Evaluating a Visual Approach for Understanding JavaScript Source Code

Martin Dias
University of Chile

Diego Orellana
University of Chile

Santiago Vidal
ISISTAN-CONICET

Leonel Merino
University of Stuttgart

Alexandre Bergel*
Dept. of Computer Science,
University of Chile

ABSTRACT

To characterize the building blocks of a legacy software system (*e.g.,* structure, dependencies), programmers usually spend a long time navigating its source code. Yet, modern integrated development environments (IDEs) do not provide appropriate means to efficiently achieve complex software comprehension tasks. To deal with this unfulfilled need, we present *Hunter*, a tool for the visualization of JavaScript applications. Hunter visualizes source code through a set of coordinated views that include a node-link diagram that depicts the dependencies among the components of a system, and a treemap that helps programmers to orientate when navigating its structure.

In this paper, we report on a controlled experiment that evaluates Hunter. We asked 16 participants to solve a set of software comprehension tasks, and assessed their effectiveness in terms of (*i*) user performance (*i.e.,* completion time, accuracy, and attention), and (*ii*) user experience (*i.e.,* emotions, usability). We found that when using Hunter programmers required significantly less time to complete various software comprehension tasks and achieved a significantly higher accuracy. We also found that the node-link diagram panel of Hunter gets most of the attention of programmers, whereas the source code panel does so in Visual Studio Code. Moreover, programmers considered that Hunter exhibits a good user experience.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**; **Software reverse engineering**; *Software maintenance tools*; *Software evolution*; *Maintaining software*.

KEYWORDS

Software visualization, Software comprehension, JavaScript

*Corresponding author: abergel@dcc.uchile.cl

ACM Reference Format:
Martin Dias, Diego Orellana, Santiago Vidal, Leonel Merino, and Alexandre Bergel. 2020. Evaluating a Visual Approach for Understanding JavaScript Source Code. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3387904.3389275

1 INTRODUCTION

Software developers interact with mainstream programming environments essentially through textual elements. Typically, structural elements, including files, classes, and packages, are presented as an expandable list widget on the left-hand side of the IDE and the content of a selected element is displayed in the center as a large textual panel. Visualization is known to be effective at assisting practitioners in carrying out software comprehension and maintenance tasks [2, 8, 12, 21, 22].

This paper seeks to complement the classical IDE layout with simple but effective visualizations to support some non-trivial comprehension tasks. Our prototype of such an IDE augmentation is called *Hunter*. Hunter is useful to navigate, analyze, and comprehend JavaScript applications. The approach is based on an interactive *dependency visualization graph* that is intuitive and technically easy to implement. Hunter is designed to assist developers in the process of software comprehension. For example, Hunter allows developers to quickly characterize the relevant structures of a software system and their dependencies. Hunter supports multiple interactions to display detailed information on demand. While there exist some commercial applications that help to visualize dependencies in software systems, such as *Structure 101*[1] and *Understand*[2], most of them do not support JavaScript.

Hunter is an interactive visual environment. Due to the limitation of a printout to adequately describes visual interactions, we recommend our readers to watch a short video about Hunter, at **http://bit.ly/2N4933e**.

We evaluated Hunter by defining a robust experimental design. We carefully measure the impact and benefits of the approach with practitioners:

(1) After defining some code comprehension tasks, we conducted a *controlled experiment* with 16 software developers that evaluate Hunter against Visual Studio Code, a popular code editor.
(2) We analyzed the performance of the developers when solving a set of tasks with Hunter and Visual Studio Code.

[1]https://structure101.com
[2]https://scitools.com/

(3) We monitored what are the relevant parts of Hunter that are effectively used by practitioners. We used a gaze tracker for that purpose.
(4) We measured the user experience of using Hunter by collecting developers' emotions and by filling a post questionnaire about the usability of the tool.

Our experiments indicate that using the visualizations provided by Hunter greatly decreases the time needed by developers to conduct some software comprehension tasks when compared to classical IDEs. Along this line, the main contributions of this paper are Hunter and a thorough evaluation of it from different perspectives.

Outline. Section 2 describes in detail the visualizations provided by Hunter. Section 3 presents the experiment design and operations. Section 4 describes the results of the experiment. Section 5 discusses the threats to the validity of our results. Section 6 analyzes related work. Section 7 presents the conclusions and outlines future work.

2 HUNTER

Hunter[3] is a standalone visualization tool designed to complement modern integrated development environments. Hunter can help programmers to understand JavaScript applications by analyzing dependencies and their structure. In this section, we elaborate on the features available in Hunter and discuss our design decisions.

2.1 In a Nutshell

To illustrate Hunter, we use the Hexo JavaScript application[4]. Hexo is a blog framework that consists of 266 JavaScript files, with a size of 22 kLOC. Like most JavaScript applications, Hexo's source code is structured into nested folders.

Figure 1 gives an overview of Hunter. Its graphical interface is composed of five panels. The left-most one (FB) is a File Browser, present in most IDEs. On the figure, the file `index.js` is selected. The panel in the center is the *File Dependencies View* (V), which represents dependencies between JavaScript source code files. The top-right panel (O) gives the file Outline that shows the structure in terms of functions nesting of a selected file. In the example, a function has been selected by the user. The left-bottom panel (S) is a Search box that can be used to identify specific files or functions in V. Finally, the right-bottom panel (SC) shows the Source Code of a selected file. In it, the selected function is highlighted.

2.2 File Browser

To improve the usability of Hunter, we included a file browser (FB in Figure 1) that is well-known for most users. We augmented the browser with colored bullets. A distinctive color is given to each root folder and its inner elements. We consistently used these colors in the file dependencies view (described below) as a visual cue that can help users to identify macro-structures in the repository.

[3]Hunter is available to be downloaded from http://bit.ly/2H95eWB
[4]https://github.com/hexojs/hexo

2.3 File Dependencies View

The File Dependencies View, marked V in Figure 1, represents (*i*) dependencies between JavaScript files, (*ii*) the size of the files, (*iii*) interactions between macro-components, and (*iv*) references to external libraries.

Figure 2 highlights a particular portion of the V panel given in Figure 1. Each circle is a JS file that belongs to the analyzed application. The color of a circle identifies a particular root folder that can be inspected in the file browser. The size of a circle indicates the number of lines of code of a represented JS file. For example, file `lib/models/post.js` is significantly larger than `lib/models/asset.js`. This view is designed to support programmers to compare the sizes of the represented JS files. The exact file size (*i.e.,* the number of lines of code) may be obtained in the SC panel.

Boxes represent external files from libraries, *i.e.,* a file that is used by the application but does not belong to it. Figure 2 indicates that both `post.js` and `asset.js` use the file `warehouse.js`, which belongs to Node.js[5]. Representing external files is relevant when assessing dependencies to externally provided libraries.

Edges among the nodes represent their dependencies in terms of usage. We place the nodes in the node-link diagram by using a force-based layout algorithm. In it, each node behaves like a repulsing magnet and edges act as springs.

Dependencies computation. Since Hunter analyzes dependencies between `js` files, they are computed based on the `require` and `import` declarations provided by Node.js and ES6[6]. We take into account both the declarations made at the beginning of the `js` files and the declarations nested into the source code.

