Using an eye tracker to measure developer focus while writing unit test from within the IDE

Martijn Gribnau,
Delft University of Technology
m.m.w.gribnau@student.tudelft.nl

***Abstract*—Background: Reading and understanding source code and writing test cases are indispensable parts of routine for modern software developers. Unit testing helps many developers to detect and prevent bugs at an early moment and is a cost-effective way to improve the effectiveness of a program. With the availability of portable eye trackers, it has become possible to measure what developer focus on while comprehending code. Goal: This paper identifies the effort spent fixating between production code, test code and feedback on test results from the IDE.**

Method: Participant developers are given production code and are tasked with testing the code like they would in their routine development situation. To enable a routine-like development situation, developers have access to a popular Java IDE. The gaze of the developer is tracked using an eye tracker. Fixations within marked areas of interest (AOI) are calculated from the gaze measurements using the i-Vt algorithm. Fixations are then mapped to areas of interest which have been created from the screen recording. From the fixation-to-AOI mapped results we can learn what developers look at when developing unit tests. Results: Developers spend the majority of their fixation time, focusing on test cases (tc), followed by the production code (ic), which encompasses the method under testing, the auxiliary code and the code comments. Little time is spent fixating on the test results, but this could be explained by the test results being only the indication of feedback to a developer. When delving deeper in understanding on why written test cases might fail, a developer will look into the ic and tc areas again, as they provide the source to find that understanding. Conclusions: We investigated where developers spent their time while unit testing. From the results we can conclude that on average developers spent slightly more time fixating on the test code than on the production code. This could imply that writing test cases requires active understanding of the test case being written. The method used did not allow for reliably identification of AOI sequences.

I. INTRODUCTION

Software testing happens on many different levels and many different processes have been developed, integrated in well known software development models such as the Waterfall model, the V-model and Rapid Application Development models [1], [2]. An important part of many of these models is component level testing [2], also known as unit testing, which usually happens right before or after the development of incrementally added program features. Unit testing has the advantage of being accessible, since it is done isolated from the rest of a program and is a cost effective method to find and prevent faults (also known as bugs) in programs in an early state of the software development cycle [2].

Unit testing is perceived to be commonly used. Beller at al. [3] reported in a study which among others tried to find out how much software testing was used in software projects, that from the 460 projects they analysed, 43% were detected to contain tests and 19% contained JUnit unit tests which the Eclipse integrated development environment (IDE) used in the study could actually run.

As biometric tools such as eye tracking have become more widely available, researchers have started using it in the program comprehension field. Various researchers have been using eye tracking to measure program comprehension within production code [4], [5]. After searching and not finding studies which focus on test code comprehension, Yu investigated (without the use of biometric tools) what test code comprehension was influenced by. This paper delves further into looking at test code comprehension by identifying the time developers spent fixating on different areas of interest while developing test code.

II. RESEARCH METHODS

The goal of this study is to explore what developers focus on during the writing of unit tests, knowledge which can be used to enhance unit testing methods. It also seeks to find which code segments are important during the writing of test cases and what focus patterns between code segments are common. The following research questions are presented to that aim:

RQ1 Where do developers spend their time while unit testing?
RQ2 Can the finding of bugs be predicted by the amount of tests written and the time spent on fixating on the test code area of interest?

A natural way to find out on what developers focus on is by finding out how much time they spend on different code segments. Sharif et al. [5] showed that spending more time reading source code could have beneficial effects on how developers comprehend source code. For developers, spending more time on a code segment indicates that the area is important to grasp enough understanding, to write proper unit tests. The code segments, also called areas of interest, used for measurements during this study are:

- ic: implementation or production code, subdivided into:
  - mut: method under testing
  - aux: auxiliary code relevant to the method under testing
  - cm: code comments
- tc: test code for method under testing
- tr: test results for method under testing

The implementation code, also known as production code ic, is the human readable source code written by a developer which implements features for a program to function. It is further divided into the method under testing mut, auxiliary code which the method under testing could rely on or is otherwise relevant, called aux, and code comments cm which can guide a human in understanding the implementation code. The segment which contains the unit test code is marked as tc. Finally, there are the areas called tr and dbg, which stands for test results and debugging tools respectively. While tr and dbg are not segments within the code, these areas provides indispensable feedback to a developer on the success or failure of a test case, the code coverage of the implementation code and specifically the method under testing and in the case of dbg feedback towards understanding of the behaviour of the code.

