Software for an eye tracking device enabling analysis of a student's interaction with program code

Aleksandra Mitrović*, Mladen Vidović*, Ivan Radosavljević*, Miloš Mladenović*, Goran Savić*, Milan Segedinac*, Zora Konjović**

* University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Serbia

** Singidunum University, Novi Sad, Serbia

aleksandramitrovic@uns.ac.rs, mladenvidovic@uns.ac.rs, ivanradosavljevic@uns.ac.rs, milosml@outlook.com, savicg@uns.ac.rs, milansegedinac@uns.ac.rs, zkonjovic@singidunum.ac.rs

Abstract*—Evaluating a student's programming knowledge is not always precise or objective. Students may give a correct answer to a question without understanding the underlying code. One approach to solving this problem is to track the student's gaze during the examination by using an eye tracking device. Data acquired from the eye tracker can show if students are actually looking at the code they are explaining by providing the x and y coordinates of their gaze. These coordinates can then be mapped to the code displayed on the screen, allowing the comparison between the code the students were looking at and the code which they were expected to look at, thereby confirming or refuting their knowledge. This paper presents a solution to the aforementioned problem. The solution includes establishing communication with the Gazepoint eye tracking device, using a separately created Java library. Coordinates received from the device are transformed using the iTrace Eclipse plugin, modified to support tracking gazes across multiple files within a project in a single Eclipse session. The transformed data is then subjected to additional processing and filtering, during which redundant data is eliminated, such as all gazes that were flagged as non-code gazes, as well as attributes which were deemed irrelevant to the current project, such as the raw coordinates of the gaze, or pupil diameters. The median filter was also applied to the data, to reduce noise caused partially by the hardware itself, and partially by chaotic eye movements. The final output data is a sequence of gazes, each consisting of the line and column of the code that was watched, the name of the file containing the code, as well as the duration of the gaze. The output is then written to a JSON file that can be used later in the process of evaluating students' knowledge.

I. Introduction

Traditional knowledge assessment of a programming student is mostly subjective and sometimes an unreliable process. This process can be a lengthy one and usually yields unsatisfactory results.

Most of the time a valid code that returns correct results is not enough to prove a student's aptitude. A student must be able to identify key elements of the solution and to be able to explain them. Any discrepancies between identified key elements and the provided explanations can be a sign of a student's lack of knowledge.

Two common approaches to programming knowledge assessment are objective testing and performance-based assessment. Objective testing methodologies, such as multiple-choice questions rely on mostly theoretical knowledge. Although they are objective, cost-effective and quick, they are not viable to test a student's ability to produce functioning software. That issue can be solved by using performance-based assessment methodologies.

Homework programming assignments require students to create larger scale projects within a specified timeframe. They are valuable because of their similarity to real working conditions. Evaluating homework assignments requires experienced staff and a larger time investment to examine larger projects. These larger assignments are vulnerable to plagiarism, due to their unsupervised nature.

Examinations, in the form of short tasks, are a way to gauge a student's ability to generate smaller sections of code in a supervised manner, to avoid plagiarism. This approach, however, makes it more difficult to assess a student's ability to operate in real life environments and produce larger scale projects.

Charettes are assignments carried out during fixed-length supervised sessions on a regular basis. While this method allows for more serious and focused tasks, it is unfair towards students who are anxious under time-constraints or in public. Such working conditions and constant supervision may also have an adverse effect on a student's performance. [1]

To evaluate the student's answers, the person in charge of the exam must rely on nonverbal cues to gauge whether the student is looking at the code they are explaining. That allows them to also identify whether the aforementioned student is anxious or just tired.

To accomplish that the examiner would require either special training or experience in the field. Even if these prerequisites are met the assessment is a subjective one and is prone to errors.

The abovementioned process can be improved by utilizing tools such as eye tracking devices. These devices can track a student's gazes across the screen and the dilation of student's pupils. The data acquired from these devices can then be used to aid in assessing the student's knowledge by providing an objective measure of nonverbal cues.