2.4 File Outline View

When a file is selected, either in the file browser system view (FB in Figure 1) or the file dependencies view (V), then a file Outline view (O) is built. The outline is a treemap visualization technique, which uses a set of nested tiles to represent hierarchical structures. The outline represents the JavaScript functions and classes defined in a selected file. The size of the structural elements is represented by the size of the tiles of the treemap. We employed a treemap because it was shown that this visual technique is effective at representing the inner and hierarchical structure of a JavaScript file [2].

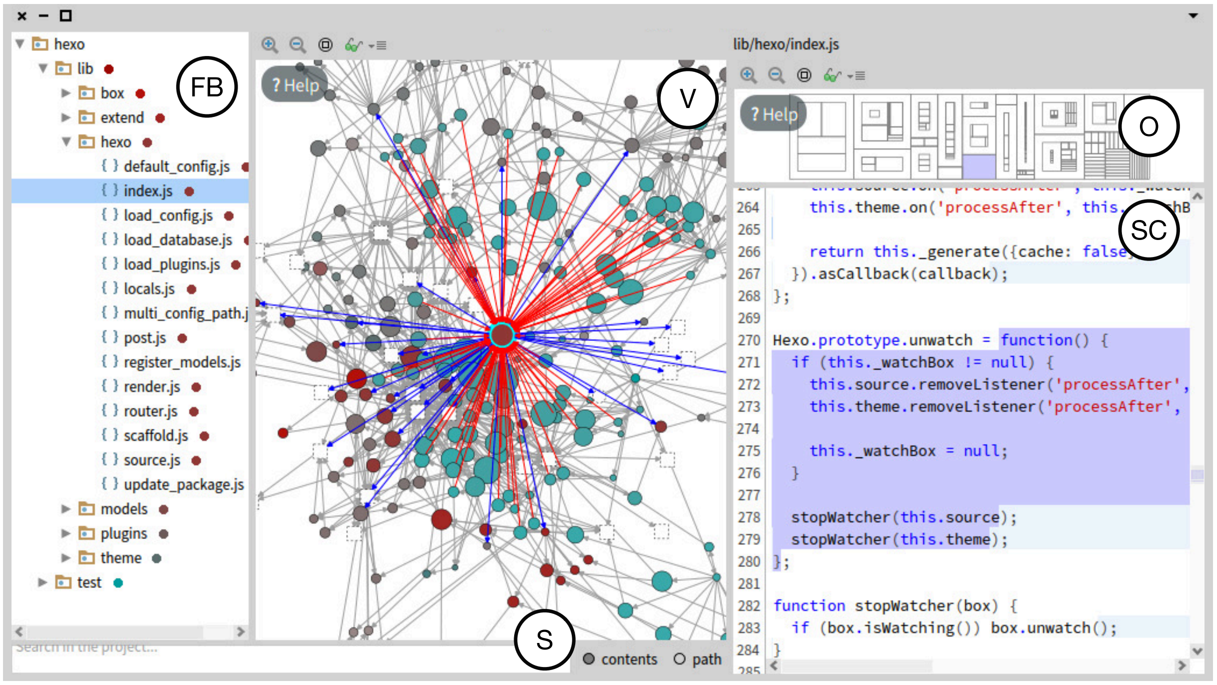
2.5 Interaction

Hunter's visualizations are highly interactive, which we exemplify by elaborating on two representative interactions.

When selecting a file, in the file browser or the file dependency view, the file is highlighted with a thick cyan border (Figure 3). Outgoing and incoming dependencies are highlighted in blue and red respectively. These interactions can be used by programmers to analyze software metrics such as fan-in, fan-out and identify components that, for instance, need to be refactored.

When selecting a folder in the file browser, all the JavaScript files that belong to the folder (or its subfolders) are highlighted (Figure 4). This interaction can be useful to identify components that are not used by others, and that eventually can be removed (*e.g.,* dead code).

[5]https://nodejs.org
[6]http://www.ecma-international.org/ecma-262/6.0/index.html#sec-imports



```
    this.theme.on('processAfter', this._watchB
    
    return this._generate({cache: false});
  }).asCallback(callback);
};

Hexo.prototype.unwatch = function() {
  if (this._watchBox != null) {
    this.source.removeListener('processAfter',
    this.theme.removeListener('processAfter',

    this._watchBox = null;
  }

  stopWatcher(this.source);
  stopWatcher(this.theme);
};

function stopWatcher(box) {
  if (box.isWatching()) box.unwatch();
}
```

Figure 1: Overview of Hunter

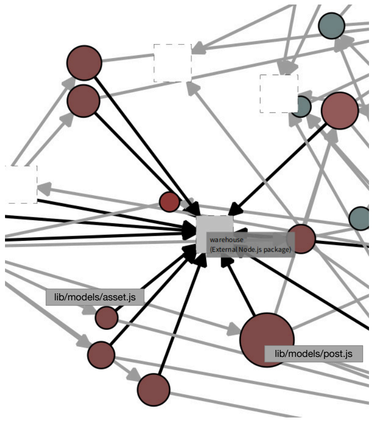
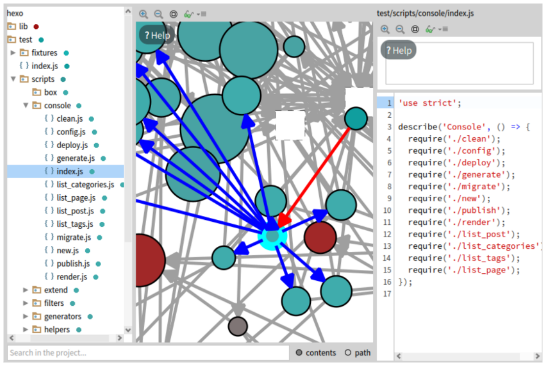


Figure 2: File dependency view



```
'use strict';

describe('Console', () => {
  require('./clean');
  require('./config');
  require('./deploy');
  require('./generate');
  require('./migrate');
  require('./new');
  require('./publish');
  require('./render');
  require('./list_post');
  require('./list_categories'
  require('./list_tags');
  require('./list_page');
});
```

Figure 3: Selecting a file

2.6 Requirements for Reverse Engineering Tools

Reverse engineering is very important to facilitate software comprehension tasks [5]. Building a tool for software reverse engineering is known to be a particularly difficult task [13, 18]. Kienle and Müller [17] have explored the issue of building tools for reverse engineering, and have identified several requirements and "generic quality attributes that reverse engineering tools should strive to meet". This section discusses how Hunter addresses these requirements and quality attributes.

Scalability. Hunter can cope with medium-sized projects. We measured the loading and visualization building time of 10 JavaScript projects, whose size ranges from 100 LOC to 104k LOC. Hunter's processing time is linear to the project size and the visualization is smooth and snappy with 104k LOC spread in 273 JS files.

Interoperability. Currently, Hunter is only able to deal with comprehension tasks that normally precede more complex reverse engineering tasks. One avenue of future work is to enable Hunter to interoperate directly with an existing programming environment.

