The Use of Eye Tracking in Software Development

Bonita Sharif(✉) and Timothy Shaffer

Youngstown State University, Youngstown, OH 44555, USA
bsharif@ysu.edu, trshaffer@student.ysu.edu

Abstract. Eye trackers have been routinely used in psychology reading experiments and in website usability studies for many years. However, it is only recently that they have been used by more researchers in the software engineering community. In this paper, we categorize two broad areas in which eye tracking technology can benefit software development in a practical way. The first area includes using the eye tracker as an assessment tool for software artifacts, tools, and techniques. The second area deals with using eye tracking data from developers to inform certain software tools and software development tasks such as providing developer recommendations and software traceability tasks. Examples of experiments and studies done in each of these broad areas is presented and discussed along with future work. The results point towards many benefits that eye trackers provide to augment the daily lives of programmers during software development.

Keywords: Eye tracking · Software development · Software traceability · Assessing software artifacts · Program comprehension

1 Introduction

Software development is an integral part of software engineering. Developing software is an intertwined process that involves multiple tasks such as program comprehension, concept/feature location [1], impact analysis, writing code, and testing to match requirements. The above tasks certainly do not encompass all the activities but the core ones that a software developer is faced with on a daily basis. In order to make the task of a software developer a little easier, there has been a lot of work done in trying to understand how a developer interacts with his or her environment. Several researchers have conducted studies, such as the one by Ko et al. [2], to determine the types of questions software developers ask and the knowledge they seek to find while they work on their daily tasks. The end goal is to develop tools to help the developer in the highly complex and intricate skill of software development. The methods used to collect such information are mainly surveys, questionnaires, interviewing, and think-aloud methods.

In this paper, we posit that besides the above methods, a new method of data collection using an eye tracker provides additional benefits to the already existing methods in use. Besides using eye trackers for assessment – which seems to be the most obvious way to use them – they can also be used to inform various software tasks such as providing recommendations based on what the developers should look at next

D.D. Schmorrow and C.M. Fidopiastis (Eds.): AC 2015, LNAI 9183, pp. 807–816, 2015.
DOI: 10.1007/978-3-319-20816-9_77

based on what they have just been looking at. Past data of an eye tracking session can also be used to determine if and when the developer is facing difficulty. There has been one study [3] that tried to determine task difficulty based on psycho-physiological measures (including eye tracking) and the results seem to be quite promising. These directions in using eye tracking data as part of algorithms to inform other software tasks have only recently been explored by a few researchers [3–5]. Seigmund et al. [6] (although not related to eye tracking) recently used functional magnetic resonance imaging to understand how developers read source code. We expect more work to be conducted using biometrics including eye tracking in the future as more software engineering researchers tackle this important human aspect of software engineering.

One viable alternative to eye tracking sessions is developer-session recording models such as Mylyn. Mylyn is an Eclipse plugin that records developer interactions such as mouse clicks, selection and searching within the source code while the developer works on tasks. The main difference between the Mylyn and eye tracking approach is that with Mylyn, one has to click on a source code entity such as an "if statement" or a method call for it to register as something that was selected. With eye tracking, we do not need the developer to click on a method call or a method signature to know that they looked at it. In other words, eye tracking provides much more fine-grained data into what a developer is actually looking at.

We believe eye tracking technology can help software developers in their daily tasks. The goal is to reduce the effort developers have to put in. One instance would be to reduce effort to search by find relevant places in the code to begin their change task solely based on past behavior of eye movements. This approach is very developer centric. The cost of eye tracking equipment has also reduced by many factors of magnitude compared to a few years ago, making this a reality in the near future. The eye trackers we refer to are the remote eye trackers where a person is not required to wear anything, making it more attractive to be adopted by a developer.

In this position paper, we put forth the idea that eye tracking has potential in future software development tasks in software engineering by first outlining the need for eye-aware integrated development environments to make these claims a reality. Next, we provide examples of eye tracking being used in the software engineering realm. We close with the current state of eye trackers, challenges, and a call for more work in this area.

2 Eye-Aware Integrated Development Environments

In order for eye tracking to be adopted in the software engineering community and in the daily life of a developer, we first need to have the working environment of developers support eye gaze recording. Most developers work within an integrated development environment (IDE) such as Eclipse or Visual Studio. For eye tracking to be most useful, the feature of tracking developers' eyes while they work should be available to them from within the IDE itself or else it is less likely to be adopted or used.

