Empirical Software Engineering _####################_
https://doi.org/10.1007/s10664-022-10209-3

For Approval

Deja Vu: semantics-aware recording and replay of high-speed eye tracking and interaction data to support cognitive studies of software engineering tasks—methodology and analyses

Vlas Zyrianov[1] · Cole S. Peterson[2] · Drew T. Guarnera[3] · Joshua Behler[4] · Praxis Weston[4] · Bonita Sharif[2] · Jonathan I. Maletic[4]

Accepted: 13 July 2022

Abstract

The paper introduces a fundamental technological problem with collecting high-speed eye tracking data while studying software engineering tasks in an integrated development environment. The use of eye trackers is quickly becoming an important means to study software developers and how they comprehend source code and locate bugs. High quality eye trackers can record upwards of 120 to 300 gaze points per second. However, it is not always possible to map each of these points to a line and column position in a source code file (in the presence of scrolling and file switching) in real time at data rates over 60 gaze points per second without data loss. Unfortunately, higher data rates are more desirable as they allow for finer granularity and more accurate study analyses. To alleviate this technological problem, a novel method for eye tracking data collection is presented. Instead of performing gaze analysis in real time, all telemetry (keystrokes, mouse movements, and eye tracker output) data during a study is recorded as it happens. Sessions are then replayed at a much slower speed allowing for ample time to map gaze point positions to the appropriate file, line, and column to perform additional analysis. A description of the method and corresponding tool, Deja Vu, is presented. An evaluation of the method and tool is conducted using three different eye trackers running at four different speeds (60 Hz, 120 Hz, 150 Hz, and 300 Hz). This timing evaluation is performed in Visual Studio, Eclipse, and Atom IDEs. Results show that Deja Vu can playback 100% of the data recordings, correctly mapping the gaze to corresponding elements, making it a well-founded and suitable post processing step for future eye tracking studies in software engineering. Finally, a proof of concept replication analysis of four tasks from two previous studies is performed. Due to using the Deja Vu approach, this replication resulted in richer collected data and improved on the number of distinct syntactic categories that gaze was mapped on in the code.

Communicated by: Zhenchang Xing, Kelly Blincoe

This article belongs to the Topical Collection: *Software Maintenance and Evolution (ICSME)*

✉ Jonathan I. Maletic
jmaletic@kent.edu

Extended author information available on the last page of the article.

Keywords Eye tracking · High-speed tracking · Empirical software engineering · Program comprehension · Replication analyses

1 Introduction

Studying how developers read and understand source code is a core research topic in software engineering. Research on mental models of program comprehension dates into the 1980's (Brooks 1983; Letovsky 1987; Rist 1986; Soloway and Ehrlich 1984; Pennington 1987; Von Mayrhauser and Vans 1995). Historically, researchers use approaches such as think-aloud and pre/post surveys to collect data for such studies. Recently, researchers are taking advantage of eye tracking technology to study how people read source code (Obaidellah et al. 2018). In general, eye trackers are a vital research tool in understanding how people observe and in turn comprehend visual stimuli (Rayner 1978). Researchers successfully use eye tracking hardware to better understand how people read natural language prose, understand diagrams, and process visual landscapes. Computer scientists use eye tracking devices to study how people interact with graphical user interfaces and web pages (Goldberg et al. 2002). The software engineering community is currently using eye tracking equipment to study how developers read and understand source code (Sharafi et al. 2015b). There is a recently published practical guide on conducting eye tracking studies in software engineering (Sharafi et al. 2020) that covers the technology and best practices to follow when conducting eye tracking studies in software engineering.

Eye tracking devices come in a wide range of forms and take advantage of a range of technologies. The devices are made up of hardware, mainly specialized cameras, along with sophisticated software that computes the focal point of the eyes using data collected by the cameras. The software is needed to map each of the eye gazes to locations on a visual stimulus (e.g., computer screen). Additionally, eye tracking devices differ greatly with regards to accuracy (of tracking eye movements) and the applications and environments they can be applied to (Andersson et al. 2010). In particular, studying how people read and comprehend text or source code requires high precision (and costly) eye tracking hardware and software. While determining general spatial regions where a person is looking (left, right, up, down) only requires simple and low cost hardware and software. Low cost systems cannot identify the exact focus of the eyes, such as the word or letter someone is looking at. They only work well on larger stimuli such as objects in computer games. A high quality, accurate, research-grade eye tracking device allows researchers to determine the exact xy-coordinate on the screen a person is examining. The higher-end eye trackers, in a controlled setting, can pinpoint down to the letter being examined. Research on reading prose and source code most often requires accuracy to the word level at minimum.

Using an eye tracker to study a developer (participant) works by presenting an image or text (stimuli) on a computer screen and then using the data from the cameras to determine the location (xy-coordinate) the person is looking. However, there are a number of limitations to this technology. The subject must be forward looking at the stimuli, cannot move around the room, and must be fairly stationary. While these are not serious limitations for conducting scientific studies, there is one underlying limitation that poses a substantial road block for studying how programmers understand large, real-world software. Accurate research-grade eye trackers only work on fixed stimuli (i.e., an image or text block) that fits on the computer screen. Changes to the stimuli (screen), such as scrolling or switching files, present a very complex problem. Mapping the (x, y) to the correct position in the stimuli (say a 1000 line file) becomes impractical.

Fortunately, infrastructure to deal with this problem has been recently constructed, namely iTrace (Sharif et al. 2016b, 2019; Guarnera et al. 2018; Sharif and Maletic 2016a). iTrace (www.i-trace.org) allows a software engineering researcher to conduct eye tracking studies directly in an integrated development environment (IDE) such as Visual Studio or Eclipse. It supports the tracking of eye gazes in the presence of scrolling and context switching. Thus, researchers can study developers in a real-world environment using large realistic software systems. iTrace does this by linking the IDE via a plugin architecture and invoking application and system calls to map the screen xy-coordinate to a line and column in the file in real time. This is then used in a post processing phase to determine the source code token being examined by the study participant.

Unfortunately there are some technical limitations to this approach that pose a problem for researchers studying developers. Eye trackers sample eye gazes x-times every second denoted by the frame rate. For example, a 120 Hz eye tracker generates 120 samples per second of raw eye gaze coordinates. Each gaze needs to be looked up in real time to map to the line, column within the file. Of course the lookup time is bound to the time it takes for the system calls to be executed and return. If the response time of this system call is too long it is not possible to map all (120) gazes coming in accurately to the correct file location in time. Through use of the iTrace infrastructure we determined that the maximum frame rate at which this can be done in real time is approximately 60 Hz (for both Visual Studio and Eclipse). This implies that anything above 60 Hz will cause the tracker in iTrace to either incorrectly map data or drop gaze points altogether. While having a faster computer may help a to some degree, getting to 120 Hz, 300 Hz or even 1000 Hz (at which reading studies are typically done in psychology) is impossible with real time mapping.

The research presented here, and previously in Zyrianov et al. (2020), addresses this limitation of the current iTrace architecture by taking all the processing offline. While the IDE API function call response time is fixed, our technique allows for all input events to be recorded and replayed back in a post processing step at a slower rate (several options exist on playback rate). This allows for accurate mapping of gaze data to source code locations with very high-speed eye trackers. The technique is implemented in Deja Vu, a novel tool that leverages the iTrace infrastructure and integrates well with its workflow. The technique and details of the Deja Vu's approach are presented.

The main contributions presented in this paper are:

- Formalization. We introduce a fundamental problem in performing eye tracking studies in practical developer environments with high-speed eye trackers.
- Technique. We present a novel technique to solve the technological problem presented using automated recording and semantics-aware replaying of eye tracking and interaction data to support cognitive studies of software engineering tasks.
- Tool. The novel technique is realized and implemented in a practical tool, Deja Vu, that is integrated in to the iTrace eye tracking infrastructure. iTrace, along with Deja Vu, is available at www.i-Trace.org. An initial release of Deja Vu is available at https://doi.org/10.5281/zenodo.3976332. Future releases of Deja Vu will be available on the iTrace website: http://www.i-trace.org/downloads/
- Evaluation. An evaluation of the fundamental problem of collecting high-speed eye tracking data with and without Deja Vu is presented in the context of three integrated development environments (i.e., Eclipse, Visual Studio, and Atom).
- Replication Analysis. A replication analysis is conducted by collecting proof of concept data with a small sample of participants using Deja Vu on four tasks from two prior

studies. The data is then compared to prior studies to show evidence of added syntactic categories mapped using the Deja Vu approach. This analysis is presented in Section 8.

This paper extends our prior conference paper (Zyrianov et al. 2020) in the following ways. *First*, experiments evaluating Deja Vu now include the Atom IDE and are described in Section 7. The prior paper only included experiments for Eclipse and Visual Studio. *Second*, a replication analysis (Section 8) of tasks in two prior studies (Saddler et al. 2020; Kevic et al. 2015) is done to provide evidence of the richer syntactic categories that are provided with the Deja Vu approach. Section 8 is a completely new addition that required the collection of proof of concept data to illustrate the additional useful information the Deja Vu approach provides. *Third*, a detailed description and illustration of the iTrace infrastructure, the delay mechanism in Deja Vu, and the post processing Toolkit (Section 5) is given including new and updated diagrams as well as usage scenarios (Section 6.4). *Fourth*, we integrated Deja Vu directly into the iTrace-Core (previously it was stand alone), thereby greatly enhancing the actual use by researchers to support their studies and *finally* the practicalities of implementation have been updated to include the addition of supporting mouse double click events during the reply of Deja Vu sessions and fixing of race conditions (Section 6.3).

The paper is organized as follows. Section 2 presents related work in interaction monitoring. Section 3 formally presents the problem and motivation for Deja Vu. Section 4 clearly defines the types of effects that could be studied with high speed tracking and the need for supporting high speed data collection. Section 5 discusses details of the Deja Vu architecture, design decisions, and how Deja Vu integrates with iTrace. Section 6 discusses implementation details of the recording and replaying stages including the challenges faced and how they were mitigated or need managed. The section also touches on usage scenarios for iTrace and Deja Vu. Section 7 provides an evaluation on the impact of data output rates from eye tracking devices on real-time analysis of eye tracking data on source code with respect to the iTrace framework (Guarnera et al. 2018; Sharif and Maletic 2016b). Section 8 presents a replication analysis on four tasks taken from two prior studies using the Deja Vu approach. Section 9 summarizes our methodology, analyses, and presents avenues of future work.

2 Related Work

This section presents related work in automatically capturing user interactions that is most relevant to this paper's scope.

Capturing user interaction data for analysis is a common approach in a variety of computational research studies. Minelli et al. (2014, 2015, 2016) record mouse, keyboard, and IDE interaction data. Fine grain interactions are grouped into broad categories such as comprehension, editing, navigating, etc. to observe developer behavioral during typical tasks. Findings about what activities consume the most developer time, the proportion of development time is dedicated to program comprehension, and the IDE navigational efficiency of developers are presented. The Blackbox project (Brown et al. 2018) has collected programming interactions within the BlueJ Java IDE for over five years. This dataset has been aimed at providing raw data for research analysis towards better understanding software development behaviors of novice developers. Mylar (Kersten and Murphy 2006), now known as Mylyn for the Eclipse IDE, allows a developer to track IDE usage activity related to defined tasks. These task contexts can be easily switched in order for developers to multitask

without the need to manually relocate artifacts upon returning to a previous task activity. Deja Vu drastically differs from Mylyn in that Deja Vu is intended to store interactions along with cognitive information (eye tracking data) for the purpose of replay and subsequent analysis while Mylyn is an active development productivity tool.

ActivitySpace (Bao et al. 2015) stores mouse and keyboard events related to applications used by software developers to accomplish daily tasks. Event information is logged to a database as an "action record" to create a historical profile of developer interactions. Action records are grouped by a user defined time window and can be queried to help remind developers of resources used and actions taken while working on a given task to improve productivity. Interaction data from ActivitySpace has also been used with machine learning techniques are compared to classify developer activity into higher level categories such as coding, debugging, testing, navigation, web browsing, and documentation (Bao et al. 2018).

In addition to the capture user interactions, running simulated interactions is a popular solution for software testing research. Sikuli is used in Sun et al. (2018) to construct synthetic macro scripts that are application agnostic based on common keyboard and mouse usage. User interactions are supplemented with desktop screenshots and image processing to determine the targets of the actions and automate GUI testing. Specific environments such as websites (Burg et al. 2013; Niño et al. 2005) and Android applications (Yan et al. 2018; Guo et al. 2019) have also been instrumented to record and replay user interactions for the purpose of testing and evaluating web or GUI based applications.

Capture and replay approaches also benefit general purpose automation techniques. The Online Synchronous Education Platform (OSEP) records and abstracts user interactions with websites allowing for editable interactions scripts to be run as pre-recorded or synchronous demonstrations to support educational environments (Sun et al. 2014). Using the same framework, an system for automating common or lengthy website interactions is also proposed to improve user productivity (Sun et al. 2015). Recent works by Ramler et al. (2020) and Bernal-Cárdenas et al. take a different approach to capturing and replaying user interaction (2020). Instead of instrumenting applications or recording interactions at an OS level, recorded video of an activity is broken down into individual still frames which are post processed to reverse engineer user interactions shown in the video. Since we do not have video as input, we did not favor this approach and instead focus on recording IDE interactions as they occur.

Summary In order to learn more about eye tracking in program comprehension, we direct the reader to a survey of eye tracking (Obaidellah et al. 2018) and a practical guide (Sharafi et al. 2020) on conducting experiments in program comprehension. These two works summarize the state of the art in eye tracking research for software engineering. While Deja Vu makes use of existing recording and replaying techniques, it differs from the state of the art by recreating an eye tracking study in its entirety. User interactions with mouse and keyboard and gaze locations are all replayed to simulate a prior eye tracking study while allowing ample time for more detailed analysis that is not feasible to perform in real-time using high speed eye tracking equipment due to system call response time limitations. Additionally, Deja Vu affords researchers an opportunity to replicate a study any number of times while analyzing the study in different ways each time to greatly increase the value of participant recording sessions. This is a novel contribution to the current state of the art and provides the eye tracking software research community added incentive to use eye tracking equipment in their studies. The additional advantage of supporting high-speed trackers above 60 Hz (most research grade trackers are 120 Hz or higher) without data loss enables

many different types of cognitive analyses (outlined in Section 4) that were unable to be done before because of the engineering problem described.

