iTrace: Enabling Eye Tracking on Software Artifacts within the IDE to Support Software Engineering Tasks

Timothy R. Shaffer[†], Jenna L. Wise[†], Braden M. Walters[†],
Sebastian C. Müller[*], Michael Falcone[†], Bonita Sharif[†]

[†]Youngstown State University, USA
Department of CS and IS
{trshaffer,jlwise,bmwalters01}@student.ysu.edu
mrfalcone@student.ysu.edu,bsharif@ysu.edu

[*]University of Zurich, Switzerland
Department of Informatics
smueller@ifi.uzh.ch

ABSTRACT

The paper presents *iTrace*, an Eclipse plugin that implicitly records developers' eye movements while they work on change tasks. *iTrace* is the first eye tracking environment that makes it possible for researchers to conduct eye tracking studies on large software systems. An overview of the design and architecture is presented along with features and usage scenarios. *iTrace* is designed to support a variety of eye trackers. The design is flexible enough to record eye movements on various types of software artifacts (Java code, text/html/xml documents, diagrams), as well as IDE user interface elements. The plugin has been successfully used for software traceability tasks and program comprehension tasks. *iTrace* is also applicable to other tasks such as code summarization and code recommendations based on developer eye movements. A short video demonstration is available at `https://youtu.be/3OUnLCX4dXo`.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Experimentation, Measurement, Human Factors

Keywords

eye-tracking, plugin, comprehension, traceability

1. INTRODUCTION

The use of eye trackers has become increasingly popular in the software engineering (SE) community, as evidenced by the increasing number of publications in mainstream conferences and journals [2, 3, 5, 7]. Eye trackers have also become cheaper and more affordable. An eye tracker allows an SE researcher to collect eye movements of software engineers while they work on software tasks such as adding a new feature, fixing a bug, or on a general comprehension task. The eye movement data is used to study the cognitive thought processes [6] of developers as they perform a task using different software artifacts. Existing eye-tracking studies have mainly studied developers comprehending software artifacts such as source code, models such as UML diagrams, and software visualizations.

An eye tracker consists of both hardware and software. The hardware is a physical device that usually sits under the monitor. The software provided by eye tracking vendors is in the form of an experiment builder (e.g., Tobii Studio from Tobii Inc.) that allows researchers to build the experiment workflow using various stimuli such as a still image, a website, a video recording, or free form desktop recording.

Most existing eye tracking studies (besides [11] and [4]) done in the SE community use small snippets of code shown as an image to study participants. An ad hoc system to support scrolling using slider bar events for a few small programs was done in [9] but the tool is unavailable. In all other studies, the image needs to be displayed on the screen all at once and participants are not allowed to scroll as scrolling would interfere with data collected and make post processing extremely difficult if not impossible. This is because the eye tracker is not aware of the type of stimulus presented to it. It reports the (x, y) coordinates where a person is looking on the screen, but is not aware of what exists at that position. When the image is kept static and not allowed to move (which happens during scrolling) it is easy to map (after the experiment) what the (x, y) coordinates represent on the stimulus (which would be a snippet of source code in case of program comprehension). This mapping process is not automatic and needs to be done by the researcher after the experiment is conducted by creating areas of interest on the stimulus, which is an extremely tedious process. Some of this difficulty is alleviated by using a tool such as eyeCode (https://github.com/synesthesiam/eyecode/) that automatically detects words in the stimuli, but even with eyeCode, the post processing is still tedious and only works on a single static image.

The above mentioned setup works well in studies conducted in psychology (where studies mainly consist of reading text paragraphs and looking at images), but does not scale well to SE. In order to run realistic experiments with eye trackers in SE, we need to be able to run eye track-

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy

http://dx.doi.org/10.1145/2786805.2803188

ing studies on realistic systems consisting of hundreds of source files and not be limited to a single static view. *iTrace* was designed exactly for this purpose. There is no need for the researcher to manually map (x, y) coordinates to source code elements as all of this time-consuming labor-intensive process is now done automatically by *iTrace*. *iTrace* runs uninterrupted in the background within Eclipse, recording developers' eye movements while they are working. The plugin is open source under the GPL license at `http://seresl.csis.ysu.edu/iTrace`.