Customizability. Custom and domain-specific rules may be defined, *e.g.*, the file dependencies view may be tailored using a set of rules to consider some aspects of the analyzed application. For example, Angular uses a particular convention to express dependencies, for which Hunter can accommodate with using regular expression matching over the abstract-syntax-tree of the analyzed application. Though, we did not include customized visualizations

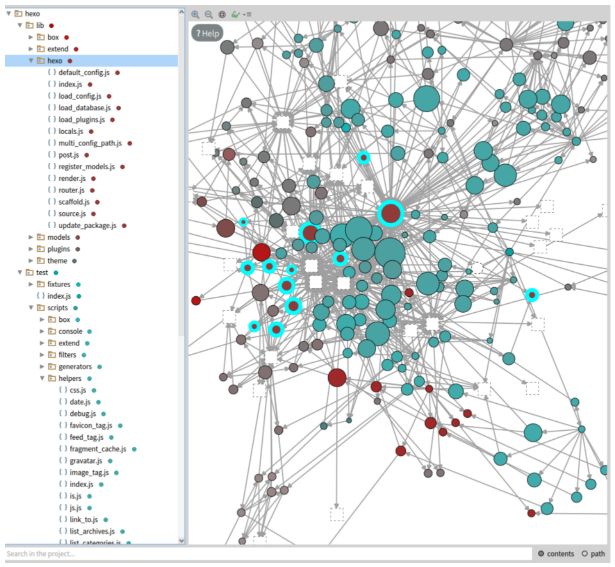


Figure 4: Selecting a folder

in our evaluation due to the specificities of customizations. Customized visualizations are therefore out of the scope of this paper.

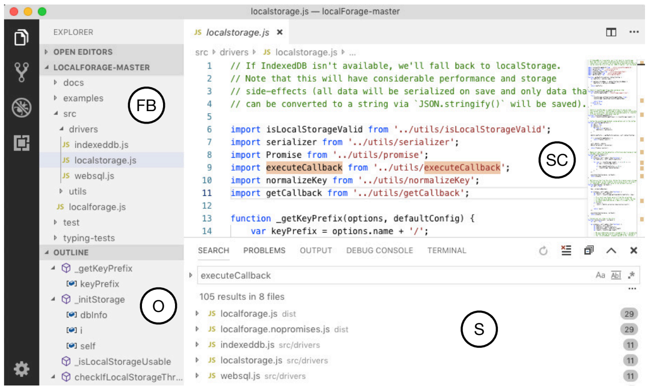
Usability. The design and implementation of Hunter were driven by five dry runs with professional JavaScript developers. The goal of these runs was to collect feedback that allowed us to improve the usability of the tool. Furthermore, details of the usability experienced by participants of our controlled experiment are discussed in Section 4.2.

Adoptability. The five dry runs were conducted with proprietary commercial JavaScript applications. The development of Hunter was guided by addressing the requirements for software maintenance and comprehension in the context of the developments of these software systems. Notice that this paper focuses on the subsequent larger evaluation we carried out, thus we do not discuss these runs any further.

3 CONTROLLED USER EXPERIMENT

The goal of our experiment is to measure and characterize the impact of using Hunter's visualizations to support some representative code comprehension tasks. Consequently, we designed a controlled user experiment to analyze the effectiveness of developers that use Hunter compared to a baseline framework, Visual Studio Code[7] (VSC), a popular IDE used for the development of JavaScript software systems. We choose VSC for several reasons:

- *Popularity.* According to the Stack Overflow 2019 Developer Survey, VSC is ranked as the most popular programming environment[8].
- *Standard layout.* VSC uses a panel layout that has become standard among other popular development environments such as Eclipse and IntelliJ IDEA. As shown in Figure 5, the VSC interface is composed of four different panels: (FB) a file browser that lists the files in the workspace; (SC) the source code editor; (O) an outline of the functions and variables defined in the file shown in (SC); and (S) a search panel.
- *Navigation and search.* Like any modern environment, VSC offers a large range of options for source code navigation and search. Notice that Hunter only supports the code navigation and search functionalities described before. As such, we constrained our comparison to such features included in a full-fledged environment (VSC) to the ones in our minimal environment (Hunter).
- *Production vs comprehension environment.* Previous studies that analyzed how programmers develop software systems [15, 19, 31] found that programmers use development environments to carry out code comprehension tasks. Based on these existing results, we decided to compare Hunter, which is designed to support software comprehension, with the capabilities offered in VSC.



```
// If IndexedDB isn't available, we'll fall back to localStorage.
// Note that this will have considerable performance and storage
// side-effects (all data will be serialized on save and only data tha
// can be converted to a string via `JSON.stringify()` will be saved).

import isLocalStorageValid from '../utils/isLocalStorageValid';
import serializer from '../utils/serializer';
import Promise from '../utils/promise';
import executeCallback from '../utils/executeCallback';
import normalizeKey from '../utils/normalizeKey';
import getCallback from '../utils/getCallback';

function _getKeyPrefix(options, defaultConfig) {
    var keyPrefix = options.name + '/';
```

Figure 5: Overview of Visual Studio Code (VSC).

Next, we describe the design, operation, and results of our experiment following the guidelines proposed by Wohlin *et al.* [38].

3.1 Design

We adopted the framework proposed by Wohlin *et al.* [38] and then adapted by Merino *et al.* [25] to describe the scope of our experiment:

"Analyze the <*Hunter*> visualization tool that supports <*software comprehension tasks*> using a <*node-link diagram*> and a <*treemap*> displayed on a <*standard computer screen*> for the purpose of <*comparison to functionalities available in Visual Studio Code*> with respect to <*effectiveness*> in terms of <*user performance*> and <*user experience*> from the point of view of software <*developers*>."

We invited professional software developers to our experiment who freely opted to participate. The experiment uses a within-subjects design. That is, each participant in the experiment was asked to solve a set of tasks using Hunter and VSC. Therefore, the independent variable is the considered treatment (Hunter or VSC).

The dependent variables we consider relate to user performance, that we measured not only in terms of traditional variables such as correctness and time that participants need to solve comprehension

[7] https://code.visualstudio.com
[8] https://insights.stackoverflow.com/survey/2019

Application			Treatment	
Name	*#LOC*	*#JS files*	*Hunter*	*VSC*
LocalForage[11]	16,265	46	Group 1	Group 2
Hexo[12]	22,325	266	Group 2	Group 1
es6-mario[13]	2,015	44	Group 3	Group 4
serverless[14]	55,258	390	Group 4	Group 3

Table 1: Applications and treatments

tasks, but also in terms of attention that we measure using eye-tracking. We also analyze variables of user experience in terms of usability and emotions.

To validate the protocol and contents of the experiment, we conducted a pilot study with a postdoctoral researcher before the actual experiment. Based on the feedback, we made adjustments and improvements in the protocol and questions. To facilitate the verifiability and reproducibility of our results, we provide a replication package that contains the used applications, questionnaires, raw data sets, and recordings of the participant sessions[9].

Apparatus/tools. All participants used the same laptop during the experiment, an Apple MacBook Pro Retina with a resolution of 2880 × 1800 pixels. Also, each participant used a Pupil Labs Core headset[10] with two cameras: one pointing to an eye (to track eye gaze) and a second camera to capture the view of the world. To interact with Hunter and VSC, participants used the keyboard and the touchpad on the laptop.

Target systems. We selected four open-source JavaScript applications of various sizes to conduct our experiment to mitigate possible threats to the validity of our experiment. Each selected application meets the following criteria: (*i*) it is open-source, (*ii*) it is written in pure JavaScript (*e.g.,* not CoffeeScript or TypeScript), and (*iii*) it is structured in more than one folder. This last requirement is a cheap way to filter out applications for which packaging and modularity were ignored by their developers. Moreover, we prioritized popular applications based on GitHub’s stars. Table 1 summarizes information about the applications used in our study and the treatments.