3 Background and Problem Formalization

There are decades of research, that take advantage of eye tracking technology, to study how people comprehend visual stimuli (Rayner 1998). Modern eye trackers collect a person's eye gaze data on the visual display (referred to as the stimulus) in an unobtrusive way while the subject is performing a given task. This eye movement data provides very valuable insight into comprehension strategies (Soloway and Ehrlich 1984) as to how and why people arrive at a certain solution. Eye movements are essential to cognitive processes because they focus a subject's visual attention to the parts of a visual stimulus that are processed by the brain. Visual attention triggers cognitive processes that are required to perform such things as comprehension. Eye movement is also a proxy for cognitive effort (Rayner 1998) and allows us to determine what parts of a visual stimuli are difficult to understand.

The underlying basis of an eye tracker is to capture various types of eye movements that occur while humans physically gaze at an object of interest. Fixations and saccades are the two types of eye movements. A fixation is the stabilization of eyes on an object of interest for a certain period of time. Fixations are made up of multiple raw gazes. Saccades are quick movements that move the eyes from one location to the next (i.e., re-fixates). Dwell time is defined as the sum of all gazes in a dwell (one visit to an area of interest from entry to exit) (Holmqvist and Andersson 2017). An area of interest is defined by the researcher as any part of the stimulus that is of interest for analysis. For example, in source code it can be a token, a line, or multiple statements. A scan path is a directed path formed by saccades between fixations. The general consensus in the eye tracking research community is that the processing of visual information occurs during fixations, whereas, no such processing occurs during saccades (Duchowski 2007). The visual focus of the eyes on a particular location triggers certain mental processes in order to solve a given task (Just and Carpenter 1980). Modern eye trackers are accurate to 0.5 degrees (0.25 in. diameter) on the screen. In Fig 1, we see eye gazes on source code (some areas having a much higher density of fixations than others). The fixations are shown as circles on the diagram. The radius of the circle represents the duration of the fixation. The bigger the radius, the more time is spent looking at that particular point. Each fixation has a number displayed in the center of the circle, which indicates the order in which the fixation occurred.

Not all eye trackers are made equal. Generally, eye trackers range from low-cost consumer grade to more expensive research-grade tracking equipment. Research-grade eye trackers are thoroughly tested for accuracy, quality, and reliability compared to low-cost models. Low-cost eye trackers costing approximately $200 USD are for consumer use (mainly gaming). Low-cost eye trackers miss the subtle differences in how humans read and navigate text. Another difference is the frame rate. Low-cost eye trackers capture gazes at a slower rate compared to the research-grade ones. More gazes captured per second give more detailed insight into how people read and analyze software artifacts.

The current generation of eye tracking devices offer a wide range of data rates (Andersson et al. 2010). Older and entry level devices tend to operate at 60 Hz meaning that 60 data points are provided within one second. When performing real-time analysis with received gaze data, analysis tools would be left with approximately 17 milliseconds (ms) for any analysis before a subsequent new data point will be received from a tracker. This window narrows as modern trackers are capable of supporting anywhere from 120 Hz to over 2000 Hz.

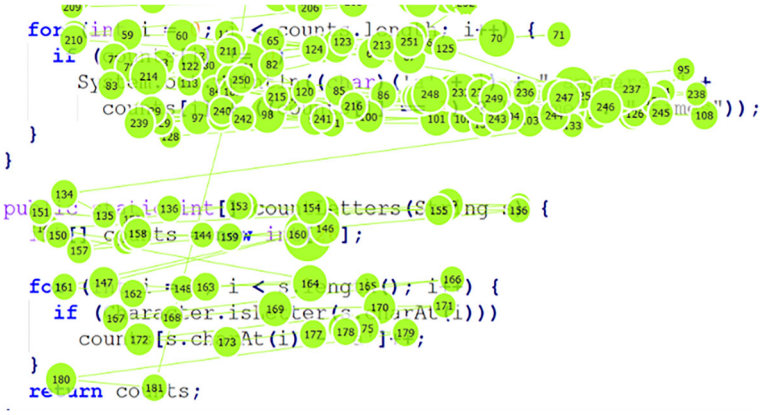


Fig. 1 Gaze plot of a developer's fixations on code. Fixations are represented as circles. The number in the circle represents the fixation index in time. Fixations are linked via a scan path shown by lines connecting consecutive fixations

Eye tracking of source code within an integrated development environment (IDE) is a serious challenge compared to the traditional approach of using static images or text that fit completely on a single screen. In the case of a static stimulus, the position of the image or the source text has little to no variation. The gaze data recorded while the stimulus is visible can easily be mapped down to the pixel on an image-based representation of the data on the display. In contrast, while using an IDE, users may manipulate the view of the source code in any number of ways such as scrolling, file switching, or even editing. These actions require that the gaze data recorded is contextually informed of state of the IDE with respect to the positioning of the source code text and interface elements at a specific moment in time. For example, if a user is scrolling through a source code file looking for a specific identifier, the user's eye positioning may remain fixed within a limited region of the display as the text scrolls past. The issue is that location of the stimulus is changed drastically due to scrolling and it is no longer possible to easily map the screen location of a gaze to the stimulus.

In the case of the iTrace infrastructure (Guarnera et al. 2018; Sharif and Maletic 2016b) (or other similar gaze analysis infrastructures), IDE plugins map gaze locations to interface elements and source code text. The high latency of IDE plugin environment API calls significantly limits the feasibility of deep real-time gaze and textual analysis at the data sampling rate of high-speed trackers. Currently, solving this problem requires serious tradeoffs. One option is to drop gaze points received while the plugin is busy performing gaze mapping operations causing valuable data points to be lost. Another choice is to buffer all gazes to prevent data loss, but this causes the mapping process to steadily fall behind as the mapping process is a real-time operation and relies on the context of the current state of the IDE when the gaze data is received. This ultimately leads to a desynchronization of the gaze data and the IDE state and renders the data invalid.

4 The Need to Support High Speed Eye Tracking

Enabling support for high speed trackers allows researchers to collect data for studying various software engineering tasks and better enable them to come to conclusions similar to

that of cognitive psychology reading studies that typically use 1000–2000 Hz trackers. We now enumerate several benefits of having support for high speed trackers implemented in Deja Vu by extending current eye tracking community infrastructure.

Running realistic studies using the community infrastructure such as iTrace on a tracker greater than 60 Hz is now possible as Deja Vu takes full advantage of the faster frame rate. Most affordable research grade eye trackers are at least 120 Hz. This enables researchers to take advantage of the higher frame rate available to them. The higher the sampling rate, the greater the precision of the eye in space, causing less error on dwell time (Holmqvist and Andersson 2017) at any given point on the stimulus. This relates directly to the accuracy of the eye tracker. Accuracy is important when drilling down to the specific token the developer is examining. Tokens are of varying length (e.g., short variable names, data types (`int`) or opening and closing braces) and accurate dwell time is critical for a study. Additionally, with higher precision we can accurately map the eyes to the parts of the stimuli with more realistically sized fonts. Currently, to overcome this limitation, researchers use a larger font, however, this is not very realistic as developers do not normally program in very large fonts. With a 60 Hz tracker, the window of error is about 32 ms—once every 16 ms in either direction (Andersson et al. 2010).

There are known attentional effects such as attentional cuing (Van der Stigchel and Theeuwes 2005), inhibition of return (Dodd et al. 2009; Klein and MacInnes 1999; Lupiáñez 2010), distractor inhibition (Stigchel and Theeuwes 2006), and flanker effects (Eriksen 1995), to name a few, that are highly significant but often quite small and range between 10-15 ms in response and in dwell time. It is impossible to capture these effects with low-precision eye trackers. Many of these effects are highly relevant to software engineering studies. But none of the current studies analyze such effects as there is currently no support to do this in current infrastructure. Note that this is still possible to do with high speed trackers if using short code snippets that fit on the screen, however it has been shown that the results from short snippets do not necessarily generalize to more realistic tasks (Abid et al. 2019).

Researchers have studied how eye curvature affects a task. These characteristics can only be discerned at a high sampling rate requiring the use of high-speed tracking. For example, the eye can be attracted to or repelled from a distractor as a function of temporal relationship between a target and a distractor (Stigchel and Theeuwes 2006). We have yet to determine if these issues impact real world programming behavior. Researchers can generally extract more information from high precision data such as pupillary activity (Duchowski et al. 2020a; Rayner 1998) and velocity measures that can help with saccade (Stigchel et al. 2010) and microsaccade analysis (Engbert and Kliegl 2003; Hafed and Clark 2002; Lowet et al. 2018). Microsaccades are miniature eye movements along with tremor and drift that are made during a fixation. They are typically found 1-2 times per second and have an amplitude of between 1'-25' (arcminute). Microsaccades have regained popularity recently and are being studied by eye tracking researchers to learn about the cognitive load (Kelleher and Hnin 2019) and task difficulty (Duchowski et al. 2020b). However, to correctly conduct microsaccade analysis, a 300 Hz or higher (500 Hz recommended) tracker is necessary to be confident in the velocity measures. Typically, oversampling of the data is used as an alternative but this is not recommended due to the artificial nature of the generated samples. Finally, with the introduction of multiple data collection streams such as studies that incorporate fMRI (Floyd et al. 2017), fNIRS (Fakhoury et al. 2018), EEG, or GSR with eye tracking, it is recommended to have high speed precision to align timing data.

In summary, we have only begun to start studying developers and cognition in software engineering using eye trackers. We have yet to learn from cognitive psychology and one of the ways to do this correctly is to have support for high-speed trackers in order to start collecting data correctly and making scientifically sound conclusions using realistic settings.

Another point of discussion is what theories support empirical analysis of studies run on such eye tracking infrastructure. Note that a detailed analysis and actually conducting an empirical study was not the scope of this paper. This paper is producing infrastructure that enables studies to be done. In order to show this in a feasibility analysis, we ran a short replication analysis (see Section 8 by collected data on prior tasks with a few participants and analyzed the results with Deja Vu and compared it to the syntactic categories of the original studies without Deja Vu. One can always refer to the mental models and theories in program comprehension (Storey 2006), however, this is best left to when a research study is designed. That was not the scope of this paper. We could come up with working theories to provide better insight into cognitive load and comprehension processes. However, this paper is not about finding cause/effect via an empirical study. It was merely showing that we can get additional syntactic categories (via a replication analysis) because now we support high speed tracking. We also want to note that sometimes theory hinders experiments (especially interdisciplinary ones such as the ones using eye tracking) as pointed out by Ko and Nelson in their award winning paper at ICER in 2018 (Nelson and Ko 2018). Even though their paper is on CS Education, the same principles still hold for general SE research.

5 The iTrace Infrastructure

Deja Vu leverages, and is now integrated into, the iTrace infrastructure (www.i-Trace.org) (Guarnera et al. 2018; Sharif and Maletic 2016b) to capture mouse and keyboard activity during an eye tracking study. To understand the role of Deja Vu it is necessary to be familiar with the architecture and workings of iTrace presented in this section.

5.1 iTrace Architecture

iTrace is eye tracking infrastructure that enables research studies within multiple types of software development environments. It was designed and built to support the software engineering community in conducting eye tracking experiments seamlessly within realistic developer environments i.e., IDEs. The infrastructure's design is modular featuring three key components, iTrace Core, iTrace Plugins (see Fig. 2), and an offline post processing application for gaze analysis called iTrace Toolkit (see Fig. 3). For a detailed low-level diagram on how iTrace works, we direct the reader to (Guarnera et al. 2018).

The Core provides a unified interface for managing supported eye tracking devices. Through this application eye trackers can be set up to calibrate or begin and end eye tracking data recording. All data generated by the eye trackers is first received by the Core which then makes quick decisions based on validity indicators whether the data is acceptable for use by other iTrace infrastructure applications (plugins). The Core also provides socket and websocket servers to allow for iTrace plugins to connect to the Core and receive gaze data for additional processing. In addition to gaze data, the socket communication also coordinates the start and stop of a recording session and subsequent plugin data processing as well as any output file storage locations for organizational purposes.

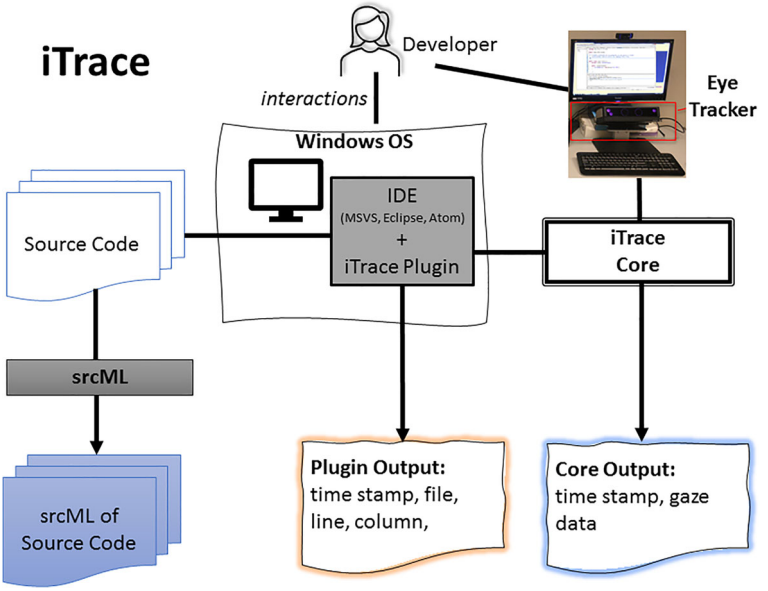


Fig. 2 iTrace Architecture Diagram. iTrace is composed of two main components: the core and an IDE plugin(s). The core interacts with the eye tracker and sends information to the plugin. Given the screen (x, y) coordinate of a gaze, the plugin determines the file, line, and column that maps to that gaze

Plugins for iTrace support applications such as Eclipse, Visual Studio, Atom, and the Google Chrome web browser to allow study participants to engage with standard development tools instead of simulated proxies. This allows for data collection to occur in a natural and realistic development environment. Plugins receive the screen coordinate location of a gaze via socket or websocket communication as well as a unique identifier from the Core. Using this information, each plugin performs real-time analysis to map a gaze to contextual information within the IDE or web browsing window. This mapping constitutes line and column positions within a visible source code editing window, IDE interface widgets, or HTML elements (with respect to Google Chrome) that fall under a participant's gaze. These contextual mappings are essential as study participants are free to manipulate the stimulus environment through scrolling, resizing, switching files or pages, searching, and other activities. Without any kind of context to associate with a gaze, combined with the volatile nature of the stimulus environment, it would be impossible to correctly determine what elements of the stimulus are actually viewed at a given moment in time.