2. ITRACE ARCHITECTURE

This section describes a high-level overview of the *iTrace* plugin architecture and its integration into the Eclipse UI. Additional details about architecture design and our initial ideas for the tool are discussed in [10]. The current implementation handles gazes on Java source code, text files including HTML and XML files, and Eclipse UI elements. *iTrace* is designed with a modular architecture, and it is easy to write new handlers for different file types to collect fine-grained data at the statement level.

2.1 Overview

Enabling eye tracking for the IDE requires implementing three high-level tasks: 1) capturing a user's gazes from the tracking device, 2) determining which UI element the user is looking at within the IDE, and 3) processing this information toward some functional goal. Each of these tasks must be done in parallel to achieve maximum responsiveness. To overcome this challenge, we use a multithreaded design consisting of a thread for each task that communicates with other threads via shared blocking queues. Our architecture makes use of a `Gaze` object class and three main Java interfaces for working with gazes: `IGazeHandler`, `IGazeResponse`, and `ISolver`. Each is described below.

`Gaze` – Represents position, time, pupil size, validity, and error information for each gaze detected by the eye tracker.

`IGazeHandler` – Describes a handler that accepts a gaze on a widget and returns a `IGazeResponse`.

`IGazeResponse` – Describes information observed by a specific gaze on a specific widget.

`ISolver` – Describes what to do with a specific `IGazeResponse` object.

2.2 Integration with Eclipse

To implement our design as a plugin within the Eclipse framework we make use of the `Widget` class (`org.eclipse.swt.widgets.Widget`). This class represents each user interface object that is part of the IDE, and as such stores some content being viewed by the user. It also exposes two methods – `getData()` and `setData()` – which we use to bind and access gaze handler objects such that each UI element with content of interest has its own gaze handler instance. The plugin is initialized following the process summarized in Figure 1.

The lifecycle event handlers manage tasks such as pausing gaze processing when the IDE loses focus or is minimized as well as initializing gaze handlers on new editor windows (e.g. when the user opens a new source file). Eye tracking devices are implemented in the plugin by implementing the `IEyeTracker` interface and modifying the eye tracker factory class. We use the Java Native Interface (JNI) to interface

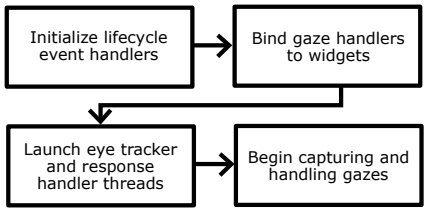

Figure 1: Plugin initialization process.

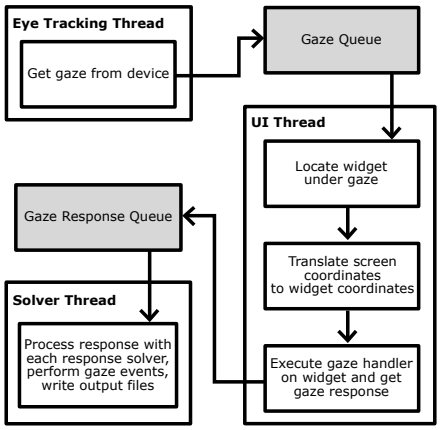

Figure 2: Handling of gazes and gaze responses by multiple threads.

with the eye tracker SDK in C or C++ (for e.g., the `TobiiTracker` class). For convenience, we have also implemented the system mouse tracker as a proxy for eye movements when an eye tracker is unavailable for testing.

Capturing and processing gazes follows the process summarized in Figure 2 and continues until stopped by the user or paused by minimizing or taking focus from the IDE. If capturing is paused, it will automatically resume when the IDE window is restored or acquires focus. Resizing of the window is also supported by *iTrace*.