1) Overview of the Experiment: To aid in finding an answer on the above research questions, the following experiment setup is presented: a participant receives a code task which deliberately contains a fault. The fault is obscured by the surrounding source code, making its unlikely to be spotted right away. Additionally the fault is a logical one, to ensure that the static analysis tools automatically running in the background of the IDE will not flag the bug and warn the participant. The participant will not explicitly be told that the code contains a fault and will be tasked with completing the code such that the code will be ready by means of testing and fixing problems in the code which came up as a result of performing software testing activities, like how the participating developers would if it was their own source code.

The focus (i.e. where on the screen the participant is looking) of the participant will be tracked in real time, and the actions taken by a participant are written down so when a participant finds a bug, it will be noted. The implementation and test code are checked afterwards to verify whether the participant found and fixed a bug.

The participant is additionally asked about their level of programming experience in years and months for both programming in general, and programming in Java, the language used in the experiment.

A. Developer Environment

The experiment used Java as programming language, which is a modern, widely used programming language and has actively maintained and mature tools and testing frameworks. As testing framework, JUnit 5 was used. With a focus on providing a development environment which a majority of participants would be used to, it was decided to use the popular Java IDE Intellij IDEA 2019.2.1 (Community Edition)[6], [7]. The measurements were taken on a Dell XPS laptop (Intel i7-8750H 2.2Ghz with 32GB RAM installed), running the Windows 10 Pro operating system. The experiment took place in a evenly lighted room without outside facing windows, to minimise possible reflections which could influence measurements taken by the eye tracker. Figure 1 shows the IDE as presented to a participant at the start of the code task.

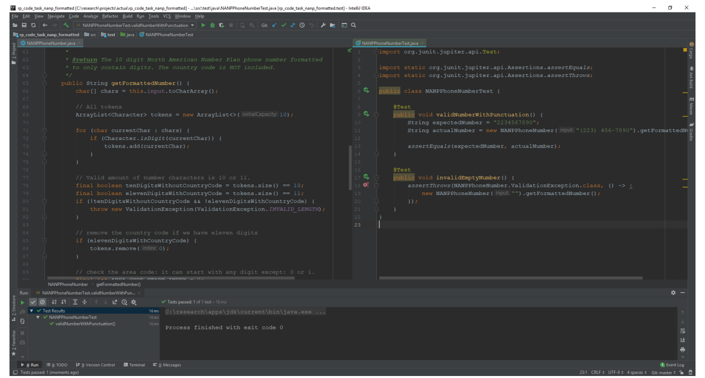

Fig. 1: Developer environment

B. Task

A key wish in the setup of the experiment was to provide a developer with a natural developer environment (i.e. an environment where a developer can feel at ease, for as much as a lab experiment allows). The task used in this experiment is no exception. The task is derived from the Phone Number exercise[8] from the code practice website *exercism.io*[1]. The selected task was chosen because it describes a real world practical problem, it does not require deep domain specific knowledge, can be rigorously tested within half an hour and requires one to break the testing of the provided implementation code in multiple smaller parts, as it contains multiple code evaluation paths.

The provided code tasks contains a bug (signalling improper implementation), which should not be instantly visible on the a first scan, but will become visible by systematically testing the implementation. In order to provide a slight direction to a participant, examples of phone numbers which should be accepted by the $getFormattedNumber$ method (the method which a participant is asked to unit test) and examples which should be rejected are provided as part of the inline documentation. The implementation code for the code task can be found in appendix A and the template for the test code can be found in appendix B.

The fault in the code sample consists of the requirement to handle a country code as input, while only accepting North American Number Plan (NANP) numbers, thus only the country code $'1'$ should be accepted. Note that country codes which are longer than one digit will be rejected based on the required amount of digits (11 if the including the country code or 10 if the country code is not included).