Note that even eye tracking vendor software does not have support that iTrace provides. They (at best) cache a page apriori if it extends screen size and need to know in advance what participants will look at. This is not the case with iTrace. iTrace completely revolutionizes the way eye tracking studies are conducted in realistic settings.

All data collected from each eye tracking recording session is stored in XML files. The Core stores participant and study metadata, calibration information, details about the specific tracker used to record the data, and all the raw gaze data points (valid or invalid) received from the eye tracking device during the session. Each plugin records valid gaze points received by the Core and contextual information about the gaze location with the IDE or web browser environment. When a study is complete, the custom offline post processing

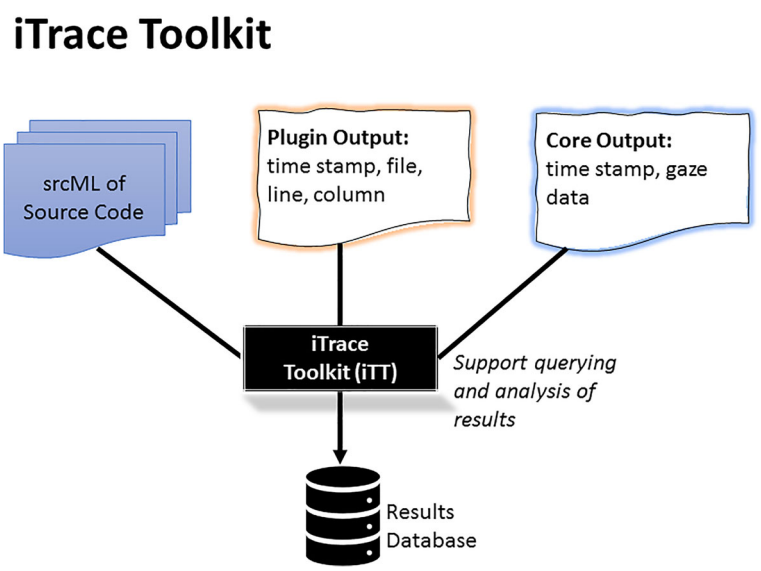


Fig. 3 The iTrace Toolkit fuses all the different data sources together into a multi-relation database (SQLite). This allows researchers to formulate queries and conduct analysis on the data. This is done as a post processing phase after replay. Fixations are computed, line/column positions and syntactic information are mapped to source code tokens via srcML

Toolkit provided by the iTrace infrastructure aggregates the data from all XML files. All study metadata and gaze data is collected into a unified Sqlite database where raw gaze data and plugin context information is joined using the aforementioned unique identifiers. Once all of the data is aggregated into the Sqlite database it can be queried using standard SQL commands or further analyzed using the post processing application.

5.2 iTrace Toolkit

The iTrace Toolkit (Fig. 3) is a post processing application written in C++ and QML. It performs two key analysis methods on the collected study data. The Toolkit makes use of the eye-tracking information gathered from iTrace Core, the contextual information gathered from the iTrace Plugin of choice, and a srcML archive file of the source code observed (if indeed source code was one of the artifacts being viewed). For artifacts that are not source code, the process is a little different and adhoc in nature and needs to be written specific to that artifact. For e.g., if the artifact is a custom built web application that uses the iTrace-Chrome plugin, the post processing will be very specific to the structure of that website. iTrace-Toolkit will need to be extended to support that specific website's data collection needs.

First, the Core and plugin data is loaded into the Toolkit. Multiple different recording sessions can be loaded in at once. Using srcML (Collard et al. 2013) in conjunction with the line and column information provided by the iTrace IDE plugins, all textual tokens and the syntactic context of each token within a source code document can be recovered and stored within the database for later querying. The related srcML archive file is loaded in, and all of the raw gaze data from the eye-tracker is matched to the contextual information to deduce what file, line number, column number, and token the participant examined. This

information is stored in a Sqlite database. The data model for this database is given in Fig. 4. As additional mapping to tokens is done, that information is also added to the database. This allows researchers to easily manage and analyze the data produced in each study.

The Toolkit supports three different fixation detection algorithms—Basic (Olsson 2007), I-VT, and I-DT (Salvucci and Goldberg 2000)—each with adjustable parameters (Andersson et al. 2017). All fixations identified are stored within the database and each fixation references the raw gaze collection that it represents. Other fixation filters can easily be added to the Toolkit as needed. The Toolkit also provides a way for a participant/researcher to query the loaded database for specific fixations; i.e., if the researcher wants to look at all fixations that focused on whitespace and happened before line 300. The queries' outputs can be saved in a variety of formats: SQL, TSV, JSON, and XML. This data can then be imported into the user's statistical package of choice for further qualitative and quantitative analysis.

5.3 Software Tasks in Research Studies

iTrace and Deja Vu are eye tracking infrastructure to support researchers in studying developers (i.e., their eye movements) while trying to understand software systems. In the context of software development, any program understanding task can be studied within a research study including debugging, bug localization, method summarization, concept location, feature location, tracing, etc. In other words, *iTrace Deja Vu is task agnostic* and supports data collection in the cases where IDE lookup times are slower than the eye tracker frequency. At this point in time, Deja Vu and iTrace do not support tasks that involve editing. The only exception to this is a specialized iTrace-Atom version with our collaborators that supports limited editing (Fakhoury et al. 2021) for a specific study that needed to be conducted.

Accurate automatic support for eye tracking in the presence of editing is a very difficult and currently open problem. However, the iTrace infrastructure and the way Deja Vu was architected forms a basis for addressing this challenge to support tracking full editing capabilities in the near future.

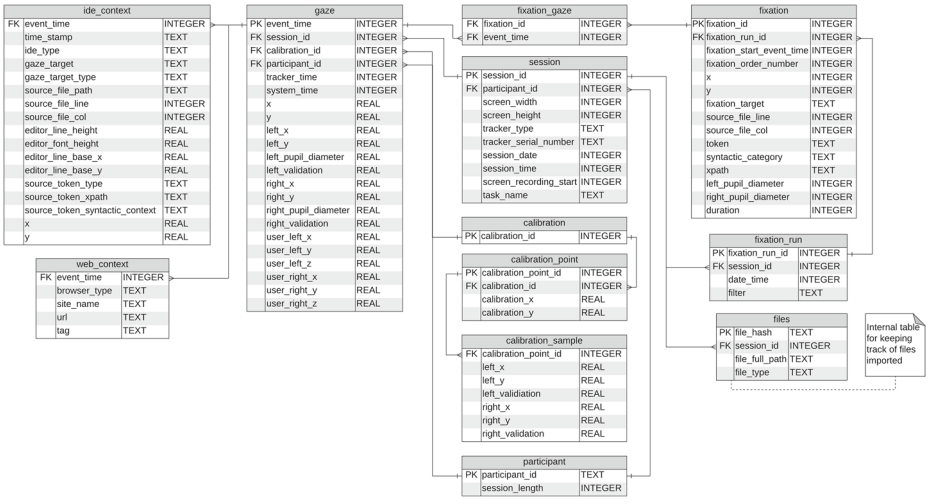


Fig. 4 An overveiw of the iTrace Toolkit Relational Database Schema

6 Realizing Deja Vu

The contextual information that iTrace provides is of great value. However, the overhead incurred by collecting this information in real time becomes problematic as the speed at which eye tracking devices are capable of transmitting data increases. To alleviate this issue and fully support high speed eye tracking while still collecting contextual stimulus environment information a new approach is required.

To address the problem, Deja Vu augments iTrace to allow all gaze analysis that occurs in real-time to be deferred to an offline post processing phase. We record all telemetry data (e.g., keyboard, mouse), along with eye tracker data, and time stamps. This requires Deja Vu to record all user interactions. A subsequent replay phase is used to synchronize each user action with respect to recorded gaze data. Hence, we are no longer is constrained by real-time performance requirements.

One method of implementing this is capturing the entire operating system after receiving each gaze during an eye tracking study session. After the study, each operating system state is loaded and all mappings are calculated. This is entirely accurate, however is not practical. It has very poor performance due to requiring copying the entirety of RAM to disk and may require introducing the complexity of a hypervisor.

Deja Vu takes an alternative approach. Only actions that get the environment to each state are recorded and stored. Practically, these are mainly human-computer interaction events—mouse movements and keyboard key state data. Other vital information includes the operating system state history, such as the exact position where a window pops up (in Windows, it depends on where it was previously opened). In these cases, a Deja Vu style approach needs to take measures to address this and ensure that replays are deterministic.

In the Deja Vu approach, the execution process is split into two steps. First, during an eye tracking study, the computer interaction data is collected in real time. After the eye tracking study session is completed, all the computer interactions can be replayed at some later time. This involves replaying the session on the same machine but at a slower frame rate. Since all data is timestamped this can be done without loss and in an accurate manner. Thus, the system/application calls to calculate the line, column in the file can be run without concern and in-depth analysis (of almost any type) can be performed during the replay. An overview of the two steps is shown in Figs. 5 and 6.

6.1 Recording Stage

During the recording phase (see Fig. 5), Deja Vu captures human-computer interaction data by recording mouse, and keyboard, along with the eye tracking gaze data. Mouse and keyboard events are captured using Win32 hooks. Hooking into operating system events is a feature of the Windows API and is done through the `SetWindowsHookEx` function. By using this function to hook into low level mouse and keyboard events, Deja Vu can capture these events before they are added into the OS input queue. If a study participant is typing code in an IDE, Deja Vu captures and saves each keystroke before the IDE even receives it. This capturing and saving step happens imperceptibly fast. Performing the capture this way allows for perfect accuracy and replays. Gaze data is collected by listening for broadcasted event data from iTrace-Core. As this data is collected, it is saved to disk in a CSV format. A sample of the recorded data is shown in Table 1. Each row is in the following format: event type, a 64-bit integer specifying the system time, and any data related to the event. This format contains all data necessary for replaying the user's computer interaction. Each recoded event type is shown in Table 2. `KeyDown` and `KeyUp` is used to represent keyboard key

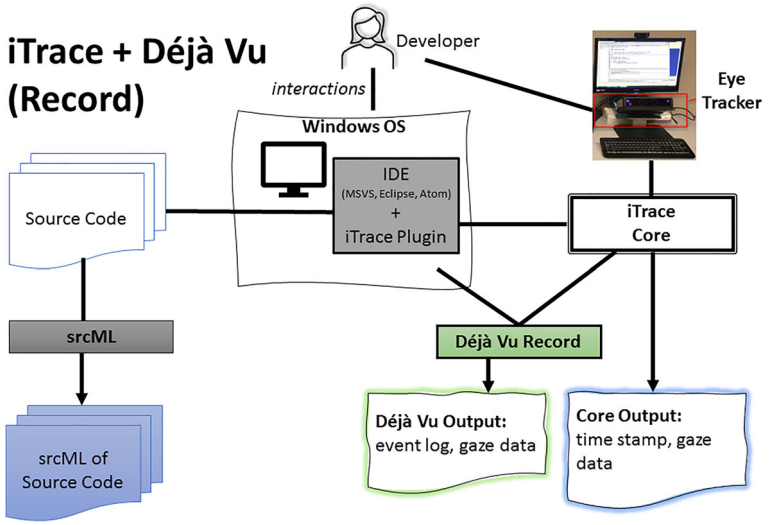


Fig. 5 Overview of the Record Stage. Deja Vu collects all interaction information of the developer along with gaze information, all of which is stored in an event log

state changes. A Windows virtual key code (which is the size of a byte) can store any keyboard key, including modifier keys such as shift or control. Each of the mouse buttons are explicitly stated as an event type. Forward and back refers to the buttons on the left side of a mouse (generally used for webpage navigation). `MouseMove` specifies the new absolute position on screen after the mouse has been moved. `MouseWheel` stores any scroll that happens with a value that specifies how much the mouse is scrolled. This event also collects touchpad scrolling on laptops. The gaze, `session_start`, and `session_end`

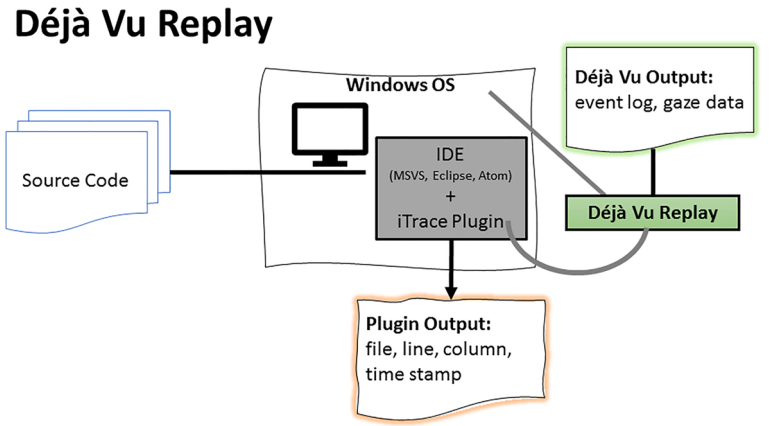


Fig. 6 Overview of the Replay Stage. Deja Vu replay takes the place of both the user and iTrace core. The event log is replayed back at a slower rate and the iTrace plugin (re)produces the file, line and column information in the same manner as iTrace without Deja Vu. This way all gaze points can be mapped correctly to a line and column

Table 1 Example of data collected during the recording phase. Some gazes omitted for brevity. The events are shown as they happen. In this case we have shown gaze, KeyDown, KeyUp, and MouseMove events

Event type	TimeStamp	Coordinates/codes
gaze	132277258033906585	314,769
KeyDown	132277258035886613	72
gaze	132277258037224389	336,790
gaze	132277258037601928	333,791
KeyDown	132277258037645064	73
gaze	132277258037758814	323,786
gaze	132277258037914237	333,794
gaze	132277258039069772	270,767
KeyUp	132277258039085245	72
KeyUp	132277258039090178	73
gaze	132277258039225920	276,771
gaze	132277258039755087	316,804
MouseMove	132277258055005185	391,823
MouseMove	132277258055085137	388,823

events are directly retrieved from iTrace Core. Gaze events store the x and y screen coordinate the participant's gaze at that time including validity codes, pupil diameter, and distance to screen. The `session_start` and `session_end` events are used by iTrace to mark the beginning and end of a study. These are primarily used to synchronize iTrace Core state with plugins.