2.1 The Problem

The problem with eye tracking so far was that most studies as discussed in the following sections were conducted with small snippets of code that fit on one screen. Every study that was done in software engineering (including some of our own), made use of small code snippets that fit on one screen. These studies mainly used eye tracking for assessing tools and techniques. The nature of the eye tracker is such that it gives you an *(x,y)* pixel coordinate on the screen of where a person is looking at. That *(x,y)* pixel coordinate is not mapped intrinsically to what the stimulus under the coordinate is. In other words, the eye tracker does not know if you scrolled in Eclipse. This is a big problem because when one scrolls, a different element will now appear under the pixel coordinates the person is looking at. So scrolling is not intrinsically captured and the underlying document you are scrolling is not mapped to the pixel coordinate. All the eye tracker will record is the *(x,y)* pixel coordinate. If a source code file is 300 lines long and the user scrolled many times through the file, there is no way for the eye tracker to tell which line in the document (source code) scrolled was under which *(x,y)* pixel coordinate in time without doing some kind of video analysis of the eye tracking session (which we did not want to do as it was not scalable).

This lack of context awareness between the eye tracker and the stimulus needed to be addressed due to the following reasons. First, developers do not just read small snippets of code in a realistic setting. They are more likely to work with an entire system with several hundred classes and long methods, so scrolling support for eye tracking with proper context is extremely important. Second, developers work with many files at a time for even a single change task. This means the eye tracker needs to know when the developer was looking at different source code files or bug reports for instance, as developers might move from reading a text file to writing source code, all of which should be captured by the eye tracking system. None of these features were available with current IDEs.

2.2 The Solution

We recognized this as a problem and decided that we had to find a better way to make eye tracking part of a developers work environment. This is when *iTrace* was born. Since there was no IDE that was eye aware (for scrolling support), we decided to build our own plugin to make eye tracking implicit within the IDE. This breaks the barrier of having small toy code snippets in future studies. It enables us to truly understand what developers are looking at while they develop software in a realistic setting. Currently, we are working on a stable version of *iTrace* and a formal tool demonstration along with an update installation package. Once this is created, it will be available in the open source domain for researchers to use as well as contribute to. We envision that *iTrace* will make an experimenter's job a lot easier since a lot of the groundwork on collecting the data and mapping the data to appropriate code elements is already taken care of by the tool. We believe *iTrace* is a huge step forward in making eye tracking accessible to software engineers as well as experimenters. See Fig. 1 for a snapshot of *iTrace*. It is written in Java and currently supports Java projects.