Synchronization of thread communication is handled through the Java generic class `LinkedBlockingQueue` used to implement queues shared by threads. Gazes are consumed by the UI thread in fixed-size chunks and gaze responses are consumed entirely by the response handler thread in order to reduce the amount of locking that must take place.

Source code tracking as described in Section 3 is implemented using the `ASTParser` class built into Eclipse to parse Java syntax and generate an abstract syntax tree (AST) storing line and column numbers of each source code entity. We implement a gaze handler for the editor window that translates gaze coordinates to line and column numbers using methods exposed by the `StyledText` Eclipse class, which stores source code content displayed by the IDE. Using the line and column numbers under the gaze and the line and column numbers of each source code entity we are able to determine the entity that the user is currently viewing.

3. FEATURES

In this section, we describe the features that *iTrace* currently provides.

3.1 Session Creation

The session information consists of a generated session ID, the session purpose (new feature, bug fix, refactoring, general comprehension or other), and a longer session description. In addition to this, the developer ID with an optional developer name is requested. Session information is required before tracking starts.

3.2 Calibration

Every eye tracker needs to be calibrated before use. During calibration, the user is asked to look at several dots that appear on the screen while the eye tracker uses the user's eye features along with its 3D eye model to calculate gaze data. *iTrace* uses a 9-point calibration mechanism, interfacing with the eye tracker's native libraries to calibrate. When calibration is complete, *iTrace* displays the results of the calibration so that the user is able to verify the calibration quality. The user can accept the calibration, or recalibrate.

3.3 Displacement Adjustment

In order to check for displacement or drift, *iTrace* supports a crosshair feature. When enabled, it displays a green crosshair showing where the user is looking. For some users, there is always some displacement of the eye gaze as shown by the crosshair vs. the actual point looked at on the screen. We test this by asking the user to look at certain words at the top left, bottom left, top right and bottom right after calibration. If the actual eye gaze is off from the intended position, one is able to adjust the displacement on the x and y axis as needed to bring the crosshair in line with the user's gaze. This displacement setting is used throughout each session. It is not always necessary to adjust for drift and should be used only when required.

3.4 Source Code Entity Level Tracking

iTrace supports fine-grained tracking of software artifacts at the line and word level. In particular, emphasis is given to source code as it is the most structured and semantically rich artifact. The current *iTrace* model is able to map gazes to source code entities (SCE) types such as classes, methods, variables, comments, method invocations, conditional expressions, and *enum*, *import*, *for*, *if*, *while*, and *switch* statements. For each of these types, it also states how the SCE type is used in the code, i.e. a declaration vs. an invocation. For example, if a user looked at a method call, it would be considered a `use` of the *method* type.

3.5 Raw Data Exports

The number of gazes recorded by *iTrace* depends on the number of samples per second output by the eye tracker. Each gaze recorded by the eye tracker is used to generate a gaze response object. *iTrace* currently supports gaze response export into JSON and XML. An example gaze response in XML format is shown in Figure 3. Properties of the gaze itself, such as time, pupil diameter, validation, and (x, y) coordinates, are stored as attributes of the `response` element. Of these properties, all except `system_time` and `nano_time` are read from the tracker. `system_time` is the POSIX time on the host system expressed in milliseconds, and `nano_time` is a high-precision time expressed in nanoseconds. The `type` attribute reports the region of the Eclipse UI where the gaze fell. In the example, the gaze was in a text editor, so the `response` element has some additional

```
<response line_base_y="412" line_base_x="90" col="23" line="38"
font_height="24" line_height="37"
path="C:\programs\src\tasks\program5.java"
nano_time="2969874330552" system_time="1433180043835"
tracker_time="1433178283244"
right_pupil_diameter="3.151641845703125"
left_pupil_diameter="3.095123291015625" right_validation="1.0"
left_validation="1.0" y="437" x="510" type="text"
name="program5.java">
  - <sces>
      <sce type="METHOD" name="java.lang.String.length()"
          end_col="30" start_col="17" end_line="38" start_line="38"
          total_length="13" how="USE"/>
      <sce type="VARIABLE"
          name="tasks.MiscUtilities.canonPath.trim" end_col="30"
          start_col="10" end_line="38" start_line="38"
          total_length="20" how="DECLARE"/>
      <sce type="IFSTATEMENT" name="IfStatement-l35c4"
          end_col="5" start_col="4" end_line="51" start_line="35"
          total_length="617" how="DECLARE"/>
      <sce type="METHOD" name="tasks.MiscUtilities.canonPath
          (java.lang.String)" end_col="3" start_col="2" end_line="77"
          start_line="17" total_length="2192" how="DECLARE"/>
      <sce type="TYPE" name="tasks.MiscUtilities" end_col="1"
          start_col="0" end_line="78" start_line="15"
          total_length="2270" how="DECLARE"/>
    </sces>
</response>
```