6.2 Replaying Stage

During the replaying phase (see Fig. 6), Deja Vu reads in the CSV data produced during the recording stage and replays each event by creating mouse and keyboard events using the Windows API. Specifically, the `mouse_event` and `keyboard_event` functions are used to synthesize button presses, mouse motions, and mouse scrolls. In addition, Deja Vu also replays all gazes and emulates the communications protocol used by iTrace Core. This allows existing iTrace plugins to connect to Deja Vu to receive gaze data and perform analysis during the replay. In essence, Deja Vu works as proxy for iTrace Core.

All events are replayed synchronously. To slow down the replay, Deja Vu pauses in between events it produces. This pause provides time for connected plugins to process received gaze data. Therefore, time in between events must be carefully considered to give ample time for each connected plugin to perform its analysis. There are multiple possible algorithms for choosing the time to wait in between replaying each event. Deja Vu implements three such methods so researchers can choose whichever fits their needs the best. Refer to Figs. 7 and 8 for a graphical illustration of how the delays work.

6.2.1 Fixed Pause Delay

The time waited after each event is a fixed amount of time based on the type of the event. Plugin processing time for each type of event received will vary depending on the type of analysis performed. Generally, most processing is done after gaze events. Other events, such as mouse movements, may not need any analysis (depending on the researcher's needs). In

Table 2 Each event type (which can originate from the mouse, keyboard, or eye tracker during a Deja Vu recording) that appears in the CSV format. Each of the event types is timestamped. The additional data description includes the main components of each event type

Event type	Additional data
Keyboard	
KeyDown	Virtual Key Code
KeyUp	Virtual Key Code
Mouse	
LeftMouseDown	
LeftMouseUp	
RightMouseDown	
RightMouseUp	
MiddleMouseDown	
MiddleMouseUp	
ForwardMouseDown	
ForwardMouseUp	
BackMouseDown	
BackMouseUp	
MouseMove	(x,y) coordinates
MouseWheel	Mouse scroll amount (positive for an upward scroll and negative for a downward)
Eye Tracker	
Gaze	event_id (used for indexing iTrace Core output) and averaged gaze coordinates (x,y) for both eyes
Study Session	
session_start	The time when study session started
session_end	The time when study session ended

these cases, processing-heavy events (such as gazes) can be set to have a greater pause time than processing-light events (such as mouse movements).

The primary drawback to this mode is that choosing a good pause length is difficult. Gaze processing latencies are not necessarily easy to predict and outliers are possible. However, via some trial runs a suitable duration could be determined and used. If the experiment is short and fairly simple the fixed paused approach should work well. A visual illustration of this replay method is shown in Fig. 7.

6.2.2 Proportional Delay

The time after each event is proportional to what it is during the recording. For example, Deja Vu can set to replay everything at exactly half the speed of recording. This mode is useful for visualizations. Screen recordings performed during the replay stage can easily be sped up by the same factor as the replay is slowed down. Using this method, the sped-up recording of the replay is identical to a recording of the session. See Fig. 7.

The drawback to this mode is that it is impossible to set a minimum time between events. If processing is to happen after each keypress, nothing stops events from being generated during replay at a very high frequency. During recording, the user can have press several

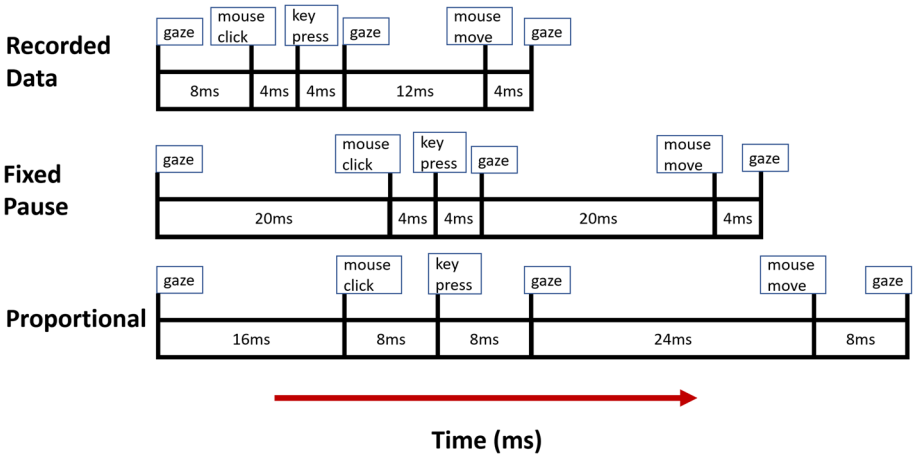


Fig. 7 Illustration of how the fixed and proportional pause delay mechanisms work

keys on the keyboard, generating key presses nearly simultaneously. It is possible that one might want to do some analysis after each keystroke. If the analysis takes 20 ms, it is impossible to set a minimum pause after each keystroke. Even if slowed down by a factor of 10, when a user presses two keys within less than 2 ms, there is not have enough time for analysis. However, this is not an issue for gaze data as eye trackers typically generate readings quite uniformly, making it possible to reinforce a minimum pause time in between gaze events.

6.2.3 Bidirectional Delay

In the third method, after gaze events, Deja Vu waits indefinitely for a reply/acknowledgement from each connected plugin. This reply marks that the plugin is finished doing processing and is ready to process more data. Communication between Deja Vu and plugins happens bidirectionally. Events that do not need to be waited on are followed by a short fixed-length pause. From a technical point of view, this is the best pausing method. The difficulty of choosing a good fixed-pause length is alleviated. Pauses after gaze events are always correct. No extra time is wasted as padding for the highest-latency lookup/processing cases.

The primary drawback to this method is that it requires modification to the existing components in the iTrace infrastructure. Plugins need to be modified to reply a ready-signal (over the TCP socket connection between the plugin and Deja Vu) in response to events that require confirmation. In addition, there is the potential added overhead due to the additional layer of communication that needs to take place. A visual illustration of this replay method is shown in Fig. 8.

6.2.4 Theoretical Foundations in Replay Pausing Strategies

In this section, we provide a summary and a theoretical foundation for analyzing the slowdown between different pausing strategies. Let c_e indicate the number of times the event e is encountered. Let p_e be the fixed pause length (ms) assigned to event e. Let t be the initial recording length (ms), and t' be the replay length (ms).

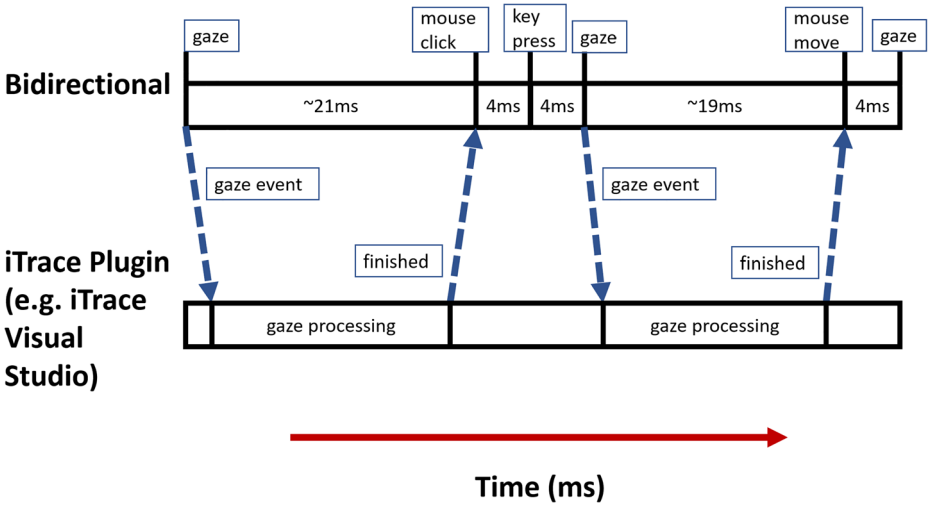


Fig. 8 Illustration of how bidirectional delay and replay works. In this example, the same recorded data log file is used as in Fig. 7. The pause after each gaze in bidirectional delay could be a variable length (depending on how long the plugin takes to do its computation)

Fixed Pause Replay:

$$t' = \sum_{e \in \text{all\_event\_types}} c_e \cdot p_e$$

Proportional Replay: Let s be the scale chosen for the proportional replay. Then:

$$t' = t \cdot s$$

Bidirectional Replay: Let m be the average time it takes each plugin to finish processing each gaze, and reply to Deja Vu. Then:

$$t' = c_{\text{gaze}} \cdot m + \sum_{e \in (\text{all\_event\_types} - \{\text{gaze}\})} c_e \cdot p_e$$

6.3 Practicalities of Implementation

While developing the Deja Vu tool, we ran into several non-obvious problems and issues, some of which are challenging to completely address. Each can be addressed in several different ways and we present our solutions to these below. We believe that these challenges generalize for the implementation of tools using a similar approach to that of Deja Vu. Hence, these descriptions may prove useful to other researchers.

6.3.1 Solving Non-Deterministic Window Placements

The initial window position is non-deterministic on MS Windows making it difficult to start the replay from the exact same position. During a replay, the position where a window opens up can be different from where it opened during recording. To address this, Deja Vu forces

each window opened during recording and replay to open in a single predefined location on the screen. In Deja Vu, this predefined location is the top left corner of the screen. This is done by frequently iterating over each window handle and checking if any new handles appear.

In theory, this method is not entirely accurate for every application, since the application can move its window without human interaction. However, we have not found an application that does this to date in the use cases Deja Vu is used for. To maintain integrity of replays, researchers performing studies need to still consider this issue and avoid using applications in studies that have this behavior.

6.3.2 Restoring Initial Interface State

A slight change in interface layouts between runs can cause replay to become out of sync with the events that happen during recording. This can happen in a butterfly-effect style. To address this, researchers need to be careful choosing a replicable initial state between runs.

Currently, ensuring that the initial interface state is the same is performed manually by the researcher. Many IDE's, such as MS Visual Studio, support saving and restoring UI layouts (e.g., through a simple hotkey). Saving a layout before running a study and restoring it before performing a replay is one method of ensuring initial interface state in an IDE with adjustable element sizes will remain consistent.

6.3.3 Relative or Absolute Mouse Positioning

The MS Windows API allows for two methods of capturing and moving the mouse: by the absolute value (directly specifying mouse position with x and y coordinates) or by relative value (changes the x and y coordinate of the mouse) (Microsoft 2018). Deja Vu uses absolute mouse values.

The advantage of absolute values over relative value is that replays are more robust. Moving the mouse accidentally during a replay using relative values will cause all subsequent mouse usages to be off. Absolute mouse values solve this issue by automatically locking the mouse back where it should be after each mouse move event.

6.3.4 Replaying Mouse Double Clicks

This challenge was discovered after the dataset for our replication analysis (given in Section 8.4) was collected. If a sequence containing a double-click (two mouse clicks in quick succession, with a pause in between) is slowed down enough, it will result in replaying two separate clicks (and not a double-click). To address this problem, Deja Vu replay looks ahead into the event log and replaces any double-clicks (which would become two separate clicks after slowing the replay down), with a double-click event.

6.3.5 Race Conditions

This was another challenge discovered after the dataset collected for our replication analysis. Due to multiple input sources (e.g., mouse, keyboard, and sockets) collecting data concurrently a race condition was possible while writing to the log. We have identified and fixed this race condition that occasionally corrupted event log data entries.

6.4 Using Deja Vu with iTrace

As far as we are aware, this is the first attempt at supporting high-speed trackers for software engineering-based studies that work on complex artifacts tracked within an IDE. Deja Vu will typically be used in the following manner.

Let's assume that a researcher is looking to investigate how developers understand class hierarchies (using a high-speed 1000 Hz eye tracker). Before the study, the researcher chooses a suitable real-world code base and the questions a study participant is to answer. The code base is imported into a project file in an IDE that has iTrace plugin support (such as Visual Studio or Eclipse). The layout is saved. During the study, a participant is put in front of the computer. The eye tracker is calibrated for the participant. The IDE is opened, and the layout is restored. Eye tracking is started in iTrace-Core with Deja Vu Recording enabled.

During the study, the participant performs the assigned set of tasks. They have the freedom to interact with the IDE, OS, and any applications if they so desire (for example, opening a web browser to access StackOverflow). During replay, all computer events will be replayed. If the participant highlighted text and pressed Ctrl+C to copy the text, the same sequence of events would be replayed during the Replay phase (the same text would be highlighted, the Ctrl+C keypress would be replayed causing the highlighted text to be copied). While users can interact with any application, the applications that support gaze-token lookups will depend on the iTrace plugins that are running. If a study participant opens Firefox (for which no iTrace Plugin exists yet), gaze data will still be collected, however the gaze {x,y} coordinates will not be mapped to specific tokens or areas-of-interest on screen. Once the participant is finished, the tracking and recording are stopped. The Recording phase is complete.

At some point after the study is completed, the researcher begins the replay phase. Deja Vu Replay is opened in Core. Analysis plugins are enabled in the IDE and are connected to Core. The IDE layout is restored again. Deja Vu Replay is started in Core. Everything that happened during the study is now replayed slowly on the computer. Analysis is being performed in the background via iTrace. Once it is finished, the researcher can collect the data from the plugins and analyze it in any statistical package. In this use case, they can investigate how the developer navigated the class hierarchies and what they looked at before they completed the task.