Tasks. Each participant was asked to perform a set of nine tasks using Hunter and another nine tasks using VSC. We observe that finding a set of representative tasks involved in a software comprehension process is difficult. As far as we are aware, no standard benchmark of tasks has been proposed yet. We, therefore, formulated our tasks inspired on tasks described in previous studies. In particular, Sillito *et al.* [31] and Kubelka *et al.* [19] observed practitioners and identified a set of frequent questions that arise when carrying out a software evolution process. Table 2 lists the tasks[15]. To avoid fatigue among participants, we included only nine tasks, which could be solved in less than 90 minutes (that we tested in a pilot study).

[9]http://bit.ly/2OWkMDC
[10]https://pupil-labs.com/
[14]https://github.com/localForage/localForage
[14]https://github.com/hexojs/hexo
[14]https://github.com/JuniorTour/es6-mario
[14]https://github.com/serverless/serverless
[15]Angled brackets indicate names that are specific to each application.

ID	Description
T1	What is the most invoked JS file?
T2	What is the JS file that most invokes other files?
T3	How many files does the <fileName> file invokes?
T4	By how many files is the file <fileName> invoked?
T5	What JS file has the most lines of code?
T6	Which JS file in the source folder contains the <functionName> function?
T7	How many JS files do not invoke or are invoked by other JS file?
T8	Identify the folder that contains the highest number of JS files
T9	What functions call the function <functionName> defined in <path>?

Table 2: Experiment’s tasks

3.2 Research Questions and Hypotheses

Our experiment is designed to answer the following research questions (RQ):

- RQ#1 How does using Hunter affect the *user performance* of developers to complete software comprehension tasks?
  - RQ#1.1 How does using Hunter affect the *accuracy* of developers to complete software comprehension tasks?
  - RQ#1.2 How does using Hunter affect the *time* that developers need to complete software comprehension tasks?
  - RQ#1.3 Which panels of Hunter and VSC are most used to complete software comprehension tasks?
- RQ#2 How does using Hunter affect the *user experience* of developers?
  - RQ#2.1 What are the *emotions* that developers feel when using Hunter?
  - RQ#2.2 How *useful* developers consider Hunter?

To analyze user performance (RQ#1), we measured the accuracy of developers and their needed time to complete software comprehension tasks using Hunter and VSC. Also, we analyzed in which panels they focused their attention during the tasks. To analyze user experience (RQ#2), we collected impressions of the emotions and subjective scores of usability of developers who carried out software comprehension tasks using Hunter and VSC.

From the research questions, we formulate five null hypotheses (the alternative hypotheses follow analogously):

- $H1_0$: Developers perform equally to complete software comprehension tasks using Hunter and VSC.
- $H2_0$: Developers require the same time for completing tasks with Hunter and VSC.
- $H3_0$: Developers pay equal attention to all the panels of Hunter and VSC when completing tasks.
- $H4_0$: Developers have equally positive than negative emotions about Hunter.
- $H5_0$: Developers do not consider Hunter useful.

3.3 Participants

In total, sixteen experienced software engineers participated in our experiment (2 females). Their average age was 29 years old (std. dev. 4.7). Participants were from industry and they exhibited various levels of experience. To recruit participants, we send an invitation to an e-mail list of graduates from the university in which the experiment was conducted.

We defined four groups of participants, each group having four participants. The participants were assigned to one of the four groups by random assignment. Before the experiment started, we asked each participant to fill out a demographic questionnaire. Participants self-assessed their experience by using a Likert scale of five steps, *i.e.,* 1 (no experience) to 5 (expert). The average experience using JavaScript was 2.5 (G1), 2.5 (G2), 2.3 (G3), and 2.8 (G4) with std. dev. 0.5, 0.5, 0.43, and 1.09 respectively. Thus, it is fair to assume that expertise is almost equally distributed within the groups.

During the experiment execution, we alternate between groups that are first presented with Hunter and those that are first presented with VSC. For example, the tasks using Hunter are presented first for *Group 1 (G1)* and *Group 3 (G3)* while the tasks using VSC are first introduced to *Group 2 (G2)* and *Group 4 (G4)* (Table 1).

3.4 Procedure

The experiment consisted of two phases. First, we asked background information of participants and gave them a tutorial to train them in the use of Hunter. Secondly, in the experimental session participants were asked to solve a set of software comprehension tasks, and were asked of their emotions and impressions of usability.

Background and tutorial. Before starting the experiment, each participant signed a consent form that informed them about the characteristics of the study, and in which they explicitly agree to participate in the study. Then, each participant filled in a short questionnaire to collect demographic data such as their age, gender, and level of experience in software development. After these questions, a video tutorial of Hunter was shown to participants. Then, they had a training phase to familiarize themselves with Hunter and VSC. To this end, participants were provided with a "tutorial" application called Madge[16]. While Hunter and VSC were presented to the participants, they had the freedom of using other tools usually employed in development activities such as a terminal window.

Experimental session. During the interview, we recorded a video of the screen and the audio of the laptop used by participants. These recordings were used later for analyzing the reasoning process followed by participants, and also they were used to measure the time employed to complete the tasks. Each question was read by the experimenter. The answers of participants were registered using a Google form questionnaire. Participants were free to spend as much time as they felt necessary to complete the tasks. Also, they were informed that they could ask questions at any time during the session. The session ended with a post-questionnaire about the usability of Hunter. Finally, with the goal of measuring the user experience of using Hunter, participants were asked to approach a table on which we placed 270 paper labels. Each label described a word to represent an emotion. Positive emotions were placed on the left side of the table, and negative emotions on the right. We organized the labels into eight groups of positive and also eight of negative emotions. We asked participants to collect ten emotions, that they experienced when using Hunter, from the table and to sort them according to the intensity of the emotion. We observe that participants engaged with this method and were willing to spend a fair time to introspect into their emotions.

Along the session, we monitored the attention that participants paid to the different panels of Hunter and VSC. To this end, participants wore an eye-tracking device.

Oracle. Employing dedicated scripts, we programmatically extracted correct answers for each task. For example, to identify the file that has the most lines of code we used the combination of *find*, *grep*, *wc -l*, and other commands. We made sure that our scripts were correct by running them on Madge, the application used in the tutorial, and checking those results with manual inspection. Notice that to build the oracle we did not use VSC nor Hunter.[17]

4 RESULTS

Next, we present and discuss the results that we collected in our experiment. To organize the section, we revisit the research questions by grouping them into user performance and user experience.

4.1 User Performance

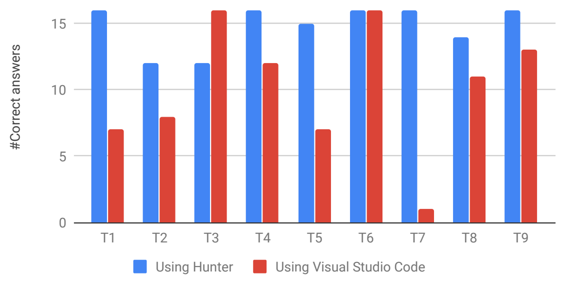
We analyzed user performance in terms of (*i*) *time* that participants needed to carry out software comprehension tasks, (*ii*) *accuracy* of their answers, and their overall (*iii*) *attention* when using our proposed tool.

To analyze statistically significant differences in the results, we used a two-tailed Student's t-test when the distribution is normal and a Mann-Whitney U-test when it is not. In both cases, we use a probability of error (or significance level) of $\alpha = 0.05$. To test whether the collected data follows a normal distribution, we used the Shapiro-Wilk test. Also, we ensure independence observations based on the design of the study.