However the eye tracking devices are not the only requirement needed for the assessment, due to the fact that the data acquired by these devices is in a form of raw on-screen coordinates. For the purpose of knowledge assessment these coordinates have to be augmented with additional data i.e. relevant onscreen objects such as blocks or lines of code.

In this paper one such solution, based on the existing iTrace Eclipse plugin, is presented.

The rest of the paper is organized in chapters, the first of which, Research questions, presents a detailed overview of the problems and existing solutions of these problems. The next chapter explores the methodology for the presented solution. The fourth chapter describes the solution. The following chapter provides the results of experiments conducted using the presented solution. The final chapter provides an overview of the methodology and the solution as well as on overview of possible further improvements.

II. RESEARCH QUESTIONS

The main goal of the solution presented in this paper is to provide a framework for the analysis of a student's interaction with program code, in order to improve existing programming knowledge evaluation techniques. This will be accomplished by utilizing an eye tracking device to record a student's gazes while being examined. These gazes are meant to indicate whether the area on the screen the student is looking at corresponds to the answers they provide.

The solution should not be limited to evaluating shorter assignments, contained within a single file, instead, it should provide means to facilitate the evaluation of large scale projects such as enterprise applications.

Thus, the solution should operate within an environment familiar to the students. This means that the environment must support exploring and managing large scale projects, consisting of multiple files and folders, as well as editing, searching and syntax highlighting.

In order to utilize an eye tracking device in the aforementioned environment, a connection between the device and the environment must be established. Although eye tracking devices usually have dedicated software, which enables the recording of raw gaze coordinates or heat maps in video format, it is inadequate for the purposes of this solution. No means for integration with existing development environments, or mapping to on-screen objects are provided. Most eye tracking devices have proprietary drivers, limiting their possibility for adapting to specific purposes.

In [2] Busjahn et al. present a case study which proves that eye tracking is a viable tool in programming education as a non-obtrusive way of analysing a student's visual behaviour.

In [3] and [4] Yusuf et al. and Cole et al., respectively, show that eye tracking can successfully be used to differentiate between experts and novices based on their visual behaviour patterns.

An existing solution to a similar problem is provided by Timothy Schaffer et al. in [5]. The provided solution is an Eclipse plugin that enables communication between the IDE and Tobii X60 and Tobii EyeX eye tracking devices, which can be expanded to work with other devices. It supports mapping gazes to lines of code, as well as different code elements, such as classes, methods etc. This, however is only supported for the Java programming language, since it relies on the Eclipse abstract syntax tree parser for Java. Gaze tracking is restricted to text documents, and a single session can only track gazes inside a single file. The gathered data can be exported to either an XML or a JSON file.

Previous research in this area indicates that the use of eye tracking devices during an examination can improve the quality of knowledge assessment. The inclusion of an eye tracker does not interfere with the examination process itself, rather, it enhances the process in an unobtrusive manner. The process of knowledge assessment with the utilization of an eye tracking device is shown in Figure 1, in the form of an activity diagram.

III. METHODOLOGY

The eye tracking device used in this solution is the Gazepoint eye tracker, as it is the device that was available to the researchers at the time. To provide students with a familiar IDE, the Eclipse IDE was chosen, as it is free, open source, supports a wide variety of programming languages and is extensible by third party plugins. An existing solution for a similar problem, iTrace, is also based on Eclipse, and provided a suitable starting point for the development of the solution presented in this paper.