6.5 Example Usage Scenario

To perform a study with iTrace Deja Vu, first a code project and associated IDE (which has an iTrace plugin) is chosen. Next, iTrace Core is started, calibrated and setup. The IDE is to be brought into a reproducible initial position (typically this is the IDE taking up the full screen space i.e., maximized). The session is setup in Core, and recording with Deja Vu is enabled. The study participant is then invited to perform the instructed task (e.g., in the case of the replication study we conduct in Section 7, they perform a bug localization task). As seen in Fig. 5, Deja Vu records computer and gaze interactions of the study participant.

Once the participant is finished with the task (see Section 5.3), the recording is stopped. We recommend starting and stopping tracking before each task is performed to have a clean data recording for each task. For example, as a researcher, if you setup your study to have four tasks per participant, you will start and stop tracking before each task within iTrace with the Deja Vu option selected. After all the data collection for all the tasks is complete and after the participant has left, the researcher can now collect detailed gaze data with Deja

Vu Replay. In order to do this, first, the initial IDE position is restored. Deja Vu Replay is selected in iTrace Core, and a previously recorded session can be selected to replay. During this time, detailed Plugin information, such as file along with line and column gaze information, is collected (as seen in Fig. 6). Once replaying is finished for all tasks and the detailed plugin data is collected, further analysis can be performed.

To perform analysis, the project code file is converted into its srcML representation. iTrace Toolkit then combines the core and plugin files and along with srcML information is able to map tokens to gaze coordinates. As shown in Fig. 3, iTrace Toolkit generates a database which can be queried for eye tracking data. iTrace Toolkit also supports various fixation event detection algorithms that are run on the raw gaze data and exported by the researcher to perform further statistical analysis. Note that iTrace Toolkit is not a statistical package. iTrace Toolkit is a post processing tool to combine core and Plugin files, generate fixations, and map the fixations to source code tokens. In addition, it can filter the data on specific criteria via queries from the user interface. See Section 5.2 for more details.

It is important to note that iTrace and iTrace Deja Vu are task agnostic. They do not directly support software engineering tasks such as bug localization or code summarization. The infrastructure provides a method for researchers to collect eye tracking data on software engineering tasks. iTrace has been used in previous eye tracking research studies to better understand tasks such as code summarization (Abid et al. 2019; Saddler et al. 2020), code review (Park and Sharif 2021), program comprehension (Peterson et al. 2019a, b), software traceability (Sharif et al. 2016a), and bug fixing (Kevic et al. 2015, 2017). The task to be studied depends on the researcher's objective and the research questions they seek to address in their study. iTrace and iTrace Deja Vu facilitate collection of (high speed) eye tracking data within the IDE while developers are working on software tasks. More information on iTrace along with video tutorials are available at https://www.i-trace.org/.

7 Evaluating the Deja Vu Approach

The evaluation of our approach is conducted via two experiments. Experiment 1 evaluates the initial problem by looking at two typical data analysis plugin implementations (iTrace Visual Studio, iTrace Eclipse, and iTrace Atom) to show data loss and degradation with high-speed trackers. Experiment 2 evaluates Deja Vu to determine whether it can recreate all gazes that were produced during the recording phase. This is done in the context of a sample eye tracking experiment.

Experiment participants are assigned to one of two groups each denoted by the identifier K and L respectively. Table 3 shows the eye trackers and data rates used by each group. Each tracker for Group K is connected to a 64-bit MS Windows 10 desktop with a 3.6 GHz Intel i7-7700 CPU, mechanical hard disk drive, 8 GB of RAM, and two 24-inch LCD displays running at a 1920x1200 resolution. Group L eye trackers ran on two separate machines. The

Table 3 Participant groups and the eye tracking devices and data rates used

Participant groups	Eye tracker model	Data rate
K	Gazepoint GP3 HD	60 Hz
	Tobii Pro X3-120	120 Hz
L	Gazepoint GP3 HD	150 Hz
	Tobii TX300	300 Hz

machine connected to the Tobii TX300 used the tracker's built-in 23" monitor running at 1920x1080 resolution on a Windows 10 desktop with 3.5 GHz Intel i7-7800X, a solid-state drive, and 32 GB of RAM. The Gazepoint GP3-HD was connected to a 27" LCD panel running at 1920x1080 resolution, on a Windows 10 laptop with 2.7 GHz Intel i7-6820HQ CPU, a solid-state drive, and 32 GB of RAM.

7.1 Experiment 1: Data Collection Without Deja Vu

This experiment evaluates the initial problem: *Does the latency for real time data collection make it infeasible to map eye gaze to semantic elements at high-speed tracking frequencies?* To determine this, all the IDE plugins that iTrace currently supports (iTrace Visual Studio, iTrace Eclipse, and iTrace Atom) are evaluated to determine the impact on data rate limitations when performing real-time gaze analysis.

7.1.1 Experiment Setup

The plugins are instrumented to collect timings from the functions related to real-time line, column lookup analysis. The evaluation is run on multiple hardware configurations (Group K and Group L) to provide a less biased performance measure. Each plugin environment (Eclipse, Visual Studio, and Atom) is also stressed with an increasing number of open source-code tabs to identify potential implementation specific overhead.

7.1.2 Data Collection

A process diagram for the first experiment is shown in Fig. 9. An eye tracking study is set up in iTrace. The IDE gaze analysis plugins are connected to iTrace Core. The study participant is instructed to have no files open in the IDE and gaze at the screen for 5 s. Then they are instructed to open a file and look at it for 5 s. This is repeated until 4 files are opened inside the IDE. Each IDE plugin is modified before the study to collect implementation and environment API performance data. In the iTrace-VisualStudio plugin, this is done using the C# Stopwatch API. Elapsed times for each call to the gaze analysis functionality within the plugin is stored in memory and written out to a file at the end of a recording session. For the iTrace-Eclipse plugin, API performance data is collected using the `System.nanoTime()` API and calculating the difference between the start and stop time for each call to the gaze analysis function. This timing data is stored in memory and written out to a file at the end of a recording session.

7.1.3 Results Showing Loss of Data

The data collection process is repeated for each plugin with 0-4 open tabs and the results are presented in Fig. 10. iTrace Eclipse provides an optimized API for translating screen coordinates to the file, line, and column at that screen coordinate. Each lookup in eclipse takes 0.015 s. 0.015 s is equivalent to approximately 66 Hz. This means that real time data collection can only happen for eye trackers operating at 66 Hz or less.

Visual Studio does not provide an optimized API for converting screen coordinates to file, line, and column data. For this reason, the lookup timings for iTrace Visual Studio plugin implementation scales linearly with respect to the number of tabs open (due to needing to iterate over all open files). When a single tab is open, the plugin is able to support up to 166hz trackers. However, typically developers have more than a single tab open and any

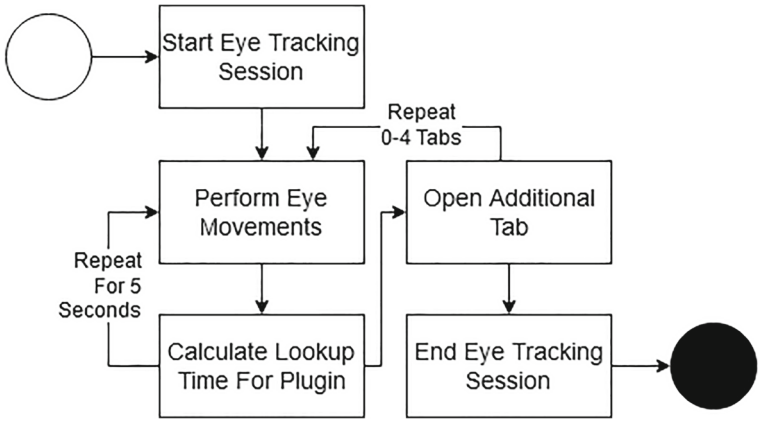


Fig. 9 Process diagram for data collection in Experiment 1 (Data Collection without Deja Vu)

number of tabs open above two will not even support 60 Hz. However, both eye tracker speeds estimates are liberal because they do not consider outliers. Figure 11 shows the raw timing data in the Visual Studio plugin.

iTrace Atom is implemented as a package for Atom, a text editor that is built on the Electron framework. Electron allows developing desktop applications using web technologies by running code using the Chromium rendering engine. Because of this, iTrace Atom has access to an optimized DOM screen coordinate to text element API. This allows iTrace Atom to perform lookups at an average of 0.223 ms per lookup, regardless of number of tabs open. Because the lookup is already very fast, Deja Vu has less potential to be useful, unless running experiments on weaker hardware or using an eye tracker that collects data at higher rate than 4500 Hz.

In conclusion, real time data collection in IDE's (with the exception of iTrace Atom) using the iTrace eye tracking infrastructure is infeasible for high speed eye trackers (running above 60 Hz).

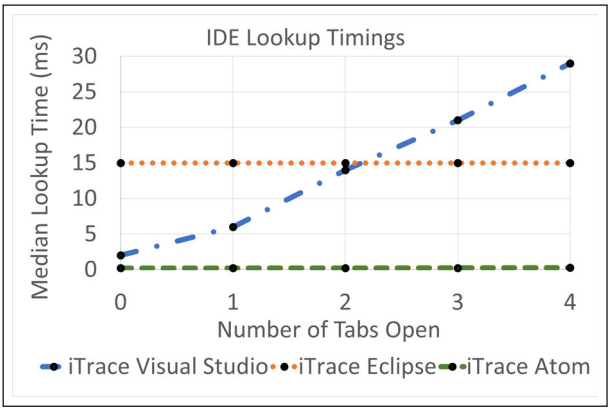


Fig. 10 IDE screen coordinate to (file, line, column) lookup times in the Visual Studio, Eclipse, and Atom iTrace Plugins. iTrace Atom took an average of 0.223 ms per lookup

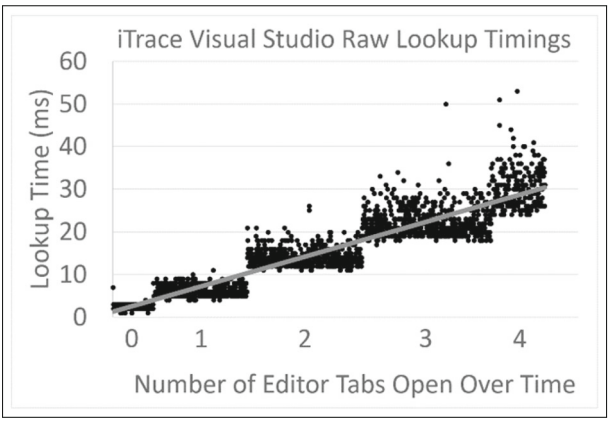


Fig. 11 Raw timing data from Visual Studio. A trendline showing the linear growth is displayed as the number of tabs open increase

7.2 Experiment 2: Is Deja Vu an Effective Solution?

In this section, we describe a simple experiment on two tasks with the goal of showing that Deja Vu is able to keep up with high speed eye trackers to collect and recreate all gazes that occurred during an experiment.

7.2.1 Experiment Setup

The simulated eye tracking experiment consists of two tasks and each task is repeated twice per participant with variations in the data rate of the eye tracking device. The first task requires participants to read out loud each method name and return type from the source code file `SvgExporter.java` taken from the JHotDraw8 project. This file contains 1,166 lines of code and 42 methods. While this task is straight forward, it will require active engagement with the source code while ensuring a long enough recording duration, minimize cognitive fatigue, and require scrolling.

The second task requires participants to summarize three methods in the `SvgExporter.java` file selected randomly from a collection of the eight largest methods (in terms of lines of code). Participants perform the summarization out loud and the selected methods are not repeated by the participant on the second run of the task when the eye tracker data rate is changed. This task is designed to engage the participant and represent a more advanced eye tracking study task.

In this study, Deja Vu was setup to use a fixed-pause delay strategy during replay as it provided us with enough time to map what we needed. Refer to Fig. 7 for a graphical illustration of the fixed pause delay mechanism. If bi-directional delay was used it would just complete in a different total time (not worse than fixed pause delay, since we chose a time that is longer than needed to compute the gaze mappings). Proportional would be slower than bidirectional and fixed-pause because everything would be slowed down equally (compared to the other two where we can choose to slow down only the gazes but replay interactions faster)

7.2.2 Data Collection

A process diagram for this experiment is shown in Fig. 12. Participant data captured during the simulated eye tracking study consists of a set of data comprised of: 1) an iTrace-Core data file representing all valid data points generated by the eye tracking device; 2) an iTrace-Eclipse or iTrace-VisualStudio plugin data file containing all data received from iTrace-Core and processed in real-time; and 3) a Deja Vu recording file storing all mouse and keyboard interactions and gaze positions sent from iTrace-Core. Each participant generates two sets of data representing tasks recorded using different eye tracking data rates. Audio recordings of participant activities are also saved via a cellular phone audio recording application. To determine the effectiveness of Deja Vu's data collection, all gaze data present in the plugin and Deja Vu output files is compared against the valid raw data points stored and transmitted to each application by iTrace-Core. Gaze data is uniquely identified by an event id value and is used to determine any data loss (e.g. data transmitted, but not received by the plugin or Deja Vu).