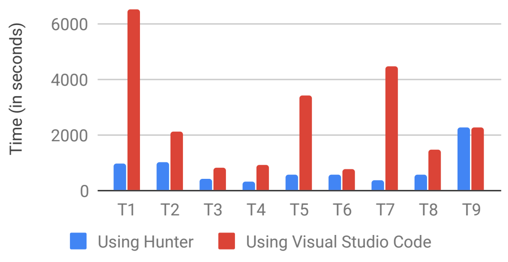
RQ#1.1: accuracy. We had sixteen participants, each completing eighteen tasks, nine with VSC and nine with Hunter. Thus, we collected 144 answers (16×9) related to Hunter and the same number of answers for VSC. Figure 6a shows the number of correct answers of participants for each task, using Hunter and VSC. As it is shown, the results of participants using Hunter are more accurate than when using VSC (in all tasks except T3 and T6). In fact, when using Hunter, at least 75% of the participants answered correctly all tasks. In the case of VSC, only four tasks (*i.e.,* T3, T4, T6, T9) were correctly answered by at least 75% of the participants. Interestingly, in some tasks (*e.g.,* T1, T5, and T7) Hunters greatly improves the accuracy of participants. Moreover, while in Hunter only one participant could not give an answer in one of the tasks, using VSC twenty-four (*i.e.,* $\sim$17%) tasks were not answered. The tasks that participants mostly struggled to answer were T1 ($\times$4), T5 ($\times$3), and T7 ($\times$11). These results could indicate that VSC provides users with little support for dealing with this kind of tasks, and therefore, programmers can benefit from complementing it with a visualization approach such as Hunter. In 79% of the tasks (=115) participants exclusively used

[16]https://github.com/pahen/madge

[17]The oracle containing the correct answers for each task can be found at http://bit.ly/2N4zZQu

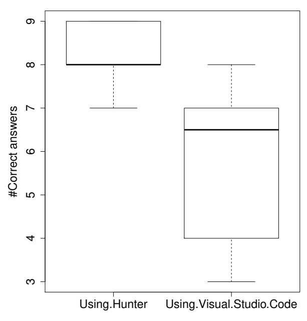


(a) The accuracy of participants by task and treatment

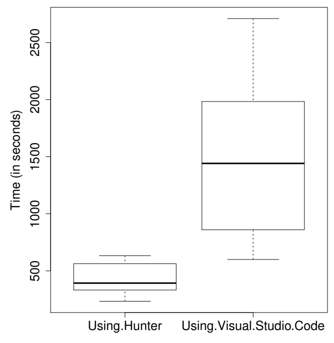


(b) Completion time by task and treatment

Figure 6: Comparison by task and treatment



(a) Correct answers by user



(b) Time spent by user

Figure 7: Participants' performance

VSC (without any other complementary tool). That is, only six participants complemented VSC with other tools in some of the tasks. Among these complementary tools, participants used a terminal window (×23), the Pharo[18] programming environment (×5), and a combination of VSC, Pharo, and the terminal (×1). Notice that when using Hunter, participants did not require complementary tools. We reflect that this could be due to a construction bias since participants knew that our goal was to evaluate Hunter against a well-known baseline, so they did not leave Hunter.

Since only two women participated in the experiment, we do not make a gender distinction in our results. However, we could not observe evident differences between men and women results.

[18]https://pharo.org

We also analyzed the number of correct answers for each participant when they used Hunter and VSC. The median of correct answers by participant is 8 and 6.5 (Figure 7a) when using Hunter and VSC, respectively. To validate $H1_0$ we tested the data for normality and concluded that the data deviates from normality (p-value = 7.102e-05). After running the Mann-Whitney test we were able to reject the null hypothesis with p-value = 2.307e-05. Thus, the ratio of correct answers using Hunter and VSC is significantly different.

In summary, we can answer RQ#1.1 by saying that participants solved software comprehension tasks more accurately when using Hunter than when using VSC.

RQ#1.2: completion time. Figure 6b shows the sum of the time spent by participants to solve each task. The figure shows that except for T9, participants spent more time to solve the tasks with VSC than with Hunter. Since we did not impose a time limit to solve the tasks, the gap is quite large in those tasks that several participants did not answer when using VSC (*i.e.,* T1, T5, and T7). Nevertheless, the difference in the time spent is still observable in tasks that were answered by all the participants such as T2 and T3. In these cases, when solving the tasks using Hunter participants spent half of the time than when using VSC.

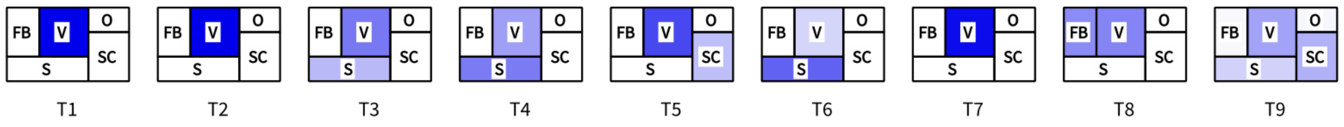
We also compared the aggregated time spent by each participant to solve the nine tasks using Hunter and VSC using its median value. As shown in Figure 7b, the median of the time that participants spent using Hunter is less than any measured time spent when using VSC (*i.e.,* 393 vs 1,386.5 seconds). To verify if this difference is significant, we first tested the data for normality. We observed that the data deviates from normality (p-value = 0.0001081). Then, we ran the Mann-Whitney test, and we were able to reject the null hypothesis $H2_0$ with p-value = 3.549e-06.

We can answer RQ#1.2 by saying that Hunter helps developers to complete software comprehension tasks faster than VSC.

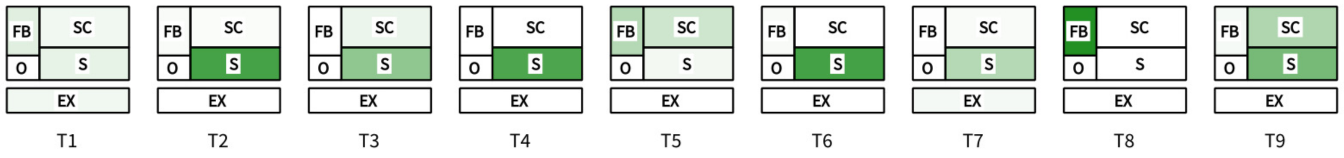
RQ#1.3: attention. Using the data obtained from an eye tracking device, we examined the attention of participants on each panel of Hunter and VSC graphical interfaces (see Figures 1 and 5). Specifically, for each task, we measured the median time that participants spent on each panel. Figure 8 uses glyphs to represent the attention of participants on the panels of Hunter (above) and VSC (below). Figure 8 shows the median of the percentage of time that participants spent on each panel when using Hunter and VSC. In the case of VSC, the *EX* panel indicates the time developers spent using complementary tools such as Pharo and terminal window.

As it is shown in Figure 8a, in the case of Hunter, developers paid great attention to the File Dependencies View panel (V) in all the tasks. Also, the Search Box panel was used with (V) in four of the tasks. In the case of VSC (Figure 8b) we do not detect any panel that was used in all the tasks. However, the Search Box panel (S) was used in seven tasks.