To establish communication between the eye tracking device and the IDE, on which the iTrace plugin was installed, a separate Java library was developed. The library was used to collect raw data from the eye tracking device and forward it to the IDE. The iTrace plugin itself had to be modified to support the input from the aforementioned Java library. Furthermore, the plugin had to be modified to support tracking within multiple files in a single session. Code and non-code

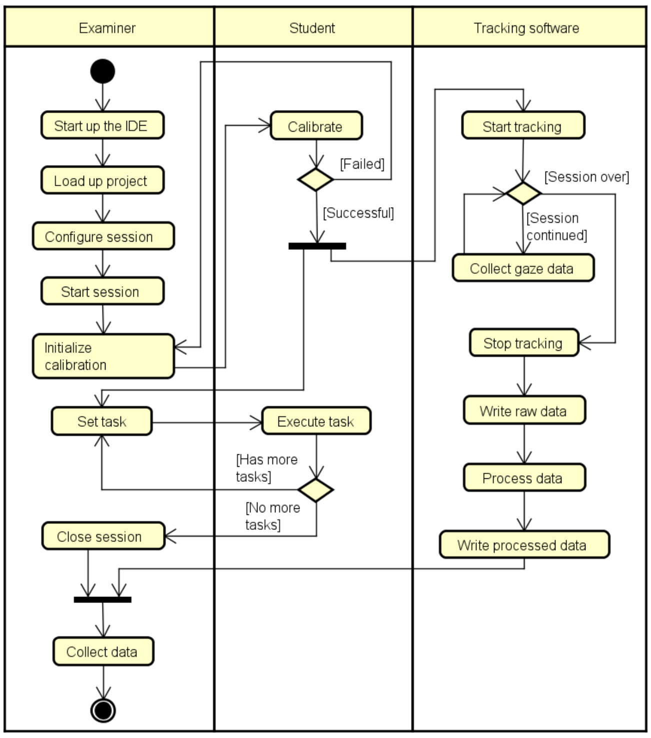

Figure 1. Activity diagram of the examination process.

discrimination was added, as well as tracking of a gaze's duration.

The data acquired by the modified iTrace plugin was filtered to eliminate noise and outliers. Filtered data is then exported for further use, beyond the scope of this paper.

IV. SOLUTION

The Gazepoint eye tracking device functions as a web server, communicating with clients via XML fragments. The communication between the device and the application is established by utilizing a standalone Java library created within this research. The library connects to the device using the standard Java sockets API. After the initial handshake with the device and the exchange of configuration data, the eye tracking device begins to send the actual gaze data in the form of XML fragments. The XML fragments received from the device are parsed using the JCABIXML Java library. This library was chosen because it is a lightweight solution that provides parsing of XML fragments. The output of the library are gaze objects. Each gaze object contains the following data:

- x on-screen coordinate,
- y on-screen coordinate,
- left pupil diameter,
- right pupil diameter,
- validity of the gaze.

A gaze is considered to be valid if it landed on the active area of the screen and if at least a single eye is detected. These objects are then forwarded to the modified iTrace Eclipse plugin.

Due to the limitation of the Gazepoint eye tracker, in order to establish a connection with the eye tracking device, the proprietary Gazepoint Control software is required to be active. Calibration of the device is also done exclusively via the Gazepoint Control software.

The iTrace plugin underwent several modifications to be made suitable for the purpose of this research. The gaze handler, used for mapping coordinates to lines of code, was modified to also track the file containing those lines of code. This allows for tracking gazes across multiple files within the same project in one session. During the session only gazes within the currently active text editor are tracked, as tracking within multiple editors proved too computationally demanding. Support for handling gazes which landed outside of text was also added. Those gazes were flagged as non-code gazes, allowing their later removal if they are deemed irrelevant. Additionally, the duration of every gaze was recorded. Long gazes may indicate that the student was observing a line of code that they perceived as a particularly relevant or complicated piece of code, or that they lacked the understanding of the code they were looking at. Short gazes may indicate that the student perceived the code as irrelevant or, if the gazes were erratic as well, that they were having difficulties finding a particular piece of code, thus implying a lack of familiarity with the project.

The plugin's JSON writer component was also modified to include the duration of the gaze, the relative path to the file, and an additional info attribute, if the gaze was a non-code gaze.

