classes appear. Core classes appear generally in several executions of the use case. The heat map visualization technique enables the user to narrow the search for classes responsible for minor changes in executions. For instance, consider the two alternative use cases: "*open email and close it*" and "*open email, add sender to address book, and close email*". We generate two heat maps representing the class activity of the two use cases, and we

compare them with the techniques stated above. The result shown in Figure 11 indicates clearly that, for example, classes *FileResourceManager*, *ResourceManager*, *VcardAddressBook*, and *Vcard* (circled in blue) are active in Figure 11(b), but not in Figure 11(a), which suggests that they are responsible, with a few others, for the treatment of *add sender to the address book*. To identify core classes, we generate heat maps representing the classes' activity in the alternative executions. Then, we aggregate them and concentrate on classes that appear in red, i.e., classes active in most alternative executions.

VI. Conclusion and Discussion

In this paper, we explore the visualization of software dynamicity. We consider program changes that occur during execution and evolution. The heat map metaphor used in our visualization technique is suitable for the representation of a program's time dimensions. Hence, in an attempt of unification, we apply the heat map metaphor for the representation of program execution and evolution. Heat map visualization allows the distinction between the software's dynamicity and its overall context. It allows the developer to analyze the system's temporal dimension and to keep in mind the general context that is often needed to understand the system.

In order to illustrate the use of our visualization approach, we realized two case studies on two different systems for the two software dynamicity aspects. The case studies illustrate that heat maps permit to answer practical questions and offer a simple and precise way about where to start the search for an answer. Consequently, they corroborate our claim that the two software time dimensions studied may be represented and analyzed using the same visualization metaphor.

Our studies have also revealed that there are still open issues when representing time aspects on top of other software properties. The color systems used for the heat map visualization are device-dependent models, and as such, do not relate well with the way colors are perceived. In particular, the euclidean distance between two color values in the color system should be proportional to their perceptual distance. This property is present in device-independent color systems such as CIE LAB and CIE LUV, which are perceptually more uniform. Even though we did not feel penalized by this limitation, we plan to utilize a perceptually coherent color system in order to augment the heat map visualization. Another issue is the challenging problem of layout stability while working on software evolution. Our layout algorithm works when we consider previous versions up to the current version. However, for future versions, our algorithm would completely re-arrange the elements to consider a recent version. This would result in a major change in classes' positions from one version to another, and temporal coherence in the scene could be affected. We plan to tackle layout stability in future work in order to preserve some coherence as the system's structure evolves. We consider two approaches: (1) use Voronoi treemaps [3], which improves stability by relaxing the constraint of rectangular subdivisions, thus allowing arbitrary shapes, or (2) use another layout type, such as *EvoStreets* [22], which improves stability by incrementally incorporating new changes to the software's structure.

References

[1] JHotDraw: a Java GUI framework. http://www.jhotdraw.org.

[2] Pooka system: a Java email client. http://www.suberic.net/pooka.

[3] M. Balzer and O. Deussen. Voronoi treemaps. In *Proc. IEEE Symp. on Information Visualization*, pages 49–56, 2005.

[4] J. Bohnet, M. Koeleman, and J. Döllner. Visualizing massively pruned execution traces to facilitate trace exploration. In *Proc. IEEE Int. Workshop on Visualizing Software for Understanding and Analysis*, pages 57–64, 2009.

[5] A. A. Bojko. Informative or misleading? Heatmaps deconstructed. In *Proc. Int. Conf. on Human-Computer Interaction. Part I: New Trends*, pages 30–39, 2009.

[6] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *Proc. ACM Symp. on Software Visualization*, pages 77–86, 2003.

[7] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proc. IEEE Int. Conf. on Program Comprehension*, pages 49–58, 2007.

[8] B. Cornelissen, A. Zaidman, B. van Rompaey, and A. van Deursen. Trace visualization for program comprehension: A controlled experiment. In *Proc. IEEE Int. Conf. on Program Comprehension*, pages 100–109, 2009.

[9] M. Fischer and H. Gall. Evograph: A lightweight approach to evolutionary and structural analysis of large software systems. In *Proc. Working Conf. on Reverse Engineering*, pages 179–188, 2006.

[10] T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *Proc. ACM/IEEE Int. Conf. on Software Engineering - Volume 1*, pages 175–184, 2010.

[11] H. Hagh-Shenas, V. Interrante, C. Healey, and S. Kim. Weaving versus blending: a quantitative assessment of the information carrying capacities of two alternative methods for conveying multivariate data with color. In *ACM SIGGRAPH 2006 Research Posters*, 2006.

[12] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[13] G. Langelier, H. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proc. IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 214–223, 2005.

[14] G. Langelier, H. Sahraoui, and P. Poulin. Exploring the evolution of software quality with animated visualization. In *Proc. IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 13–20, 2008.

[15] D. Mashima, S. Kobourov, and Y. Hu. Visualizing dynamic data with maps. In *Pacific Visualization Symposium*, pages 155–162, 2011.

[16] T. M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. 1996.

[17] A. Pryke, S. Mostaghim, and A. Nazemi. Heatmap visualization of population based multi objective algorithms. In *Proc. Int. Conf. on Evolutionary Multi-criterion Optimization*, pages 361–375, 2007.

[18] M. Renieris and S. P. Reiss. Almost: exploring program traces. In *Proc. Workshop on New Paradigms in Information Visualization and Manipulation*, pages 70–77, 1999.

[19] D. Roethlisberger, O. Nierstrasz, S. Ducasse, D. Pollet, and R. Robbes. Supporting task-oriented navigation in IDEs with configurable heat maps. In *Proc. IEEE Int. Conf. on Program Comprehension*, pages 253–257, 2009.

[20] S. Silva, B. S. Santos, and J. Madeira. Using color in visualization: A survey. *Computers and Graphics*, 35(2):320–333, 2011.

[21] O. Spakov and D. Miniotas. Visualization of eye gaze data using heat maps. *Electronics and Electrical Engineering*, 2(74):55–58, 2007.

[22] F. Steinbrückner and C. Lewerentz. Understanding software evolution with software cities. *SAGE*, 12(2):200–216, 2013.

[23] J. Trümper, J. Döllner, and A. Telea. Multiscale visual comparison of execution traces. In *Proc. IEEE Int. Conf. on Program Comprehension*, 2013.

[24] L. Voinea and A. Telea. Multiscale and multivariate visualizations of software evolution. In *Proc. ACM Symp. on Software Visualization*, pages 115–124, 2006.

[25] R. Wettel and M. Lanza. Visual exploration of large-scale system evolution. In *Proc. Working Conf. on Reverse Engineering*, pages 219–228, 2008.