We now analyze whether there is a statistically significant difference in the attention paid by the developers to the panels of Hunter and VSC when completing software comprehension tasks ($H3_0$). Specifically, we employ the Kruskal-Wallis non-parametric



(a) Hunter



(b) VSC

Figure 8: Heatmap Eye Tracking. The glyphs represents the physical panel layout of Hunter (Figure 1) and VSC (Figure 5). For VSC, the EX panel represents the use of complementary tools to solve the task (*e.g.,* terminal, Pharo).

test with a probability of error of $\alpha = 0.05$. For both Hunter and VSC, we obtained p-values < 0.05 for all tasks. It means that for each task there is a significant difference in at least two panels.[19] We also conducted Pairwise Wilcoxon Rank Sum Tests post-hoc comparisons to determine which pairs of panels have significant time differences in the attention paid by developers. The post-hoc tests revealed that, in the case of Hunter, there is a significant statistical difference to claim that the panel V is the most used in task T1, T2, T3, T5, and T7. Whereas, in tasks T4, T8, and T9, V is the most used panel together with other panels (panel S in T4, FB in T8, and S and SC in T9). Differently, in the case of VSC, there are no significant differences between the panels that caught the attention of participants (except for panel O, which was rarely used). Specifically, in tasks T1, T2, T5, and T7, there are no significant differences between the attention paid to panels FB, SC, S, and EX. In the case of T3, T4, and T6, panel S is the most used one (whereas in T3 there is not a statically difference with SC).

Interestingly, we noticed during the analysis of eye-tracking data that when using VSC, participants tended to "jump" more between panels to solve a task than when using Hunter. Specifically, this was more noticeable in T1, T5, and T7 (using FB, SC, S, and EX), and in T9 (using SC and S). For the case of Hunter, this behavior was only detected in T9 (using V, S, and SC).

In summary, these results indicate that users need to analyze fewer panels in Hunter than VSC, exhibiting a higher efficiency.

We can answer RQ#1.3 by saying that the *File Dependencies View* (V) is the most used pane in Hunter, while in VSC developers used multiple panels to solve tasks.

4.2 User Experience

We analyzed user experience in terms of the *emotions* and the *usability* perceived by participants in the study.

RQ#2.1: emotions. In Figure 9, we present a summary of the emotions experienced by the participants of the user study. The chart illustrates frequent emotions that were reported at least by two participants. Positive emotions are represented with blue bars and negative ones with red bars. The chart shows that participants experienced mostly positive emotions (and only a few negative).

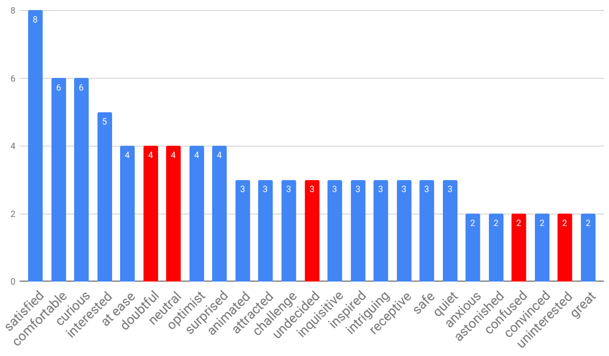


Figure 9: A summary of emotions experienced by participants in the user study.

To analyze these emotions, as a first approach, we applied a *Plain Algebraic Sum* (PAS) of the number of positive and negative emotions. However, we noticed that, in general, emotions are not entirely positive or negative. For instance, a participant who reports feeling *safe* might have a positive experience of being in control, but at the same time, might have a negative experience of lacking excitement. To deal with this fact, we used the *SentiWordNet* catalog, a lexical resource in which contextual emotions are associated with positive and negative scores [10]. Consequently, in the *Plain SentiWordNet Sum* (PSS), we used the scores specified in the SentiWordNet catalog as weights to calculate the algebraic sum of the number of emotions. However, as a result, we observed the differences among emotions that were noticeable before when using the plain algebraic sum, now were barely distinguished.

Since we also asked participants to specify the intensity of their emotions by sorting them in a ranked list, we applied the *Weighted Algebraic Sum* (WAS) that was used in a previous study [23]. In

[19]The results of the statistical tests for each task can be found at http://bit.ly/2KwXH60

it, the intensity of emotions is used as a weight in the algebraic sum to score the overall emotions. The emotion score metric is the sum of the top ten emotions (ranked by intensity) weighted by their type (positive or negative). We then inspired on WAS but used as weights the scores of each emotion in the SentiWordNet catalog. Along this line, we defined a new metric, that we called *Weighted SentiWordNet Sum* (WSS). WSS does not only include a more fine-grained characterization of emotions by using the score of SentiWordNet, but also considers their intensity. The formula to calculate the WSS metric is shown in Equation 1.

$$score = \sum_{i=1}^{10} i \times pos(emotion_i) \times neg(emotion_i) \qquad (1)$$
$$pos(emotion), neg(emotion) \text{ in SentiWordNet}$$

We analyzed the emotions that participants felt during the user study using four metrics in total that are shown in Figure 10. The uniform shape of the red line that represents the PAS metric impedes identifying differences among the emotions of participants. Instead, we observe that our proposed WSS metric can help analyze emotions by taking into account two important aspects: positive and negative aspects of emotions, and their intensity. In summary, most participants felt positive emotions such

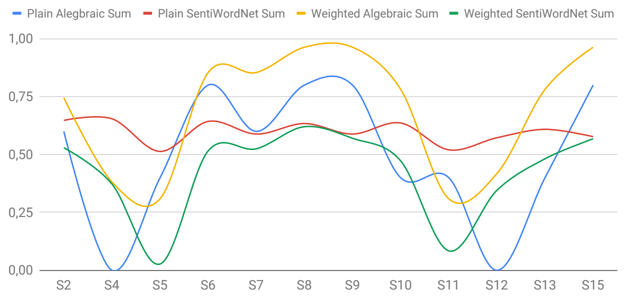


Figure 10: A line chart of the four metrics used to analyze the emotions experienced by participants in the user study.

as satisfaction, comfort, and curiosity. On the other hand, two participants reported several negative emotions. We conjecture these emotions arise from a lack of affinity to exploratory tasks that made them feel doubtful, undecided, and sometimes confused.

We can answer RQ#2.1 by saying that most participants who used Hunter felt mostly positive emotions that contributed to good overall user experience.

RQ#2.2: usability. To collect data for the analysis of the usability of the system, we asked participants to specify their level of agreement to the System Usability Scale (SUS) questionnaire [6] statements listed in Table 3. The SUS questionnaire is a reliable tool for measuring usability. To assess each statement participants used a 7-step Likert scale (1 means completely disagree and 7 means completely agree). The results are summarized in Figure 11. To analyze the results, we grouped the statements into four categories: (*i*) *design* –participants perceived that the style and features were appropriate to accomplish comprehension tasks as well as that they considered easy to browse data, however, in a few cases they considered that the responsiveness of the user interface could be improved; (*ii*) *information* – participants considered that the information provided by the tool is sufficient, suitable, trustable, and in an appropriate format, but for a few cases not very accurate; (*iii*) *quality* –participants observed that implemented features are robust and that since they trust in the information provided by the tool they feel more certain, however, they missed features to obtain details-on-demand; and (*iv*) *immersion* –the tool is perceived to promote curiosity and suitable for exploration, however, it seems to offer a moderate degree of immersion.