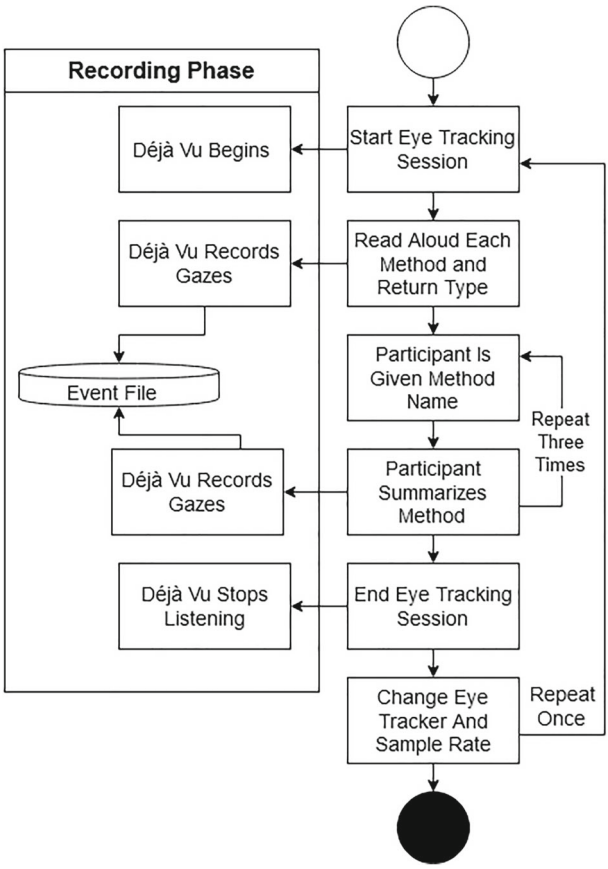


Fig. 12 Process diagram for data collection in Experiment 2 (Data collection with Deja Vu)

7.2.3 Results

Table 4 shows the data rates of eye tracking devices and the amount of valid data successfully captured by iTrace-Core, Deja Vu, and the iTrace plugins for Eclipse and Visual Studio. From the table we see that an eye tracking device running at 60 Hz, tends to moderately tax the real-time analysis component of the iTrace plugins. As the data rate increases to 120 Hz, real-time analysis in the plugins falls behind and nearly half of the data transmitted to the plugins for analysis is lost as plugins cannot keep up with the faster data generation rate of the eye trackers. It is interesting to note that in nearly all cases, the data rate of the eye tracker poses no issue for Deja Vu with nearly 100% of the data sent from iTrace-Core is also recorded by Deja Vu along with participant mouse and keyboard interactions. Note that iTrace-Atom was not compared as the IDE lookup time for the (file, line, column) were much smaller and will be at least as good and most likely better (see Fig. 10) than iTrace-VisualStudio and iTrace-Eclipse.

Note that we cannot claim a link between the data rate and data loss in terms of percentages. The plugin behaves very undeterministicaly in how it drops gazes when moving from 150 Hz to 300 Hz as seen from Table 4. The main point to note is that Deja Vu takes care of the data loss. That said, we did use two different machine configurations to collect the 150 Hz data and another machine to collect the 300 Hz data. The 150 Hz dataset was collected on a laptop fitted with the GazePoint tracker whereas a dedicated machine for the TX-300 tracker which comes incorporated into a monitor was used for the 300 Hz dataset. It is not straightforward to just move the GazePoint tracker on the TX-300 machine. We do not believe collected the data on two different configurations invalidates the fact that there is data loss regardless.

7.2.4 Limitations

We are not implying that the high-speed support for trackers will be needed for every study. Similarly, not every study needs to be an eye tracking study (there needs to be a specific reason). Likewise, not every eye tracking study will need to be done using a high-speed tracker. However, as we pointed out in Section 3, there are specific use cases for when a high speed tracker is needed. In those cases, Deja Vu will significantly improve data collection

Table 4 Raw gaze datapoints collected during study. The percent shows the data loss. The K samples were collected in the Visual Studio plugin. The L samples (last four) were collected in the Eclipse plugin

Sample	Data rate	Core data	Deja Vu	Plugin
K1	60 Hz	22629	22629 (0%)	15817 (30%)
K2	60 Hz	21333	21333 (0%)	19833 (7%)
K3	60 Hz	28392	28392 (0%)	16306 (43%)
K1	120 Hz	41999	41999 (0%)	23424 (44%)
K2	120 Hz	48405	48405 (0%)	26087 (47%)
K3	120 Hz	67024	67023 (<1%)	35786 (47%)
L1	150 Hz	52506	52506 (0%)	25047 (52%)
L2	150 Hz	48090	48088 (<1%)	15858 (67%)
L1	300 Hz	138442	138441 (<1%)	79967 (42%)
L2	300 Hz	106674	106674 (0%)	68852 (35%)

without any data loss or incorrect mapping. Closer investigation of the instances where Deja Vu did not manage to capture all data points transmitted from iTrace-Core revealed a bug in the research prototype. Occasionally, Deja Vu can corrupt a data entry which we believe to be caused by a race condition on the output file resource. While this can explain the missing data points given Deja Vu's generally consistent performance, we still consider these data points to have been lost in Table 4 to avoid under-reporting the findings. This issue has since been fixed.

8 Replication Analysis

We present a replication analysis of four tasks taken from two prior eye tracking studies. The first two tasks consist of bug fixing tasks from Kevic et al. (2015) where participants need to find the location of a bug and propose a potential solution. These two bug fixing tasks are both on the JabRef system. The last two consist of code summarization tasks from Saddler et al. (2020) where participants are asked to provide a summary of one method and one class. Both summarization tasks are on the Eclipse project.

During the study, participants only have access to the code files present in the project corresponding to their current task, a text file describing the current task or bug. Each participant completes the study in the same order using the same stimuli. Participants' eye movement data is collected using the iTrace-Eclipse plugin.

8.1 Research Questions

The research questions we seek to address in this replication analysis are given below:

- RQ 1: What additional syntactic categories does Deja Vu provide over prior work's results?
- RQ 2: What further analysis can be done with the additional syntactic categories that Deja Vu provides?

The motivation behind RQ 1 is to show the added insight that Deja Vu provides by comparing the results of prior work with the current study's results. The motivation behind RQ 2 is to provide examples of some further analysis that can be done with the additional syntactic categories Deja Vu provides. Note that the goal of Section 8 is not to completely replicate the prior studies but to show via replication analysis on two different types of tasks that Deja Vu works in these settings and generates additional, and useful, information for analysis. The researcher can then take this information and use it towards some functional goal related to their specific research question.

8.2 Tasks

For the bug fixing tasks, JabRef is selected as the subject system. JabRef is a desktop application for managing bibliographic databases with many import and export formats. JabRef version 1.8.1 is used in this study. Only two of the three original bug fixing tasks are selected due to time constraints labeled in this study as Task 1 and Task 2. In both of these tasks, the bug descriptions submitted to the JabRef project are added to the text file describing the current task. Participants are given a maximum of 20 minutes to fix the bug to avoid fatigue. This is also done in the original study. Further details of the tasks can be viewed in the original study (Kevic et al. 2015). In this paper we name our tasks as follows: Task

1 refers to Bug 2 in the original paper and Task 2 refers to Bug 4 in the original paper (Kevic et al. 2015).

For the code summarization tasks, Eclipse is selected as the subject system. Eclipse is an IDE used primarily for Java programming. Eclipse version 4.2 is used in this study. Due to time constraints, only the two summarization tasks about code elements in Eclipse in the original study are used in this replication labeled in this study as Task 3 and Task 4. Participants are given either a method name they needed to summarize or a class name they needed to summarize. Once they navigated to the code element, read the code, they provided their summary for their task. Further details of the tasks can be viewed in the original study (Saddler et al. 2020). In this paper we name our tasks as follows: Task 3 refers to T1 in the original paper and Task 4 refers to T2 in the original paper (Saddler et al. 2020). See Table 5 for an overview of the tasks.

8.3 Eye Tracking Apparatus

Two different eye tracking setups are used at the different universities. At UNL, the study is conducted with a Tobii TX300 set to capture eye gaze data at 120 Hz with an accuracy of 0.5 degrees. The built-in 23-inch, 1920px by 1080px monitor is used. At KSU, the study is conducted with using the Tobii X3-120 eye tracker set to capture eye gaze data at 120 Hz with an accuracy of 0.5 degrees. A laptop with a 15.6", 1920px by 1080px monitor is used with this eye tracker. Deja Vu is run with a fixed-pause strategy. See Fig. 7 for a graphical illustration of the fixed-pause replay strategy.

8.4 Data Collection Process

There are two sets of participants from the two collaborating universities that participated in this study to collect this proof of concept data and includes the first four authors of the paper plus two additional members from their respective research labs. No one from outside the current research team is used to collect this data. This is important to note because this is not a typical replication study. It is a proof of concept replication analysis of two prior studies also done by the some of the authors. In order to do the replication with Deja Vu, the data needed to be collected with Deja Vu Record. We used our own research team for this evaluation. However, none of these participants had done the prior study nor were they familiar with the study tasks apriori. This was done to simulate a real study environment.

On the day of the study, participants came into the research lab (alone due to COVID-19 restrictions at the time). Next, participants are asked to find the location of a described

Table 5 An overview of the tasks used in the replication study analysis

Task No.	Type	System	Description
Task 1	Bug Fix	JabRef	No comma added to separate keywords
Task 2	Bug Fix	JabRef	Failure to import big numbers Acrobat Launch fails on Win98
Task 3	Summarization	Eclipse	Summarize Method `core.databinding.binding.dispose`
Task 4	Summarization	Eclipse	Summarize Class `swt.SWTError`

bug and find a solution for two bug fixing tasks (see Table 5). After they finished proposing a solution, participants rated their confidence of their solution's correctness and their perceived difficulty of the task. Next, they are asked to summarize a method and a class from the Eclipse project. After they finished summarizing a code element, participants rated their confidence of their summary and their perceived difficulty of the task. We do not use the confidence ratings in this paper however we did this to keep the protocol as similar as possible with what was done in the prior studies. Eye tracker calibration is done at the start of each of the four tasks to ensure the best eye tracking accuracy during the tasks (also done in prior studies). Participants are also allowed a short break between tasks if needed to reduce fatigue over the entire study.

The participants had access only to the Eclipse IDE which containes the code files of the entire subject system of the task, JabRef for the bug fixing tasks and Eclipse for the summarization tasks, and a text file containing the task instructions and bug description for bug fixing tasks. The demographics of the previous studies participants were very similar with the group of participants we used in this replication analysis study. None of them were complete novices and all had similar programming experience and experience in bug fixing and code summarization tasks.

As stated earlier, we do not use these results for a research study goal. We do however try to keep the environment and questions the same as done in the previous two studies.

8.5 Replication Analysis Results

This section presents results of the replication analysis conducted based on each of the two research questions.

8.5.1 RQ1 Results: Additional Syntactic Categories Deja Vu Provides over Prior Work

When comparing the benefits of using the latest version of iTrace (at http://www.i-trace.org/) alongside Deja Vu with previous versions of iTrace (which is still available via an archive site at https://github.com/SERESLab/iTrace-Archive), we look at the number of unique syntactic categories that are able to be extracted from the eye tracking data from both the original dataset and the data collected from this replication. We directly compare these distinct categories for each task independently, and find that the current version of iTrace alongside Deja Vu consistently provides finer grained syntactic categories.

The overlap between the syntactic categories in these datasets provides further insight into the type of information that the current version of iTrace provides. See Table 6 for a list of distinct syntactic categories in each dataset. It is clear that Deja Vu provides more

Table 6 Comparing the number of distinct syntactic categories between the original studies' analysis and replication analysis on the same set of tasks

Task	Task type	Original dataset	Replication dataset
Task 1	Bug Fixing	34	48
Task 2	Bug Fixing	34	41
Task 3	Summarization	7	23
Task 4	Summarization	7	24

syntactic category types compared to the original set. We do not believe this is due to technical skill bias as the populations studied were very similar both in terms of programming experience and experience in bug fixes and summarization.

This is due to three reasons, a) iTrace has been completely rewritten to make optimal use of srcML which provides extended mapping to all source tokens b) a higher speed tracker is used which gives more samples per second and c) no gaze data are lost when using a high speed tracker greater than 60 Hz. Note that this replication study is done with a 120 Hz tracker to show the practicality of Deja Vu while the prior studies are done with 60 Hz trackers. We use the number of fixations to determine the counts for the syntactic categories in Table 6. We direct the reader to additional metrics (Sharafi et al. 2015a) that can be used in future emipircal studies.

In the summarization tasks, most of the syntactic categories in the replication dataset do not have a clear one-to-one relationship with the categories in the original dataset. For example, the argument list category in the replication dataset can be assigned the categories, Method Use, Method Declare, or Variable Declare in the original dataset. Vice versa, the Method Use category in the original dataset can be assigned to the argument list, parameter list, name, or specifier categories in the current replication dataset. Another important detail from the original dataset is that the Outer Class and Inner Class declarations are differentiated into separate categories while they are not in the replicated dataset. While at first this appears to be a limitation, these categories can easily be obtained using the fine-grained categories and syntactic hierarchy provided by srcML to derive the aforementioned higher-level categories in a post processing step via iTrace Toolkit.

In the bug fixing tasks, most syntactic categories in the datasets are shared. Syntactic categories relating to keywords and other low level units, e.g., for, if, or comment, have a one to one relationship between the two datasets. However, certain categories in the original dataset are at a higher level than the replicated dataset. Method call, method declaration, and variable declaration in the original dataset are composed of several unique category types in the replicated dataset such as argument list, parameter list, or name. Overall, these two category sets are much more similar to each other but there are still some high level aggregation that occurs within the original dataset.

While fine-grain syntactic category information is useful, it can be hard to comprehend and analyze findings with the larger number of categories. The advantage is that aggregation of these syntactic categories is always possible to derive categories at higher levels of granularity with the additional syntactic context srcML provides. The same however is not true in reverse, if only high level syntactic categories are used as in our original dataset, we are unable to produce token categories at a finer level of granularity. For sake of an example, in the original dataset a mapping can be made to a method use, but it is not possible to discern if the gaze fell on a argument list or the name of the method. In comparison to the fine-grain syntactic data from the replication dataset, this ambiguity does not exist. Providing a clear syntactic hierarchy down to the lowest level permits researchers to perform analyses at nearly any granularity to identify the source code content developers view while solving a task.

8.5.2 RQ2 Results: Further Analysis with Deja Vu

One large benefit of the current version of iTrace and Deja Vu is that the syntactic categories are generated post-hoc allowing syntactic categories to be constructed at varying levels of specificity. The previous versions generated these categories on the fly meaning that they

are baked into the data at any given point. Once the session is done, there is no way of going back to a more specific level.

To address this research question, we provide some examples of further analysis that can be done with the varying levels of specificity of syntactic categories available. One example of analysis is to investigate how participants fixate on more specific structures. We examined the method signatures and how participants view the method name and the method parameters. In the previous two studies, these sub-components of the method signature are not recorded as they are not the focus of the study and as such this analysis cannot be done with the previous data. While the previous version of iTrace is capable of collecting data like this, it required the researcher to know this is the use case before the study is done.

With the current approach, since the lowest mapping is collected, we can drill up or down the abstract syntax tree to get any level of abstraction we require using srcML. We can easily find the sub-components described earlier by using a simple XPath expression (as the srcML format is XML). The method name can be retrieved with:

- `//src:function/src:name`
- `//src:constructor/src:name`

Method parameters can be retrieved with:

- `//src:function/src:parameter_list`
- `//src:constructor/src:parameter_list`

The method signature is retrieved by collecting all fixations inside function and constructor that are not immediately followed by a block element.