Other faults which a participant could evaluate and decide to be bugs are the automatically acceptance of any character which is not a digit (e.g. alphabetical letters) and the use of an $ArrayList$ with the $remove(0)$ method which shifts all other characters to the front upon removal of the first character.

Before a developer participant starts with the experiment, an explanation of the build up of the code task is given, so the participant will have a general understanding of the case.

[1]The tasks are provided by exercism.io under the MIT License

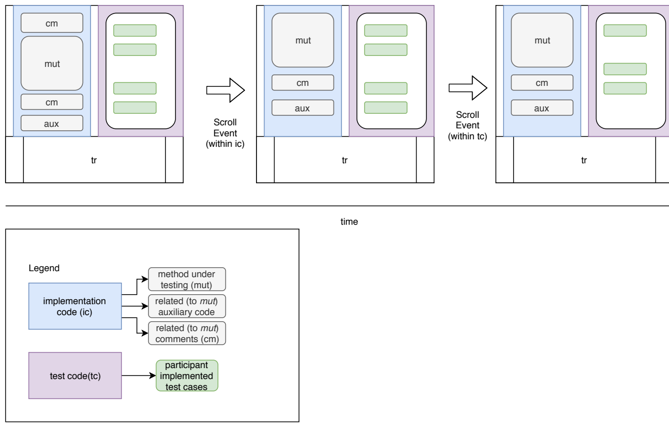

Fig. 2: When content on the screen moves, AOIs should move with the content

Normally, a developer would have gained such understanding by researching the feature to implement or by implementing it in source code. Specifically, the specification of the code task is explained including, but not limited to, the requirement of only allowing 1 as the country code within the input. The examples which are also part of the inline documentation at the class level are given as examples during the instruction of the participants. A participant is told to test the code in a similar way to how they would normally test their own code.

C. Measurements

In order to precisely measure what a participant focuses on, a Tobii X2-30 eye tracker [9] was used. The eye tracker, in combination with a screen recording, enabled us to measure on which areas of interest (AOI) a participant fixates while developing test code. For this to work, data recorded by the eye tracker had to be mapped on the screen recording, as the developer environment is not static and allows interactions such as scrolling windows and switching between tab views of source code files.

The iMotions[10] software was used to record, synchronise and initially process the raw data received from the Tobii X2-30 eye tracker. The raw data contains the gaze positions and eye positions and the iMotions software calculated eye fixations based on the i-Vt[11] algorithm. Fixations are small, domain limited planes of eye gaze positions which are within a short distance of each other, within a short period of time [12]. The metrics used to measure how much a developer focuses on an area of interest are the fixation time and the fixation count. The fixation time represents the duration of the time spent fixating within an area of interest. The fixation count is the number of fixations which happened within an area of interest. Fixation was chosen as a metric because fixations are considered to imply the moments of comprehension[13].

D. Post processing

The recorded raw data and calculated fixations required post processing to map measured eye gaze positions on the screen to the recorded screen content within the IDE and find out whether a developer's gaze is within an area of interest at a given moment. Areas of interest were manually created by dragging rectangles on the code segments and test results screen. Before the areas of interest could be created with iMotions, the video recording of the content on the screen had to be converted to individual image frames as the video based area of interest tool was not designed for large amounts of screen details and low contrast. Since the screen recording contained moving content (for example by means of scrolling or typing), marked areas of interest would have to move with their content on each recording frame. The excessive amount of recorded frames made this impossible. Observing the recording showed that for the majority of the time, content on the screen did not move. Because of this observation, an approximation of the measurements taken (weighted up against the large amount of recorded data) was used in which frames were no movement or no likely movement took place were merged. After completing this process, the results were scanned for rough errors in and detected errors were corrected manually. Figure 2 shows the necessity to remap areas of interest between moments of motion.

This process started with the export of scroll event data as recorded by iMotions, which consisted of scroll events which were the primary source of vertical motion within the IDE. The other major source of vertical motion was typing in new test cases, however as test cases were only tracked by the tc code segment, the moving of text within that window did not matter for the mapping of areas of interest to the recorded screen content. The scroll event data consisted of time stamps in milliseconds from the start of a scroll event.