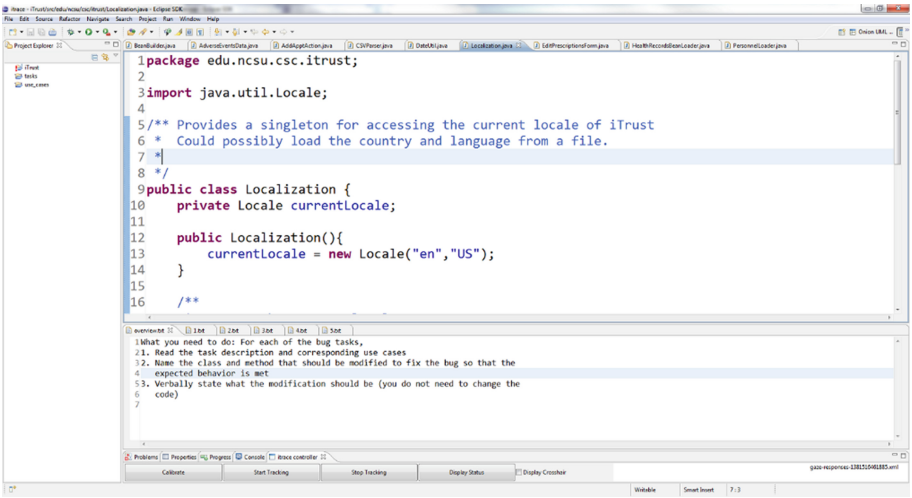

Fig. 1. A snapshot of iTrace

In the figure, we see the Eclipse window with the *iTrace* perspective open. At the very bottom is the *iTrace* dashboard (horizontal set of buttons) to perform a calibration, start tracking, stop tracking, and export sessions. Before an eye tracking session, it is essential that the eye tracker is first calibrated for each individual. During calibration a sequence of nine dots appear on the screen and the user is instructed to look at them. After the calibration is successful, the start tracking button can be activated. From then on, whatever the developer looks at is recorded and mapped to the corresponding source element (if they are looking at source code) until they click Stop tracking. For example, if the developer looked at the Locale object created in the Localization constructor, this gaze would be mapped to that new object created on line 13 in the figure. All context is also maintained i.e., it would also record that the new object was inside the constructor which was inside the class Localization. All of this data is exported in both XML and JSON (user configurable). The amount of data exported per second depends on the frequency of samples taken by the eye tracker. We have successfully used *iTrace* with 30 fps and 60 fps eye trackers for extended periods of time up to 50 or 60 min while collecting data on studies involving real change tasks.

iTrace currently works with the Tobii series eye trackers but supports any other eye tracker as it was designed to be extensible. Recently, we also developed an interface for the EyeTribe Mirametrix S2 eye tracker. In order to use another eye tracker, one needs to write an interface to communicate with the eye tracker and *iTrace*.

In a nutshell, *iTrace* provides a seamless way to collect eye tracking data from inside an IDE while keeping the context even when scrolling. The detailed architecture and design decisions of *iTrace* are out of scope of the current paper and have been left as a future formal tool description and demonstration.

3 Eye Tracking as Assessment

This section describes how eye tracking has been used as an assessment tool in software engineering studies. This is the first broad area in which eye tracking directly benefits daily software engineering activities, tools, and processes by studying if they are useful and how to make them better by directly studying developer eye gaze while they are at work doing the activity, or using the tool or following the process.

Crosby and Stelovksy explored the way subjects viewed algorithms for comprehension and discovered that the eye movements of experts and novices differ in the way they looked at English and Pascal versions of an algorithm [7]. They found that experts spend more time viewing complex statements. They also discovered that while both groups devoted time to viewing comments, novices spent significantly more time doing so.

Bednarik et al. investigated the visual attention of experts versus novices in debugging code [8]. Subjects used a debugger with multiple views to track down one or more defects in each program. No significant correlation was found between patterns of eye movement and performance in using the debugger. In a separate study, Bednarik et al. [9] also gave subjects a choice between viewing code and viewing an alternative representation of programs and asked whether the role of program representation differs over the course of time. They found that many experts choose to look at code ahead of an alternative representation. Subsequently, they studied the eye gaze of pair programmers [10].

Uwano et al. [11] also studied eye gaze patterns of five individuals while they were detecting defects in source code. They also studied eye movements of subjects who looked for inconsistencies in comparing requirements/design documents to source code [12]. The study in [11] was later replicated by Sharif et al. to give similar results [13]. Sharif et al. [14, 15] study the impact of identifier style (i.e., camel case or underscore) on code reading and comprehension using an eye-tracker. They find camel case to be an overall better choice for comprehension. Sharafi et al. [16] replicated the identifier study but tried to determine if gender played a role in identifier style selection for source code reading. They found that males and females follow different comprehension methods. For example: females tend to thoroughly check all answers to make sure they have selected the correct one whereas males do not. This was an interesting finding and nice extension to the studies done in [14, 15].

Yusuf et al. [17] conducted a study to determine if different class diagram layouts with stereotype information help in solving certain tasks. Guehéneuc [18] investigated the comprehension of UML class diagrams. Jeanmart et al. [19] conducted a study on the effect of the Visitor design pattern on comprehension using an eye tracker.

Sharif et al. also conducted several eye tracking studies assessing the role layouts have in the comprehension of design pattern roles [20], assessing a 3D tool [21] and, assessing the comprehension of C ++ and Python programs [22]. A statement advocating the use of eye tracking in assessing software visualizations is given in [23].

Recently, Busjahn et al. [24] used eye tracking in the field of computing education to assess how experts read and understand code. Another study [25] by the same group by authors extends the work and looks in the linearity of natural language text reading

and how this linear order is not followed when reading source code. They look at both experts and novices and show that novices also exhibit a less linear order when they read source code compared to natural language text. They also found that experts tend to be much more non-linear than novices.