Figure 3: An *iTrace* gaze response record.

attributes, including the filename, line, column, editor font height in points, line height in pixels, and the (x, y) coordinates of the upper left corner of the line. For Java files, the Eclipse AST is queried to determine the source code elements on which the gaze fell. This step is necessary, since relying only on the line number would not be accurate if the content of the file is changed during a tracking session.

In the example, the `sce` elements, sorted from most specific to least specific, report the types of the source code elements, how they are being used, their positions within the file, and the number of characters comprising the source code elements. In this case the gaze fell on a call to the method `java.lang.String.length()` in the declaration of the `trim` variable, inside an if statement in the `canonPath` method, within the `tasks.MiscUtilities` class.

3.6 Fixation Exports

In eye-tracking terminology, a fixation [6] is when the eye stabilizes for a certain duration at a particular point of interest. *iTrace* calculates fixations by running a basic fixation filter on the raw data. In simple terms, a set of raw gazes that fall around the same area are grouped and merged together to form a fixation record, along with the fixation start time and the duration.

4. USAGE SCENARIOS

The usage scenarios of *iTrace* fall into two broad categories. First, it can be used as a method to assess and learn about how developers navigate and look at different software artifacts. Second, the data can be used to inform software development tasks. An example of *iTrace* being used in each of these two scenarios is described below.

4.1 Program Comprehension

iTrace has recently been used by Kevic et al. [4] to investigate detailed developer behavior while performing realistic change tasks on a large open source system. The study was conducted on 22 developers. To investigate the added benefit of data generated from *iTrace*, the study compared eye tracking data with Mylyn interaction history data, both of which were gathered simultaneously. The authors found that *iTrace* does capture more contextual data on source

code elements, and more importantly captures different aspects of developer activity compared to interaction data.

4.2 Software Traceability

The data generated from *iTrace* has also been used by Walters et al. [11] as input to help recover software traceability links. The links are automatically derived between bug reports and source code entities from a set of developer eye tracking sessions on bug fixes. The concept of collective intelligence (ability to gather knowledge from other developers towards a common goal) [8] was used in the above algorithm. The results were very promising, and eye gaze does indeed seem to work well to uncover links between relevant code entities and the bug report. They developed an algorithm to find relevant entities looked at using a weighting scheme based on time. This helps weed out source code entities that are looked at initially but later abandoned. The version of *iTrace* used in this study is available as a release[1]. The link generation algorithm used on *iTrace* data along with the replication package is also publicly available[2].

4.3 Future Scenarios

Besides the above scenarios, there are many software tasks that can directly benefit from the data that *iTrace* provides. Both Rodeghero et al. [7] and Ali et al. [1] use eye tracking data as a means to improve automatic code summarization and weighting schemes in software traceability link recovery respectively. Fritz et al. [3] also uses eye tracking data to predict task difficulty. However, these studies use small snippets of code that had to fit on one screen. With *iTrace* these types of studies could be conducted on large systems or even larger snippets of code, thereby providing fine grained eye tracking data that is mapped to source code entities.