Here, the time stamps were converted to seconds. Then for each time stamp t_i, we round it up (c_i) and down (f_i) to an integer. This is done to ensure we do not capture a frame in the middle of a scroll event. Then, for any pair of subsequent events (c_{i-1}, c_i) where $c_i - c_{i-1} \leq 1$, we merge these events together (f_i and c_{i-1} are marked as unused; the merged event spans from f_{i-1} to c_i).

The next step is to actually create the separate scenes in the video recording using the sequence intervals as found in the first step. These scenes are created using the scene tool in the iMotions software. To reduce the chance of human mistakes while entering each sequence, the scenes were created by generating the input using the AWT Robot functionality [2]. The first and last scene from the beginning and the end of the screen recording respectively were created manually to ensure only scenes were created which captured data from a participant which was working on the task.

Now we have static images on which we can mark areas of interest. This process was done completely manually for all scenes. The areas of interest were labelled according to the code segments as defined at the beginning of this section. Note that the mut, aux and cm code segments overlap the ic code segment. Any tracked time spent within mut, aux and cm will thus also be registered for ic (measurements registered for

[2] AWT Robot: https://docs.oracle.com/javase/8/docs/api/java/awt/Robot.html

mut, aux and cm should thus be subsets of the measurements registered for ic).

E. Analysis

After post-processing the data, the data is further prepared and analysed with Rust using the csv and serde packages (crates), and Python 3.7 with the Pandas ans Seaborn packages.[3].

To answer RQ2, the post-processed data was used in combination with the collected data on whether a bug was found or not. Logistic regression was applied with the binary output variable whether a participant found a bug, and with for the input variables either the number of test cases written by a participant or the amount of time spent fixating on the test code area. The logistic regression was performed in Excel using the Solver add-in.

F. Participants

For this experiment 10 healthy participants were recruited, from which 9 male and 1 female (aged between 17-25 years old). From this selection, experiments run by 3 participants failed (for 2 experiments, recording stopped in the middle of the experiment for unknown reasons and 1 run of the experiment was not possible due to participant requiring glasses to correct eyesight which caused imprecision on the eye tracking recording. The 7 remaining participants were all male, aged 17-25 years old. There were 2 participants with between 6 months and 1 year of programming experience, 2 participants with roughly 3 years of programming experience and 3 participants with approximately 6 years of programming experience. All participants were Computer Science students studying at the Bachelor or Master level.

III. Results

A. RQ: Where do developers spend their time while unit testing?

Figure 3 and table I show the time as measured for fixations on areas of interest per individual participant. Figure 4 and table II show the measured fixation count on areas of interest per participant. The figures and tables show that participants spent the majority of their time on focusing on the test code. A significant amount of time is also spent on the implementation code. Depending on the participant,

Note that the implementation code area of interest (labelled ic), contains the method under testing (mut), auxiliary code (aux) and comments related to the method under testing (mut) areas of interest. Any fixation time and fixation count towards these sub areas is also counted towards the implementation code area. These sub areas however do not necessarily span the total implementation code area. Additionally there are unmapped areas, such as menus within the IDE and moments where participants did either not look at the screen, or no valid eye tracking data was collected. As a consequence, the total processed time (the time for which eye tracking data was