Listing 1. JSON schema of a raw output file

```
{
  "title": "Session",
  "type": "object",
  "properties": {
    "gazes": {
      "type": "array",
      "items": {
        "title": "gaze",
        "type": "object",
        "properties": {
          "file": {"type": "string"},
          "x": {"type": "integer"},
          "y": {"type": "integer"},
          "left_validation": {"type": "double"},
          "right_validation": {"type": "double"},
          "tracker_time": {"type": "integer"},
          "gaze_duration": {"type": "integer"},
          "system_time": {"type": "integer"},
          "nano_time": {"type": "integer"},
          "line_base_x": {"type": "string"},
          "line": {"type": "string"},
          "col": {"type": "string"},
          "relative_path": {"type": "string"},
          "absolute_path": {"type": "string"},
          "line_base_y": {"type": "string"},
          "info": {"type": "string"}
          "left-pupil-diameter": {
            "type": "double"
          },
          "right-pupil-diameter": {
            "type": "double"
          },
  }}}}
}
```

The structure of the output file is represented by a JSON schema, as is shown in Listing 1.

The data was then processed by a separate Python script, using the SciPy Python library. During processing, all gazes flagged as non-code were removed. Afterwards, all irrelevant gaze attributes were removed, leaving only the filename, line and column of code, as well as the duration of the gaze. All gazes with durations longer than one frame were oversampled. This was done to equally distribute the gazes over the duration of an entire session. This data, shown on Figure 2, representing a whole experiment session was split into several sequences of gazes, each corresponding to a single file, taking into consideration that a file may be observed several times in different timespans. The raw data obtained from an eye tracking device proved to be noisy. The noise manifests as an inordinate variance of the observed line of code in a short time span. The noise originates partially from the hardware, and partially from chaotic eye movement of the student participating in the experiment. Reduction of this noise was accomplished by applying the median filter to the data. The resulting data may then be directly utilized for the evaluation of the student's interaction with the code. This data is exported to a JSON file. The structure of this file, in the form of a JSON schema is shown in Listing 2.

The interface of the iTrace plugin was also modified, removing redundant elements and adding fields for defining the name of the output file. Also an additional button was added for the purpose of activating the calibration of the Gazepoint eye tracking device from within the IDE.

Listing 2. JSON schema of a processed output file

```
{
   "title":"GazesArray",
   "type":"array",
   "items":{
      "title":"Gaze",
      "type":" object",
      "properties":{
         "line":{
            "type":"integer"
         },
         "file":{
            "type":"string"
         },
         "column":{
            "type":"integer"
         }
      },
      "required":[
         "line",
         "file",
         "column"
      ]
   }
}
```

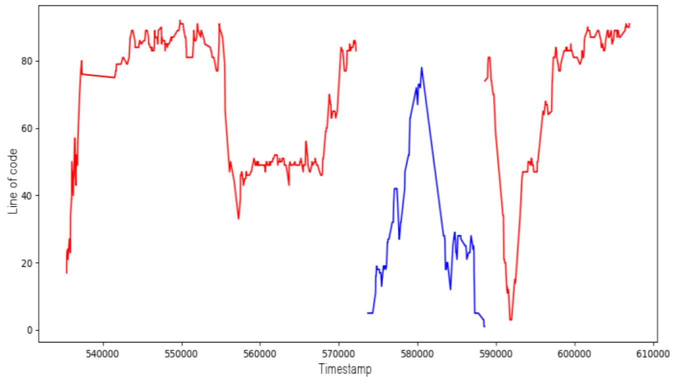

Figure 2. A session with multiple files. Each file is represented with a different color.

V. Results

The validation of the system was performed by conducting experiments with volunteers. The volunteers were asked to find a specific section of code inside a project containing multiple files, while reading the code they are currently looking at out loud, as well as vocalizing each change of file. The audio of the session was recorded. The data retrieved from the experiments was compared to the audio recordings, and proved to be consistent, especially after the reduction of noise. Figure 3 shows the data before noise reduction. As is evident on Figure 3, the observed line varies inordinately in both directions in a short time span, which is inconsistent with the expected manner in which a person would read.

Figure 4 shows the data after noise reduction. The variance of the observed line is lower in a short time span, making it easier to interpret the observed line in any given moment, as well as the direction in which the person is reading.