We can answer RQ#2.2 by saying that most participants perceived that the tool offers a high degree of overall usability. Though, a few participants considered that the interface responsiveness, the accuracy of information, and a lack of details-on-demand represent opportunities for improvement.

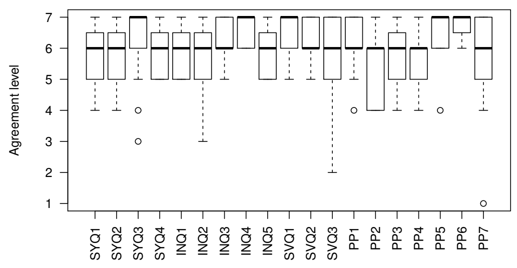


Figure 11: Results of the 7-step Likert scale questionnaire administered to the participants in the study.

Theme	Id	Question
Design	SYQ1	The design style is appropriate
	SYQ2	Browsing data is easy
	SYQ3	The user interface is highly responsive
	SYQ4	The included features are appropriate
Information	INQ1	The content is sufficient for the required information
	INQ2	The information is accurate
	INQ3	The information is suitable
	INQ4	The information is trustable
	INQ5	The information has an appropriate format
Quality	SVQ1	The implemented features are robust
	SVQ2	*Hunter* is trustable and reduces uncertainty
	SVQ3	*Hunter* provides users details-on-demand
Immersion	PP1	I lost track of time while using *Hunter*
	PP2	I do not perceive noises while using *Hunter*
	PP3	I enjoy solving tasks while using *Hunter*
	PP4	I have fun solving tasks while using *Hunter*
	PP5	*Hunter* promotes my curiosity
	PP6	*Hunter* helps with exploratory tasks
	PP7	*Hunter* boosts my imagination

Table 3: Usability questionnaire.

5 THREATS TO VALIDITY

We grouped the threats to the validity of our study into four topics: applications, participants, tasks, and tools.

Applications. the sizes and domains of the applications used could influence the results of the experiment. We mitigated this threat by using applications of different sizes and domains, and by splitting the participants into four groups that used different treatments.

Participants. the expertise of participants might not be representative of a real-world sample of professional software developers. To mitigate this threat, we collected in the background questionnaire data to assess their experience using JavaScript and IDEs. Another threat is that the experience of the participants could not have been equally distributed across the groups. To mitigate this threat, all the participants were assigned randomly to each group. We also checked that the expertise across groups was balanced.

Tasks. we found that tasks T1, T5, and T7, were difficult to be carried out with VSC. This could indicate that (*i*) the tasks chosen for the experiment could be biased towards one of the tools, or (*ii*) VSC is indeed not adequate to solve these tasks. To mitigate this threat, we chose tasks that usually arise during the software comprehension process. Furthermore, we checked that all the tasks were possible to be answered using either tool.

Tools. the degree of familiarity with the tools by participants could represent a threat to the validity of the results. We mitigated this threat by providing all participants the same tutorial, and by allowing participants to continue using the tools until they felt confident with both tools. Another threat could be that participants had an affinity to Hunter based on our involvement in the construction of the tool. Also, how dependencies are computed in Hunter could be a possible threat. Specifically, some dependencies could be unresolved since they are created dynamically (e.g. `require(base + '../lib/extend/deployer')`). We manually analyze the source code of our case studies for this kind of cases and the number of them is negligible.

6 RELATED WORK

In this section, we highlight some of the most relevant related work. In it, we discuss the differences of our work with previous studies in terms of visualization techniques, problem domains, and conducted evaluations, as it is a central aspect in our work. We restrict the coverage of the discussed related work to approaches based on static analysis.

Software visualization for comprehension. Software visualization using multiple techniques and metaphors are commonly proposed to support software comprehension tasks [20, 26, 33, 35]. For example, representing a software as a city has gained attention [24, 32, 36], due to the inherent intuition one can have about a familiar environment. Software is naturally multi-concern as it "involves a variety of activities carried out with a number of tools, components and environment, that relate to many different aspects of a system" [7]. One way to comprehend multi-concern aspects is to explore dependencies between components [29]. Treemap is a 2D visual layout designed to represent hierarchical structures (as a software system often complies with). Voronoi treemaps [3] and tree visualization [2] are proven techniques that adequately support comprehension. Although Hunter does not introduce a novel visualization technique, it adequately combines proven visualization techniques in coordinated views that packaged in a tool enable their evaluation. Moreover, Hunter supports the visualization of JavaScript applications, which despite its popularity it is rarely supported by proposed software visualizations.

Software visualization evaluation. Visual environments to support software comprehension tasks are notoriously difficult to evaluate [25, 30]. Typically, software visualization approaches are evaluated through usage scenarios [1, 11, 27, 28, 34]. That is, a demonstration by the authors of visualizations to exemplify their benefits. Usage scenarios can be helpful to identify and discuss the strengths of a visualization approach, however, they should be considered only a first step towards validating the benefits of a visualization approach. In fact, only a few software visualizations been evaluated via thorough user studies [4, 9, 14, 16, 37]. Among them, most focus on completion time and correctness. We think that the effect of a visualization tool for software comprehension in human cognition requires to involve other variables. Therefore, in our evaluation, we decided to examine attention and emotion.

7 CONCLUSION AND FUTURE WORK

In this paper, we presented Hunter, a visualization approach that supports developers on software comprehension tasks to understand JavaScript applications.

To assess the benefits of Hunter, we conducted a thorough controlled experiment driven by five research questions. In total, 16 software developers participated in our experiment with a proper background and experience. We ask each participant to solve nine tasks using Visual Studio Code and using Hunter. We found that when using Hunter, developers can increase their user performance in terms of their accuracy to solve software comprehension tasks (RQ#1.1), and the time that they need to perform such tasks (RQ#1.2). In particular, we found that from nine tasks, the median of the correctness of developers' answers when using Hunter was eight against only six and a half when using Visual Studio Code. Regarding the completion time, we found that the median of the time spent for participants using Hunter is 393 seconds against 1,386 of Visual Studio Code. Moreover, we found that participants needed to use fewer panels in Hunter than in Visual Studio Code to solve a task (RQ#1.3). In the case of user experience, we found that developers feel, in general, positive emotion when using Hunter (RQ#2.1) and that they think that the tool is useful (RQ#2.2).

As future work, we plan to integrate Hunter's visualizations in an Integrated Development Environment (IDE), such as Visual Studio Code. Also, we plan to conduct a long case study in the wild with developers using Hunter on a daily basis, integrated into their development environment to analyze their projects.

ACKNOWLEDGMENTS

This research is partially supported by STICAmSud project 18STIC-02. Merino acknowledges funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 251654672 – TRR 161. Bergel is very grateful to Lam Research and Fondecyt Regular 1200067 for partially sponsoring this work. Dias acknowledges Conicyt-Fondecyt project 3180386.

REFERENCES

[1] Andrea Adamoli and Matthias Hauswirth. 2010. Trevis: A context tree visualization & analysis framework and its use for classifying performance failure reports. In *Proceedings of the 5th international symposium on Software visualization*. ACM, 73–82.