4 Eye Tracking as Informing Software Tasks

In the previous section, we presented and highlighted a few studies that use eye tracking as a means to assess if the software tool, process, or technique is useful. We have seen examples of studies comparing different layouts in UML diagrams, different identifier styles, and different programming languages to name a few. The bulk of work done in the field of eye tracking in software engineering is in this first area of assessment as it is the most intuitive and reasonable one to perform especially on visual artifacts such as diagrams and well as source code.

The second broad category in software engineering that can benefit from eye tracking is to make eye gazes directly inform the software tools and tasks developers are working on. Decisions can be based not only on the artifacts in question but also on what and how long the developer looks at these artifacts. Note that we are not implying using eye gaze as input to interact with the environment. This is a different concept that is applicable to natural user interfaces (using gestures, voice, and eye gaze to issue commands) and one that is not touched on in this paper.

We describe a few studies in this paper that touch on the aspect of using eye tracking as a means of informing and improving existing software tasks.

Fritz et al. [3] used eye movements along with EEG signals to determine task (reading source code) difficulty in professional programmers adding additional evidence that eye tracking is indeed a viable and useful source of information. They were able to use the pupil dilation as one of the features to predict if a developer found the task to be difficult. Here eye tracking was used in a unique way to help determine if a programmer should possibly take a break if they are finding the task difficult.

Ali et al. [26] used eye tracking in the field of software traceability. They first asked a few developers to read small code snippets and noted where the developers looked at most to understand them. They found that developer look mainly at method names and not class names. They used this information to change the weighting scheme of a software traceability link retrieval algorithm that emphasizes methods more as this is what developers look at more. Here eye tracking was not used to assess an artifact, rather it was used as evidence to modify a weighting scheme in information retrieval methods such as Latent Semantic Indexing (LSI) to get better results.

Rodeghero et al. [5] took an approach similar to [26] and also used eye tracking on small snippets on code to determine what developers are looking at. In their study the task was to summarize about 60 methods from open source Java systems. They then used this information on what developers look at to build better summaries for the methods. Again this approach uses eye gaze to inform a software engineering task namely, method summarization.

In the field of software traceability, Sharif et al. propose to use eye gaze as input to generate traceability links for change tasks [27]. They first proposed this idea of

moving towards an eye aware IDE in [28]. In 2014, they conducted a pilot study [4] to determine if it was feasible to generate traceability links from eye gaze alone. The results they found were very promising and eye gaze does indeed seem to work really well to find links between relevant code entities and the task at hand. They developed an algorithm to find relevant entities using a weighting scheme based on time. This helps weed out entities that are looked at initially but later abandoned.

There is still a lot more work to be done in the area of having eye tracking inform more software engineering tasks.

5 Discussion and Future Work

Eye trackers have become more accessible to researchers who are using them to gain additional insights into software development activities. Modern eye trackers implicitly collect developers' eye gaze data on the visual display (stimulus) in an unobtrusive way while they are performing a given task. This eye movement data could provide much valuable insight as to how and why subjects arrive at a certain solution. We believe these measures can add a new additional dimension in supporting software engineering tasks. In March 2012, Tobii received $21 M from Intel to continue R&D investments in making eye-tracking mainstream in personal computers. It is not farfetched to say that we will eventually find an eye tracker inbuilt in every PC in the near future.

The goal is to use an eye tracker that is suitable for the task at hand and the accuracy needed. Not every study is in need of a high-end eye tracker. There are many options to choose from. The main vendors are Tobii, Facelab, SmartEye, and SMI but there have been many more emerging cheaper alternatives on the market in the past couple of years such as the EyeTribe and Mirametrix trackers among others. Tobii also recently put out a less than $100 eye tracker (Tobii EyeX) in 2014 that works with USB 3.0 ports.

Imagine a scenario where a developer would turn eye tracking on with the click of a button within their work environment, just before they start to work on fixing a bug. Then, they go about working in their environment and fix the bug. When they are done with the bug fix, they click another button to stop the eye data being recorded. The amount of data collected for this one change task i.e., fixing a bug is useful for a variety of tasks (summarization, recommendations to name a few). Later, at some point he or she would like to see what and for how long they looked at certain source code elements to learn about their behavior. Another possibility is to detect via eye gaze when the developer is having a difficult time understanding the code and provide some recommendations for him/her to make it easier to proceed. The possibilities are limitless.