Using this analysis, there are two main observations from the fixation counts and percentages seen in Table 7. First, more fixation attention inside the method signature is spent on the method name than the method parameters in every task. This indicates that for both bug fixing and code summarization, the method name is the more important (or at least is given a lot more attention) to participants than the parameters of a method. Lastly, we see that the task has a large influence on the distribution of fixations inside the method signature. The two bug localization and fixing tasks are relatively similar but are different to the two summarization tasks. In Task 3, a summarization of a single method, the emphasis on the method name and parameters is reduced along with much fewer total fixations on a method signature. However, in Task 4, a summarization of a single class, the method parameters have a higher percentage of fixation attention in the method signature than any other task. This clearly indicates that the task also plays a big role in how developers spend time examining different elements. All of these observations help developers answer very specific questions in eye tracking studies using Deja Vu.

Table 7 List of fixation metrics on the method name and method parameters in the method signature

	Task 1	Task 2	Task 3	Task 4
Total Fixations	15756	9148	2655	2825
Total Fixations on Method Signature	873	629	318	755
Avg Percentage of Method Signature Fixations on Method Name	41.58%	47.57%	29.01%	45.18%
Avg Percentage of Method Signature Fixations on Method Parameters	24.52	28.20%	14.25%	37.82%

8.6 Threats to Validity

One threat to validity is the measurement of the unique number of syntactic categories in the fixation data as a metric for usefulness. If there is added redundancy in a category, reduction is always possible. The syntactic categories in the previous studies can be seen as a subset of the larger category set present in the replicated data which the replicated data's categories can be reduced down into. New information cannot be added easily to the smaller number of distinct categories in the original datasets.

In studies where specificity provided by the updated version of iTrace and Deja Vu is not needed, the added specificity can be reduced to a more useful subset of categories for the research goals of any studies.

The small amount of participants used in this replication study was for demonstration purposes only to show that we are able to provide much finer grained mapping and not loose any gazes with high speed trackers. With more participants and more fixations we would have a larger amount of distinct categories (not less) meaning that the results would only become more pronounced.

9 Conclusions and Future Work

The paper presents a novel solution to a fundamental technological problem for studying software developers using high-speed, high-quality eye trackers while working in a natural and familiar development environment on production sized software systems. A methodology and tool—Deja Vu—is introduced that captures all relevant user and system interactions for later replay of a user session within a study. The replay allows for accurate mapping of user gaze points on the entire stimulus being viewed i.e., the specific elements of source code or other software artifacts. This overcomes serious real-time limitations posed in mapping screen coordinates to line and column in a given file. To add to our prior work (Zyrianov et al. 2020), we provide additional timing experiments in iTrace-Atom and conduct a replication analysis of two prior studies by collecting data with Deja Vu to provide evidence of the richer syntactic categories that are provided with the Deja Vu record and replay approach. iTrace and Deja Vu directly facilitate software engineering researchers in studying how developers read software during various types of tasks such as general comprehension, bug fixes, and refactoring to name a few. It also allows the software engineering research community to apply additional eye tracking analyses (such as microsaccade analyses) from cognitive psychology research (that require high-speed tracker output) on text understanding. We believe this will lead to a much deeper understanding of how developers read source code and solve problems which is a complex mixture of many factors.

As part of future work, the Deja Vu approach will be extended to support eye tracking studies in the presence of editing source code. Supporting editing is a very difficult engineering problem and more research and tests are needed to support this type of data collection in an accurate manner. Recently, we released a version of iTrace-Atom that supports editing, however this is restricted to just Atom and is a first attempt at supporting editing (Fakhoury et al. 2021) in eye tracking studies. Supporting editing in the iTrace infrastructure with high speed trackers is a bigger challenge that we plan to work on in future iTrace releases.

Another avenue for future work includes adding support for other popular IDEs. We have had many requests for the supporting Atom and so we prioritized that first. iTrace is designed in a way that supports ease of extension. We foresee members of the community contributing support for other plugins and call on the community to do so.

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments and suggestions. This work has been partly funded by the US NSF under Grant Numbers CNS 17-30181, CNS 18-55753, and CCF 18-55756

Declarations

Competing Interests The authors have no competing interests with the editorial board members or editors. The authors have no other competing interests, financial or otherwise.

The work is supported in part by a grant from the US National Science Foundation CNS 17- 30181/30307.

References

Abid NJ, Sharif B, Dragan N, Alrasheed H, Maletic JI (2019) Developer reading behavior while summarizing java methods: size and context matters. In: Atlee JM, Bultan T, Whittle J (eds) Proceedings of the 41st international conference on software engineering, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019. IEEE/ACM, pp 384–395. https://doi.org/10.1109/ICSE.2019.00052

Andersson R, Nyström M, Holmqvist K (2010) Sampling frequency and eye-tracking measures: how speed affects durations, latencies, and more. J Eye Mov Res 3(3). https://doi.org/10.16910/jemr.3.3.6

Andersson R, Larsson L, Holmqvist K, Stridh M, Nyström M (2017) One algorithm to rule them all? An evaluation and discussion of ten eye movement event-detection algorithms. Behav Res Methods 49:616–637

Bao L, Ye D, Xing Z, Xia X, Wang X (2015) Activityspace: a remembrance framework to support interapplication information needs. In: Cohen MB, Grunske L, Whalen M (eds) 30th IEEE/ACM international conference on automated software engineering, ASE 2015, Lincoln, NE, USA, November 9–13, 2015. IEEE Computer Society, pp 864–869. https://doi.org/10.1109/ASE.2015.90

Bao L, Xing Z, Xia X, Lo D, Hassan AE (2018) Inference of development activities from interaction with uninstrumented applications. Empir Softw Eng 23(3):1313–1351. https://doi.org/10.1007/s10664-017-9547-8

Bernal-Cárdenas C, Cooper N, Moran K, Chaparro O, Marcus A, Poshyvanyk D (2020) Translating video recordings of mobile app usages into replayable scenarios. In: Rothermel G, Bae D (eds) ICSE '20: 42nd international conference on software engineering, Seoul, South Korea, 27 June–19 July, 2020. ACM, pp 309–321. https://doi.org/10.1145/3377811.3380328

Brooks R (1983) Towards a theory of the comprehension of computer programs. Int J Man-Mach Stud 18(6):543–554. https://doi.org/10.1016/S0020-7373(83)80031-5. https://www.sciencedirect.com/science/article/pii/S0020737383800315

Brown NCC, AlTadmri A, Sentance S, Kölling M (2018) Blackbox, five years on: an evaluation of a large-scale programming data collection project. In: Malmi L, Korhonen A, McCartney R, Petersen A (eds) Proceedings of the 2018 ACM conference on international computing education research, ICER 2018, Espoo, Finland, August 13–15, 2018. ACM, pp 196–204. https://doi.org/10.1145/3230977.3230991

Burg B, Bailey R, Ko AJ, Ernst MD (2013) Interactive record/replay for web application debugging. In: Izadi S, Quigley AJ, Poupyrev I, Igarashi T (eds) The 26th annual ACM symposium on user interface software and technology, UIST'13, St. Andrews, United Kingdom, October 8–11, 2013. ACM, pp 473–484. https://doi.org/10.1145/2501988.2502050

Collard ML, Decker MJ, Maletic J (2013) srcML: an infrastructure for the exploration, analysis, and manipulation of source code: a tool demonstration. In: 2013 IEEE International conference on software maintenance, pp 516–519. https://doi.org/10.1109/ICSM.2013.85

Dodd MD, der Stigchel SV, Hollingworth A (2009) Novelty is not always the best policy: inhibition of return and facilitation of return as a function of visual task. Psychol Sci 20(3):333–339. https://doi.org/10.1111/j.1467-9280.2009.02294.x. pMID: 19222812

Duchowski A (2007) Eye tracking methodology: theory and practice. https://doi.org/10.1007/978-1-84628-609-4

Duchowski A, Krejtz K, Gehrer N, Bafna T, Baekgaard P (2020a) The low/high index of pupillary activity. In: 2020 CHI conference on human factors in computing systems, pp 1–12. https://doi.org/10.1145/3313831.3376394

Duchowski AT, Krejtz K, Żurawska J, House DH (2020b) Using microsaccades to estimate task difficulty during visual search of layered surfaces. IEEE Trans Visual Comput Graph 26(9):2904–2918. https://doi.org/10.1109/TVCG.2019.2901881

Engbert R, Kliegl R (2003) Microsaccades uncover the orientation of covert attention. Vis Res 43(9):1035–1045. https://doi.org/10.1016/S0042-6989(03)00084-1. https://www.sciencedirect.com/science/article/pii/S0042698903000841

Eriksen C (1995) The flankers task and response competition: a useful tool for investigating a variety of cognitive problems. Vis Cogn 2:101–118

Fakhoury S, Ma Y, Arnaoudova V, Adesope O (2018) The effect of poor source code lexicon and readability on developers' cognitive load. In: Proceedings of the 26th conference on program comprehension, ICPC '18. ACM, New York, pp 286–296. https://doi.org/10.1145/3196321.3196347. http://doi.acm.org/10.1145/3196321.3196347

Fakhoury S, Roy D, Pines H, Cleveland T, Peterson CS, Arnaoudova V, Sharif B, Maletic J (2021) gazel: supporting source code edits in eye-tracking studies. In: 2021 IEEE/ACM 43rd international conference on software engineering: companion proceedings (ICSE-Companion), pp 69–72. https://doi.org/10.1109/ICSE-Companion52605.2021.00038

Floyd B, Santander T, Weimer W (2017) Decoding the representation of code in the brain: an fmri study of code review and expertise. In: 2017 IEEE/ACM 39th international conference on software engineering (ICSE), pp 175–186. https://doi.org/10.1109/ICSE.2017.24

Goldberg JH, Stimson MJ, Lewenstein M, Scott N, Wichansky AM (2002) Eye tracking in web search tasks: design implications. In: Proceedings of the 2002 symposium on eye tracking research & applications, ETRA '02. ACM, New York, pp 51–58. https://doi.org/10.1145/507072.507082. http://doi.acm.org/10.1145/507072.507082

Guarnera DT, Bryant CA, Mishra A, Maletic JI, Sharif B (2018) itrace: eye tracking infrastructure for development environments. In: Proceedings of the 2018 ACM symposium on eye tracking research & applications. ACM, p 105

Guo J, Li S, Lou J, Yang Z, Liu T (2019) Sara: self-replay augmented record and replay for android in industrial cases. In: Zhang D, Møller A (eds) Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2019, Beijing, China, July 15–19, 2019. ACM, pp 90–100. https://doi.org/10.1145/3293882.3330557

Hafed ZM, Clark JJ (2002) Microsaccades as an overt measure of covert attention shifts. Vis Res 42(22):2533–2545. https://doi.org/10.1016/S0042-6989(02)00263-8. https://www.sciencedirect.com/science/article/pii/S0042698902002638

Holmqvist K, Andersson R (2017) Eye-tracking: a comprehensive guide to methods, paradigms and measures. Oxford University Press, Oxford

Just M, Carpenter P (1980) A theory of reading: from eye fixations to comprehension. Psychol Rev 87(4):329–54

Kelleher C, Hnin W (2019) Predicting cognitive load in future code puzzles. Association for Computing Machinery, New York, pp 1–12. https://doi.org/10.1145/3290605.3300487

Kersten M, Murphy GC (2006) Using task context to improve programmer productivity. In: Young M, Devanbu PT (eds) Proceedings of the 14th ACM SIGSOFT international symposium on foundations of software engineering, FSE 2006, Portland, Oregon, USA, November 5–11, 2006. ACM, pp 1–11. https://doi.org/10.1145/1181775.1181777

Kevic K, Walters BM, Shaffer TR, Sharif B, Fritz T, Shepherd DC (2015) Tracing software developers eyes and interactions for change tasks. In: Proceedings of the 10th joint meeting of the european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering

Kevic K, Walters B, Shaffer T, Sharif B, Shepherd DC, Fritz T (2017) Eye gaze and interaction contexts for change tasks—observations and potential. J Syst Softw 128:252–266. https://doi.org/10.1016/j.jss.2016.03.030

Klein RM, MacInnes WJ (1999) Inhibition of return is a foraging facilitator in visual search. Psychol Sci 10(4):346–352. https://doi.org/10.1111/1467-9280.00166

Letovsky S (1987) Cognitive processes in program comprehension. J Syst Softw 7(4):325–339. https://doi.org/10.1016/0164-1212(87)90032-X. https://www.sciencedirect.com/science/article/pii/016412128790032X

Lowet E, Gomes B, Srinivasan K, Zhou H, Desimone R (2018) Enhanced neural processing by covert attention only during microsaccades directed toward the attended stimulus. Neuron 99:207–214.e3

Lupiáñez J (2010) Inhibition of return. Scholarpedia 3:17–34

Microsoft (2018) mouse_event function (winuser.h). https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-mouse_event

Minelli R, Mocci A, Lanza M, Kobayashi T (2014) Quantifying program comprehension with interaction data. In: 2014 14th International conference on quality software, pp 276–285. https://doi.org/10.1109/QSIC.2014.11

Minelli R, Mocci A, Lanza M (2015) I know what you did last summer—an investigation of how developers spend their time. In: 2015 IEEE 23rd International conference on program comprehension, pp 25–35. https://doi.org/10.1109/ICPC.2015.12

Minelli R, Mocci A, Lanza M (2016) Measuring navigation efficiency in the ide. p to be published. https://doi.org/10.1109/IWESEP.2016.11

Nelson GL, Ko AJ (2018) On use of theory in computing education research. In: Malmi L, Korhonen A, McCartney R, Petersen A (eds) Proceedings of the 2018 ACM conference on international computing education research, ICER 2018, Espoo, Finland, August 13–15, 2018. ACM, pp 31–39. https://doi.org/10.1145/3230977.3230992