[2] Ivan Bacher, Brian Mac Namee, and John D. Kelleher. 2016. On Using Tree Visualisation Techniques to Support Source Code Comprehension. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*. 91–95. https://doi.org/10.1109/VISSOFT.2016.8

[3] Michael Balzer, Oliver Deussen, and Claus Lewerentz. 2005. Voronoi treemaps for the visualization of software metrics. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*. ACM, New York, NY, USA, 165–172. https://doi.org/10.1145/1056018.1056041

[4] Titus Barik, Kevin Lubick, Samuel Christie, and Emerson Murphy-Hill. 2014. How developers visualize compiler messages: A foundational approach to notification construction. In *2014 Second IEEE Working Conference on Software Visualization*. IEEE, 87–96.

[5] Alexander Bergmayr, Hugo Bruneliere, Jordi Cabot, Jokin García, Tanja Mayerhofer, and Manuel Wimmer. 2016. fREX: fUML-based reverse engineering of executable behavior for software dynamic analysis. In *Proceedings of the 8th International Workshop on Modeling in Software Engineering*. ACM, 20–26.

[6] John Brooke. 1996. *"SUS-A quick and dirty usability scale." Usability evaluation in industry*. CRC Press. https://www.crcpress.com/product/isbn/9780748404605

[7] Tommaso dal Sasso, Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. Blended, Not Stirred: Multi-concern Visualization of Large Software Systems. In *Proceedings of VISSOFT 2015 (3rd IEEE Working Conference on Software Visualization)*. 106–115. https://doi.org/10.1109/VISSOFT.2015.7332420

[8] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. 1993. Visualizing the Behavior of Object-Oriented Systems. In *Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*. 326–337. https://doi.org/10.1145/165854.165919

[9] Niklas Elmqvist and Philippas Tsigas. 2003. Growing squares: Animated visualization of causal relations. In *Proceedings of the 2003 ACM symposium on Software visualization*. ACM, 17–ff.

[10] Andrea Esuli and Fabrizio Sebastiani. 2006. Sentiwordnet: A publicly available lexical resource for opinion mining.. In *LREC*, Vol. 6. Citeseer, 417–422.

[11] Michael D Feist, Eddie Antonio Santos, Ian Watts, and Abram Hindle. 2016. Visualizing project evolution through abstract syntax tree analysis. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 11–20.

[12] Alison Fernandez and Alexandre Bergel. 2018. A domain-specific language to visualize software evolution. *Information and Software Technology* 98 (2018), 118–130. https://doi.org/10.1016/j.infsof.2018.01.005

[13] Günter Fleck, Wilhelm Kirchmayr, Michael Moser, Ludwig Nocke, Josef Pichler, Rudolf Tober, and Michael Witlatschil. 2016. Experience Report on Building ASTM Based Tools for Multi-language Reverse Engineering. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 683–687. https://doi.org/10.1109/SANER.2016.33

[14] Carlos Gouveia, José Campos, and Rui Abreu. 2013. Using HTML5 visualizations in software fault localization. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 1–10.

[15] Lile Hattori, Marco D'Ambros, Michele Lanza, and Mircea Lungu. 2013. Answering software evolution questions: An empirical evaluation. *Information and Software Technology* 55, 4 (jan 2013), 755 – 775. https://doi.org/10.1016/j.infsof.2012.09.001

[16] Taimur Khan, Henning Barthel, Achim Ebert, and Peter Liggesmeyer. 2015. Visual analytics of software structure and metrics. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*. IEEE, 16–25.

[17] Holger M Kienle and Hausi A Müller. 2010. The tools perspective on software reverse engineering: requirements, construction, and evaluation. In *Advances in Computers*. Vol. 79. Elsevier, 189–290.

[18] Claus Klammer and Josef Pichler. 2014. Towards tool support for analyzing legacy systems in technical domains. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 371–374. https://doi.org/10.1109/CSMR-WCRE.2014.6747197

[19] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. 2018. The Road to Live Programming: Insights From the Practice. In *Proceedings of the 40th ACM/IEEE International Conference on Software Engineering (ICSE '18)*.

[20] Adrian Kuhn, David Erni, and Oscar Nierstrasz. 2010. Embedding spatial software visualization in the IDE: an exploratory study. In *Proceedings of the 5th international symposium on Software visualization*. ACM, 113–122.

[21] G. Langelier, H. Sahraoui, and P. Poulin. 2008. Exploring the evolution of software quality with animated visualization. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. 13–20. https://doi.org/10.1109/VLHCC.2008.4639052

[22] Michele Lanza. 2001. The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques. In *Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution)*. 37–42. https://doi.org/10.1145/602461.602467

[23] Leonel Merino, Johannes Fuchs, Michael Blumenschein, Craig Anslow, Mohammad Ghafari, Oscar Nierstrasz, Michael Behrisch, and Daniel Keim. 2017. On the impact of the medium in the effectiveness of 3D software visualization. In *Proc. of VISSOFT*. IEEE, 11–21.

[24] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. 2017. CityVR: Gameful Software Visualization. In *ICSME'17: Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (TD Track)*. IEEE, 633–637. https://doi.org/10.1109/ICSME.2017.70

[25] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. 2018. A Systematic Literature Review of Software Visualization Evaluation. *Journal of Systems and Software* 144 (2018), 165–180.

[26] Leonel Merino, Mircea Lungu, and Oscar Nierstrasz. 2014. Explora: Infrastructure for Scaling Up Software Visualisation to Corpora.. In *SATToSE*. 25–36.

[27] Leonel Merino, Mircea Lungu, and Oscar Nierstrasz. 2015. Explora: A visualisation tool for metric analysis of software corpora. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*. IEEE, 195–199.

[28] Katsuya Ogami, Raula Gaikovina Kula, Hideaki Hata, Takashi Ishio, and Kenichi Matsumoto. 2017. Using high-rising cities to visualize performance in real-time. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 33–42.

[29] Doreen Seider, Andreas Schreiber, Tobias Marquardt, and Marlene BrÃijggemann. 2016. Visualizing Modules and Dependencies of OSGi-Based Applications. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*. 96–100. https://doi.org/10.1109/VISSOFT.2016.20

[30] Bonita Sharif, Grace Jetty, Jairo Aponte, and Esteban Parra. 2013. An empirical study assessing the effect of SeeIT 3D on comprehension. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 1–10.

[31] Jonathan Sillito, Gail C Murphy, and Kris De Volder. 2006. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 23–34.

[32] Frank Steinbrückner and Claus Lewerentz. 2013. Understanding software evolution with software cities. *Information Visualization* 12, 2 (April 2013), 200–216. https://doi.org/10.1177/1473871612438785

[33] Alexandru Telea and Lucian Voinea. 2008. An interactive reverse engineering environment for large-scale C++ code. In *Proceedings of the 4th ACM symposium on Software visualization*. ACM, 67–76.

[34] Simon Urli, Alexandre Bergel, Mireille Blay-Fornarino, Philippe Collet, and Sébastien Mosser. 2015. A visual support for decomposing complex feature models. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*. IEEE, 76–85.

[35] Bradley Wehrwein. 2013. Lightweight software reverse engineering using augmented matrix visualizations. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 1–4.

[36] Richard Wettel and Michele Lanza. 2007. Program Comprehension through Software Habitability. In *Proceedings of ICPC 2007 (15th International Conference on Program Comprehension)*. IEEE CS Press, 231–240.

[37] Richard Wettel, Michele Lanza, and Romain Robbes. 2011. Software systems as cities: a controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 551–560. https://doi.org/10.1145/1985793.1985868

[38] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Bjöorn Regnell. 2012. *Experimentation in Software Engineering*. Springer. I–XXIII, 1–236 pages.