VI. Conclusion

The solution presented in this paper is a software for collecting data of a student’s interaction with program code. Its main purpose is to aid in the evaluation process of a student’s programming knowledge. The solution’s application is not limited to small scale projects contained within a single file.

The solution is based on iTrace, an existing Eclipse plugin. It is modified to receive data from the Gazepoint eye tracking device by using a separately developed Java library. The raw data is converted to gaze objects. These gaze objects are then mapped to lines of code, filename and gaze duration. The gazes are finally processed to remove noise and enable further analysis and assessment of a student’s programming knowledge.

Tests conducted with volunteers returned promising results. The output data was interpretable i.e. the observed line in any time frame was discernible, as well as the direction of the reading.

Plans for further development include:

- Tracking of gazes inside the Eclipse output console, error report, and project explorer.
- Integration of pupil diameter data into the output data.
- Tracking of gazes inside multiple editors simultaneously.

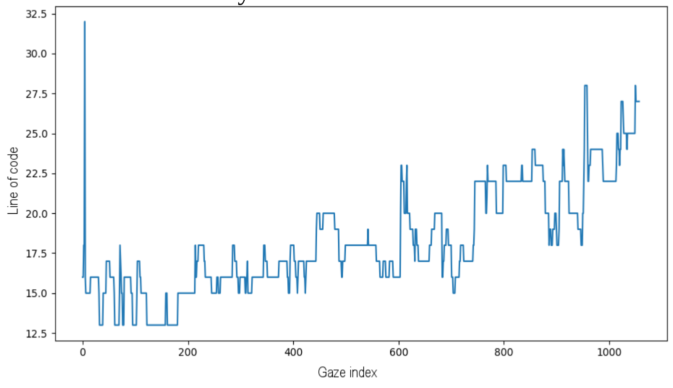

Figure 3. Unfiltered gazes

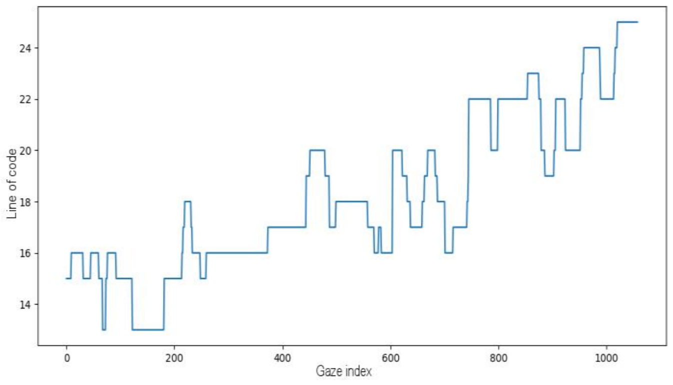

Figure 4. Filtered gazes

References

[1] M. W. Mccracken, V. L. Almstrum, M. Guzdial and B.-D. Y. Kolikant, "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students," in *ITiCSE*, Canterbury, 2001.

[2] T. Busjahn, C. Schulte, B. Sharif, A. Begel, M. Hansen, Simon, R. Bednarik, P. Orlov and P. Ihantola, "Eye tracking in computing education," in *Proceedings of the tenth annual conference on International computing education research*, Glasgow, United Kingdom, 2014.

[3] S. Yusuf, H. Kagdi and J. I. Maletic, "Assessing the comprehension of UML class diagrams via eye tracking," in *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference*, Banff, Canada, 2007.

[4] M. J. Cole, J. Gwizdka, C. Liu, N. J. Belkin and X. Zhang, "Inferring User Knowledge Level from Eye Movement Patterns," *Information Processing and Management*, pp. 1075-1091, 2013.

[5] T. R. Schaffer, J. L. Wise, M. B. Walters, S. C. Müller, M. Falcone and B. Sharif, "itrace: Enabling eye tracking on software artifacts within the ide to support software engineering tasks," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Bergamo, Italy, 2015.