Niño IJ, de la Ossa B, Gil JA, Sahuquillo J, Pont A (2005) CARENA: a tool to capture and replay web navigation sessions. In: Al-Shaer E, Pras A, Owezarski P (eds) Third IEEE/IFIP workshop on end-to-end monitoring techniques and services, e2EMON 2005, 15th May 2005. IEEE Computer Society, Nice, pp 127–141. https://doi.org/10.1109/E2EMON.2005.1564474

Obaidellah U, Al Haek M, Cheng PCH (2018) A survey on the usage of eye-tracking in computer programming. ACM Comput Surv 51(1):5:1–5:58. https://doi.org/10.1145/3145904. http://doi.acm.org/10.1145/3145904

Olsson P (2007) Real-time and offline filters for eye tracking. KTH Electrical Engineering, Stockholm

Park K, Sharif B (2021) Assessing perceived sentiment in pull requests with emoji: evidence from tools and developer eye movements. In: 6th IEEE/ACM international workshop on emotion awareness in software engineering, SEmotion@ICSE 2021, Madrid, Spain, May 31, 2021. IEEE, pp 1–6. https://doi.org/10.1109/SEmotion52567.2021.00009

Pennington N (1987) Stimulus structures and mental representations in expert comprehension of computer programs. Cogn Psychol 19(3):295–341. https://doi.org/10.1016/0010-0285(87)90007-7. https://www.sciencedirect.com/science/article/pii/0010028587900077

Peterson CS, Abid NJ, Bryant CA, Maletic JI, Sharif B (2019a) Factors influencing dwell time during source code reading: a large-scale replication experiment. In: Krejtz K, Sharif B (eds) Proceedings of the 11th ACM symposium on eye tracking research & applications, ETRA 2019, Denver, CO, USA, June 25–28, 2019. ACM, pp 38:1–38:4. https://doi.org/10.1145/3314111.3319833

Peterson CS, Saddler JA, Halavick NM, Sharif B (2019b) A gaze-based exploratory study on the information seeking behavior of developers on stack overflow. In: Mandryk RL, Brewster SA, Hancock M, Fitzpatrick G, Cox AL, Kostakos V, Perry M (eds) Extended abstracts of the 2019 CHI conference on human factors in computing systems, CHI 2019, Glasgow, Scotland, UK, May 04–09, 2019. ACM. https://doi.org/10.1145/3290607.3312801

Ramler R, Gattringer M, Pichler J (2020) Live replay of screen videos: Automatically executing real applications as shown in recordings. In: Kontogiannis K, Khomh F, Chatzigeorgiou A, Fokaefs M, Zhou M (eds) 27th IEEE international conference on software analysis, evolution and reengineering, SANER 2020, London, ON, Canada, February 18–21, 2020. IEEE, pp 664–665. https://doi.org/10.1109/SANER48275.2020.9054833

Rayner K (1978) Eye movements in reading and information processing. Psychol Bull 85(3):618–660

Rayner K (1998) Eye movements in reading and information processing: 20 years of research. Psychol Bull 124(3):372–422

Rist RS (1986) Plans in programming: definition, demonstration, and development. In: Papers presented at the first workshop on empirical studies of programmers on empirical studies of programmers. Ablex Publishing Corp., USA, pp 28–47

Saddler JA, Peterson CS, Sama S, Nagaraj S, Baysal O, Guerrouj L, Sharif B (2020) Studying developer reading behavior on stack overflow during api summarization tasks. In: 2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 195–205

Salvucci DD, Goldberg JH (2000) Identifying fixations and saccades in eye-tracking protocols. In: Proceedings of the 2000 symposium on eye tracking research & applications, ETRA '00. ACM, New York, pp 71–78. https://doi.org/10.1145/355017.355028. http://doi.acm.org/10.1145/355017.355028

Sharafi Z, Shaffer T, Sharif B, Guéhéneuc Y (2015a) Eye-tracking metrics in software engineering. In: Sun J, Reddy YR, Bahulkar A, Pasala A (eds) 2015 Asia-Pacific software engineering conference, APSEC 2015, New Delhi, India, December 1–4, 2015. IEEE Computer Society, pp 96–103. https://doi.org/10.1109/APSEC.2015.53

Sharafi Z, Soh Z, Guéhéneuc YG (2015b) A systematic literature review on the usage of eye-tracking in software engineering. Inf Softw Technol (IST)

Sharafi Z, Sharif B, Guéhéneuc Y, Begel A, Bednarik R, Crosby ME (2020) A practical guide on conducting eye tracking studies in software engineering. Empir Softw Eng 25(5):3128–3174. https://doi.org/10.1007/s10664-020-09829-4

Sharif B, Maletic J (2016a) itrace: overcoming the limitations of short code examples in eye tracking experiments. In: 2016 IEEE International conference on software maintenance and evolution, ICSME 2016, Raleigh, NC, USA, October 2–7, 2016. IEEE Computer Society, p 647. https://doi.org/10.1109/ICSME.2016.61

Sharif B, Maletic J (2016b) itrace: overcoming the limitations of short code examples in eye tracking experiments. In: 2016 IEEE International conference on software maintenance and evolution (ICSME), pp 647–647. https://doi.org/10.1109/ICSME.2016.61

Sharif B, Meinken J, Shaffer T, Kagdi H (2016a) Eye movements in software traceability link recovery. Empir Softw Eng 1–40. https://doi.org/10.1007/s10664-016-9486-9

Sharif B, Shaffer T, Wise JL, Maletic JI (2016b) Tracking developers' eyes in the IDE. IEEE Softw 33(3):105–108. https://doi.org/10.1109/MS.2016.84

Sharif B, Peterson C, Guarnera D, Bryant C, Buchanan Z, Zyrianov V, Maletic J (2019) Practical eye tracking with itrace. In: 2019 IEEE/ACM 6th international workshop on eye movements in programming (EMIP), pp 41–42. https://doi.org/10.1109/EMIP.2019.00015

Soloway E, Ehrlich K (1984) Empirical studies of programming knowledge. Software Engineering. IEEE Trans SE 10:595–609. https://doi.org/10.1109/TSE.1984.5010283

Stigchel S, Theeuwes J (2006) Our eyes deviate away from a location where a distractor is expected to appear. Exp Brain Res. Experimentelle Hirnforschung Expérimentation Cérébrale 169:338–49. https://doi.org/10.1007/s00221-005-0147-2

Stigchel S, Mills M, Dodd M (2010) Shift and deviate: Saccades reveal that shifts of covert attention evoked by trained spatial stimuli are obligatory. Atten Percept Psychophys 72:1244–50. https://doi.org/10.3758/APP.72.5.1244

Storey MD (2006) Theories, tools and research methods in program comprehension: past, present and future. Softw Qual J 14(3):187–208. https://doi.org/10.1007/s11219-006-9216-4

Sun Y, Chen D, Jiao W, Huang G (2014) An online education approach using web operation record and replay techniques. In: IEEE 38th Annual computer software and applications conference, COMPSAC 2014, Vasteras, Sweden, July 21–25, 2014. IEEE Computer Society, pp 456–465. https://doi.org/10.1109/COMPSAC.2014.68

Sun Y, Chen D, Xin C, Jiao W (2015) Automating repetitive tasks on web-based ides via an editable and reusable capture-replay technique. In: Ahamed SI, Chang CK, Chu WC, Crnkovic I, Hsiung P, Huang G, Yang J (eds) 39th IEEE annual computer software and applications conference, COMPSAC 2015, Taichung, Taiwan, July 1–5, 2015, vol 2. IEEE Computer Society, pp 666–675. https://doi.org/10.1109/COMPSAC.2015.12

Sun J, Zhang S, Huang S, Hui Z (2018) Design and application of a sikuli based capture-replay tool. In: 2018 IEEE International conference on software quality, reliability and security companion, QRS companion 2018, Lisbon, Portugal, July 16–20, 2018. IEEE, pp 42–44. https://doi.org/10.1109/QRS-C.2018.00021

Van der Stigchel S, Theeuwes J (2005) The influence of attending to multiple locations on eye movements. Vis Res 45(15):1921–1927. https://doi.org/10.1016/j.visres.2005.02.002. https://www.sciencedirect.com/science/article/pii/S0042698905000945

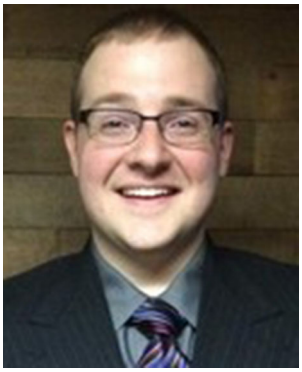
Von Mayrhauser A, Vans A (1995) Program comprehension during software maintenance and evolution. Computer 28(8):44–55. https://doi.org/10.1109/2.402076

Yan F, Qi Z, Xia M, Liu X (2018) Efficient and deterministic replay for web-enabled android apps. In: Chaudron M, Crnkovic I, Chechik M, Harman M (eds) Proceedings of the 40th international conference on software engineering: companion proceedings, ICSE 2018, Gothenburg, Sweden, May 27–June 03, 2018. ACM, pp 329–330. https://doi.org/10.1145/3183440.3194994

Zyrianov V, Guarnera DT, Peterson CS, Sharif B, Maletic JI (2020) Automated recording and semantics-aware replaying of high-speed eye tracking and interaction data to support cognitive studies of software engineering tasks. In: IEEE International conference on software maintenance and evolution, ICSME 2020, Adelaide, Australia, September 28–October 2, 2020. IEEE, pp 464–475. https://doi.org/10.1109/ICSME46990.2020.00051

Vlas Zyrianov is a second year PhD student at University of Illinois Urbana-Champaign, Champaign, IL. Prior to this he worked under the supervision of Dr. Jonathan Maletic at Kent State University doing his Bachelor of Science in Computer Science. His research interests are in machine learning, computer vision, compilers, and eye tracking.

Cole S. Peterson graduated with her Masters in Computer Science from the University of Nebraska at Lincoln in 2020 under the supervision of Dr. Bonita Sharif. She current works as a software engineer at a local company in Lincoln. Her research interests are in software engineering, eye tracking, polyglot programming studies, and program comprehension.



Drew T. Guarnera is an Assistant Professor of Computer Science at the College of Wooster in Wooster, Ohio (USA). He received his M.S. and B.S. in Computer Science from The University of Akron (USA) and is working to complete his Ph.D. in Computer Science from Kent State University (USA). His research focuses on software engineering in the areas of program comprehension, software evolution, mining software repositories, and the application of eye tracking in a software engineering context. He has served as the Proceedings Chair for ICPC 2016 and Web Chair for ICSME 2019. He also worked on the NSF funded iTrace infrastructure (https://www.i-trace.org/) as lead developer and as a contributor for the srcML infrastructure (https://www.srcml.org/).

Joshua Behler is a graduate student and Research Assistant at Kent State University, Kent Ohio USA, working under the supervision of Dr. Jonathan Maletic. Joshua received his B.S. in Computer Science from Kent State in 2021, and plans to receive his M.S. in 2023. He currently works on both the iTrace and srcML projects.

Praxis Weston is a software engineer at the Management Council Ohio Education Computer Network writing software for Ohio Public Schools. They graduated from Kent State University in 2021. While attending Kent State they participated in eye tracking research under Dr. Jonathan Maletic for their last 3 semesters and over the summers. Praxis contributed to iTrace Toolkit.

Bonita Sharif is an Associate Professor in the School of Computing at University of Nebraska at Lincoln (UNL), Lincoln, Nebraska USA. She received her Ph.D. in 2010 and MS in 2003 in Computer Science from Kent State University, U.S.A and B.S. in Computer Science from Cyprus College, Nicosia Cyprus. Her research interests are in eye tracking related to software engineering, empirical software engineering, program comprehension, emotional awareness, software traceability, and software visualization to support maintenance of large systems. She serves on numerous program committees including ICSE, ASE, ESEC/FSE, ICSME, VISSOFT, SANER, and ICPC. She served as general chair of VISSOFT 2016 and ETRA 2018 and 2019. She served as program chair for ICSME 2019 late breaking track and ICPC 2023 technical track. She is also the Steering Committee Chair for the ACM Symposium on Eye Tracking Research and Applications. Sharif is a recipient of the NSF CAREER award and the NSF CRI award related to empowering software engineering with eye tracking. She also received the NCWIT Undergraduate Student Mentoring award in 2016. She directs the Software Engineering Research and Empirical Studies Lab at UNL.

Jonathan I. Maletic Ph.D. is Professor in the Department of Computer Science at Kent State University, Kent Ohio USA. He received the Ph.D. and M.S., both in Computer Science, from Wayne State University in 1995 and 1989 respectively. His research interests are centered on software evolution, with a focus on the comprehension, static analysis, exploration, and manipulation of large-scale software systems. Prof. Maletic has authored over 150 refereed publications. For of his publications have received Most Influential Paper Awards. Prof. Maletic is regularly funded by the US National Science Foundation (NSF) and has averaged approximately 150K USD annually in external funding over the past 15 years. He has served on numerous program committees including, but not limited to, the International Conference on Software Maintenance & Evolution (ICSME), the International Conference on Automated Software Engineering (ASE), the International Conference on Program Comprehension (ICPC), and the Working Conference on Mining Software Repositories (MSR). He was general chair of ICSME 2019, program chair of ICPC 2016 and ICSM 2012. Prof. Maletic has graduated 22 doctoral students, 18 of whom hold an academic position. Six of these individuals have been awarded US NSF funding and two of these have received NSF CAREER awards.

Affiliations

Vlas Zyrianov[1] · Cole S. Peterson[2] · Drew T. Guarnera[3] · Joshua Behler[4] · Praxis Weston[4] · Bonita Sharif[2] · Jonathan I. Maletic[4]

Vlas Zyrianov
vlasz2@illinois.edu

Cole S. Peterson
Cole.Scott.Peterson@huskers.unl.edu

Drew T. Guarnera
dguarnera@wooster.edu; dguarner@kent.edu

Joshua Behler
jbehler1@kent.edu

Praxis Weston
gweston2@kent.edu

Bonita Sharif
bsharif@unl.edu

1 Department of Computer Science, University of Illinois at Urbana-Champaign, Champaign, IL, USA

2 School of Computing, University of Nebraska-Lincoln, Lincoln, NE, USA

3 Department of Computer Science, The College of Wooster, Wooster, OH, USA

4 Department of Computer Science, Kent State University, Kent, OH, USA