[3]Source code for data processing: http://doi.org/10.5281/zenodo.3261826

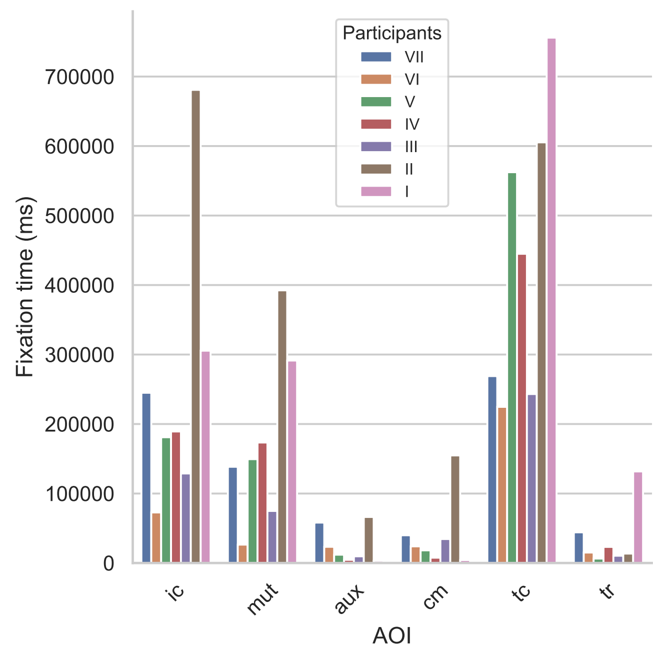

Fig. 3: Time of fixations per area of interest per participant

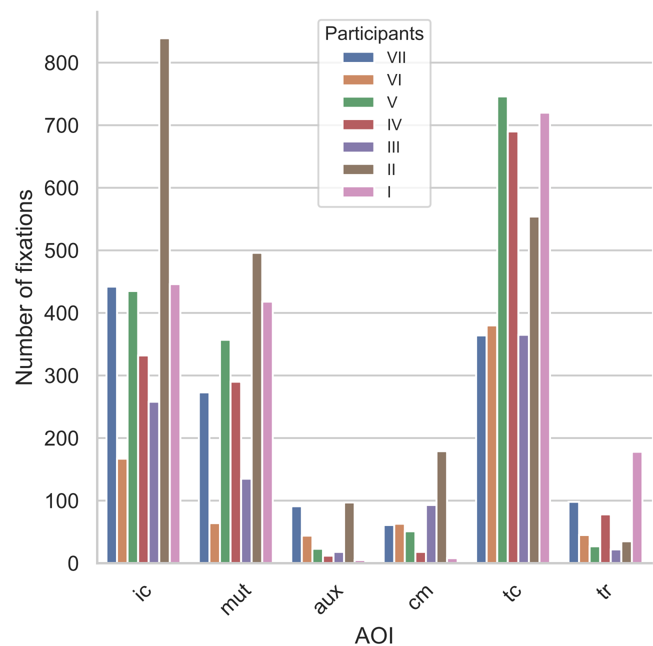

Fig. 4: Number of fixations per area of interest per participant

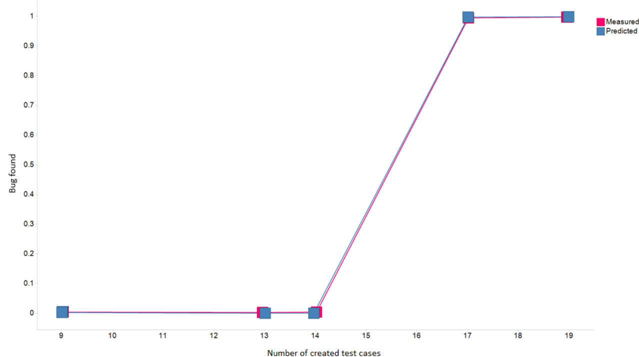

Fig. 5: Logistic Regression of finding a bug as function of number of tests created

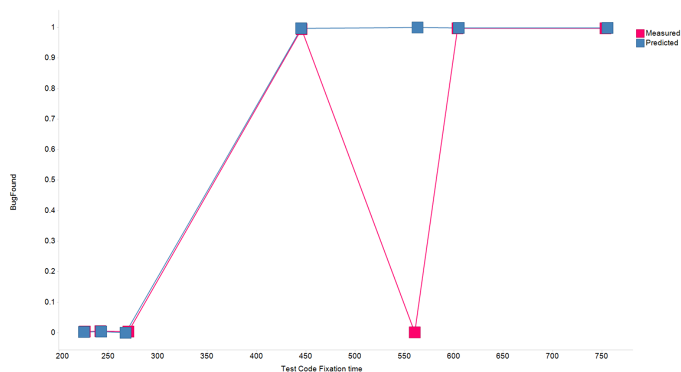

Fig. 6: Logistic Regression of finding a bug as function of time spent fixating the test code

mapped to areas of interest) will be larger than the sum of the individual areas of interest.