The data generated from such sessions is extremely valuable since it is not just the bug fix classes that are important but also the classes that were required to be read and understood in order to find the final set of lines that needed to be changed and committed. With eye tracking data, we are able to see all the places that the developer looked to get to the answer and not just the final answer. Eye tracking is able to give us the fine grained line-level gaze data of a developer while they work. Given that we now

have a tool (*iTrace*) that seamlessly integrates with an IDE, the possibilities are endless as to what task can benefit from eye gaze.

One can envision the possibility of recording all gaze activity to help for future bug fixes that are all related to the same feature. Of course we are not there yet. We need an integrated set of tools including a visualization suite for eye tracking data built into the environment to help make sense of all the data that is generated. A 20 min session with a 60 fps eye tracker can easily generate data that is approximately 50 megabytes. An efficient and useful algorithm [4] to weed out stray glances is also necessary.

Since 2006, there has been a surge in eye tracking studies in the software engineering community. The field and technology used is quite intriguing making this an attractive area for young researchers. Even though there are approximately 35 papers using eye tracking in software engineering, we expect this number to drastically increase as more researchers enter the field.

6 Conclusions

The paper presents two broad views on how eye tracking has currently been used in software engineering. The first deals with using an eye tracker as a tool to assess where and for how long a person looks at certain software artifacts. The second area deals with using eye gaze to improve existing software tasks such as software traceability link generation and code summarization. A short description of an eye aware plugin for Eclipse, namely *iTrace* is given that makes larger more realistic studies possible. Eye tracking has potential to greatly enhance a software developer's experience as shown by the studies conducted. The benefit is even more noticeable if researchers use eye gaze to inform software tasks by developing new and innovative algorithms that are developer centric thereby bringing the human aspect back into the more artifact-driven nature of software engineering studies.

References

1. Gethers, M., Dit, B., Revelle, M., Poshyvanyk, D.: Feature location in source code: A taxonomy and survey. J. Softw. Maint. Evolut: Res. Pract. **25**, 53–95 (2013)
2. Ko, A.J., Myers, B.A., Coblenz, M.J., Aung, H.H.: An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans. Softw. Eng. **32**, 971–987 (2006)
3. Fritz, T., Begel, A., Müller, S., Yigit-Elliott, S., Züger, M.: Using psycho-physiological measures to assess task difficulty in software development. In: International Conference on Software Engineering (ICSE), pp. 402–413. IEEE, Hyderabad (2014)
4. Walters, B., Shaffer, T., Sharif, B., Kagdi, H.: Capturing software traceability links from developers' eye gazes. In: 22nd International Conference on Program Comprehension (ICPC), Hyderabad, India, pp. 201–204 (2014)
5. Rodeghero, P., McMillan, C., McBurney, P.W., Bosch, N., D'Mello, S.: Improving automated source code summarization via an eye-tracking study of programmers. In: 36th