5. CURRENT LIMITATIONS

At the time of this writing, two eye trackers, the Tobii X60 and Tobii EyeX are supported. However, the modular architecture of *iTrace* allows for easy addition of new devices. Additionally, the tool is implemented as an Eclipse plugin and therefore cannot capture gazes outside of Eclipse. It can however determine that the Eclipse window is not in focus due to the lack of gazes. Tracking of elements behind an open dialog box such as the search window is not currently supported. Finally, the current version of *iTrace* does not support code folding yet. In an upcoming version of *iTrace*, we plan to support search view tracking and code folding.

6. CONCLUSIONS AND FUTURE WORK

The paper describes *iTrace*: an Eclipse plugin that makes collecting eye gaze on software artifacts possible on large software systems. For source code documents, *iTrace* maps the eye gazes to fine-grained source code entities looked at. The main contributions of this paper are a) an eye-aware Eclipse plugin b) easy to comprehend gaze export format for source code entities and c) demonstrated usage of *iTrace* for program comprehension and software traceability tasks.

Currently, fine-grained line and word level support is provided for Java files. Other files such as text, xml and html files are tracked, but not at the line-level. *iTrace* can also record gazes on Eclipse UI elements, such as the project explorer. Additional handlers can be written for finer-grained information from structured text files. These custom handlers can be specifically tailored to each researcher's needs.

The current research provides several directions for future work. First, support for more eye trackers is needed. Next, support for tracking Javadocs and UML diagrams will be added. Other features, such as support for tracking during search, code folding, replaying gazes over a particular method and keeping track of method renaming are planned.

[1] `https://github.com/YsuSERESL/iTrace/releases/tag/icpc2014`

[2] `http://www.csis.ysu.edu/~bsharif/itrace-pilot/`

Acknowledgments

Special thanks to Huzefa Kagdi for inspiring conversations leading to the development of *iTrace*.

References

[1] N. Ali, Z. Sharafi, Y.-G. Guéhéneuc, and G. Antoniol. An empirical study on requirements traceability using eye-tracking. In *ICSM 2012*, pages 191–200, 2012.

[2] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. Paterson, C. Schulte, B. Sharif, and S. Tamm. Eye movements in code reading: Relaxing the linear order. In *ICPC 2015*, page 12 pages to appear, 2015.

[3] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger. Using psycho-physiological measures to assess task difficulty in software development. In *ICSE 2014*, pages 402–413, 2014.

[4] K. Kevic, B. Walters, T. Shaffer, B. Sharif, T. Fritz, and D. Shepherd. Tracing software developers' eyes and interactions for change tasks. In *ESEC/FSE 2015*, page 12 pages to appear, 2015.

[5] S. Müller and T. Fritz. Stuck and frustrated or in flow and happy: Sensing developers' emotions and progress. In *ICSE 2015*, page 12 pages to appear, 2015.

[6] K. Rayner. Eye movements in reading and information processing: 20 years of research. *Psychological bulletin*, 124(3):372, 1998.

[7] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *ICSE 2014*, pages 390–401, 2014.

[8] M.-A. Storey, L. Singer, B. Cleary, F. Figueira Filho, and A. Zagalsky. The (r)evolution of social media in software engineering. In *Proceedings of the on Future of Software Engineering*, pages 100–116, 2014.

[9] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, ETRA '06, pages 133–140. ACM, 2006.

[10] B. Walters, M. Falcone, A. Shibble, and B. Sharif. Towards an eye-tracking enabled ide for software traceability tasks. In *Workshop on Traceability in Emerging Forms of Software Engineering*, pages 51–54, 2013.

[11] B. Walters, T. Shaffer, B. Sharif, and H. Kagdi. Capturing software traceability links from developers' eye gazes. In *ICPC 2014*, pages 201–204, 2014.