IEEE/ACM International Conference on Software Engineering (ICSE 2014), Hyderabad, India, (2014)
6. Siegmund, J., Kästner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., Saake, G., Brechmann, A.: Understanding understanding source code with functional magnetic resonance imaging. In: Proceedings of the 36th International Conference on Software Engineering, pp. 378–389. ACM (2014)
7. Crosby, M.E., Stelovsky, J.: How do we read algorithms? a case study. IEEE Comput. **23**, 24–35 (1990)
8. Bednarik, R., Tukiainen, M.: Temporal eye-tracking data: evolution of debugging strategies with multiple representations. In: Symposium on Eye Tracking Research & Applications (ETRA), pp. 99–102. ACM, Savannah (2008)
9. Bednarik, R., Tukiainen, M.: An eye-tracking methodology for characterizing program comprehension processes. In: Symposium on Eye Tracking Research & Applications (ETRA), pp. 125–132. ACM Press, San Diego (2006)
10. Pietinen, S., Bednarik, R., Glotova, T., Tenhunen, V., Tukiainen, M.: A method to study visual attention aspects of collaboration: eye-tracking pair programmers simultaneously. In: 2008 Symposium on Eye Tracking Research & Applications, New York, NY, USA, pp. 39–42 (2008)
11. Uwano, H., Nakamura, M., Monden, A., Matsumoto, K.: Analyzing individual performance of source code review using reviewers' eye movement. In: 2006 Symposium on Eye Tracking Research & Applications (ETRA), pp. 133–140. ACM Press, San Diego (2006)
12. Uwano, H., Monden, A., Matsumoto, K.-i.: DRESREM 2: An analysis system for multi-document software review using reviewers' eye movements. In: 3rd International Conference on Software Engineering Advances (ICSEA), Sliema, Malta, pp. 177–183 (2008)
13. Sharif, B., Falcone, M., Maletic, J.I.: An eye-tracking study on the role of scan time in finding source code defects. In: Symposium on Eye Tracking Research and Applications (ETRA), Santa Barbara, CA, pp. 381–384 (2012)
14. Sharif, B., Maletic, J.I.: An eye tracking study on camelcase and under_score identifier styles. In: 18th IEEE International Conference on Program Comprehension (ICPC 2010), Braga, Portugal, pp. 196–205 (2010)
15. Binkley, D., Davis, M., Lawrie, D., Maletic, J.I., Morrell, C., Sharif, B.: The impact of identifier style on effort and comprehension. Empir. Softw. Eng. J. **18**, 219–276 (2013)
16. Sharafi, Z., Soh, Z., Gueheneuc, Y.-G., Antoniol, G.: Women and men - different but equal: on the impact of identifier style on source code reading. In: International Conference on Program Comprehension (ICPC 2012), pp. 27–36. IEEE, Passau (2012)
17. Yusuf, S., Kagdi, H., Maletic, J.I.: Assessing the comprehension of UML class diagrams via eye tracking. In: IEEE International Conference on Program Comprehension (ICPC 2007), Banff, AB, pp. 113–122 (2007)
18. Guéhéneuc, Y.-G.: TAUPE: towards understanding program comprehension. In: 16th IBM Centers for Advanced Studies on Collaborative Research (CASCON), pp. 1–13. ACM Press, Canada (2006)
19. Jeanmart, S., Guéhéneuc, Y.-G., Sahraoui, H., Habra, N.: Impact of the visitor pattern on program comprehension and maintenance. In: 3rd International Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, Florida, pp. 69–78 (2009)
20. Sharif, B., Maletic, J.I.: An eye tracking study on the effects of layout in understanding the role of design patterns. In: 26th IEEE International Conference on Software Maintenance (ICSM 2010), Timisoara, Romania, pp. 1–10 (2010)

21. Sharif, B., Jetty, G., Aponte, J., Parra, E.: An empirical study assessing the effect of SeeIT 3D on comprehension. In: 1st IEEE International Working Conference on Software Visualization (VISSOFT 2013), Eindhoven, Netherlands, pp. 1–10 (2013)
22. Turner, R., Sharif, B., Lazar, A.: An eye-tracking study assessing the comprehension of C++ and Python source code. Symposium on Eye Tracking Research & Applications (ETRA 2014), Safety Harbor, Florida, USA, pp. 231–234 (2014)
23. Kagdi, H., Yusuf, S., Maletic, J.I.: On using eye tracking in empirical assessment of software visualizations. In: ACM Workshop on Empirical Assessment of Software Engineering Languages and Technologies, Atlanta, GA, pp. 21–22, (2007)
24. Busjahn, T., Schulte, C., Sharif, B., Simon, Begel, A., Hansen, M., Bednarik, R., Orlov, P., Ihantola, P.: Eye tracking in computing education. In: International Computing Education Research (ICER 2014), Glasgow, Scotland, pp. 3–10 (2014)
25. Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J.H., Schulte, C., Sharif, B., Tamm, S.: Eye Movements in Code Reading: Relaxing the Linear Order. In: Proceedings of ICPC, Florence, Italy (2015)
26. Ali, N., Sharafi, Z., Guéhéneuc, Y.-G., Antoniol, G.: An empirical study on requirements traceability using eye-tracking. In: 28th International Conference on Software Maintenance (ICSM), pp. 191–200. IEEE Computer Society Press (2012)
27. Sharif, B., Kagdi, H.: On the use of eye tracking in software traceability. In: 6th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2011), Honolulu, Hawaii, USA, pp. 67–70 (2011)
28. Walters, B., Falcone, M., Shibble, A., Sharif, B.: Towards an eye-tracking enabled ide for software traceability tasks. In: 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE), San Francisco, CA, pp. 51–54 (2013)