The time spent fixating on the ic was 30 to 50 percent compared to the time spent fixating on the tc for participants I, III, IV, V and VI. Participant II and VII spent approximately equal amount of time on both the ic and tc areas of interest.

On the test results tr, significantly less time was spent. Participant I spent relatively more time on this area of interest than any of the other participants.

Within the implementation code, most time is spent on the method under testing. The time spent on axillary code and code comments related to the method under testing is rather small in general, except for participant II who spent a significant amount of time fixating on cm.

The trends in fixation count are similar to the time spent fixating for all participants.

B. RQ2: Can the finding of bugs be predicted by the amount of tests written and the time spent on fixating on the test code area of interest?

Figure 5 shows the result of a logistic regression, where the finding of a bug is used as output variable ($0 = no$ and $1 = yes$). The number of test cases developed by the participant is used as input variable. When taking a closer look at the figure, we can see that there is a good fit to the data, as the predicted values are on the $y = 0$ or $y = 1$ line. The offset value on approximately $x = 15.4$ indicated that there is a relation between the number of test cases developed by a participant and the identification of a bug by a participant.

Figure 6 shows a similar relation. Here finding a bug by a participant is also used as output variable ($0 = no$ and $1 = yes$), but the time spent on fixating on the test code area by a participant is used as the input variable instead. This results of this analysis show that developers which spent more than 357 seconds fixating on the test code are predicted to finding a bug, whereas developers which spent less time fixating on the test code are predicted to not finding a bug.

IV. Discussion

In the research method, we asked ourselves the following questions:

RQ1 Where do developers spend their time while unit testing?

RQ2 Can the finding of bugs be predicted by the amount of tests written and the time spent on fixating on the test code area of interest?

A. RQ1

As noted in the results section, we found that participants spent the majority of their time fixating on test code. On average developers spent slightly more time fixating on the test code than on the production code. This could imply that writing test code requires significant comprehensibility effort. We should however not forget that the production code was given, and the test code was written by the participants. The fixations on the test code are thus made up from both the reading of test code which already exists, and the writing of new test code. The effort to write tests (and applying the understanding on how effective the test will be on testing the implementation) could be larger than one would expect. The test results area of interests was not fixated on much. Probably this is the case, since only limited amount of fixations are enough to see failure of test cases. If a developer then want to understand why it happened they will fixate on the ic and tc again, Further research is needed to track down the effectiveness of the fixation on the selected areas of interest.

B. RQ2

Research question 2 asked whether implementing more test cases, or whether spending more time fixating on the test code area of interest would result in a better probability to discover a bug in the implementation code. Figure 5 and 6 show that this is the case for both parts of the question. It indicates that more time spent on testing will result in potentially less errors in production code. This can potentially be generalised to a more general statement that spending time on developing test code may result in the early detection and prevention of bugs in production code. It should be noted however, that the number of participants in this study is quite small, thus the results found may still be too optimistic.

AOI	I	II	III	IV	V	VI	VII
	Participant						
ic	305.6 (20%)	680.9 (39%)	128.8 (23%)	189.3 (16%)	181.0 (19%)	72.8 (14%)	245.0 (23%)
mut	291.4 (19%)	392.4 (22%)	74.9 (13%)	173.4 (15%)	149.4 (15%)	26.5 (5%)	138.5 (13%)
aux	3.1 (<1%)	66.3 (4%)	9.7 (2%)	4.6 (<1%)	12.0 (1%)	23.3 (4%)	58.2 (5%)
cm	4.2 (<1%)	154.8 (9%)	34.5 (6%)	7.7 (1%)	18.2 (2%)	24.0 (5%)	39.9 (4%)
tc	756.0 (49%)	605.3 (34%)	243.1 (43%)	445.0 (39%)	562.4 (58%)	224.7 (43%)	269.0 (25%)
tr	131.777 (9%)	13.556 (1%)	10.489 (2%)	23.185 (2%)	6.385 (1%)	15.071 (3%)	44.166 (4%)
$\sum$ processed time (s)	1546	1768	569	1155	977	524	1081

TABLE I: Total fixation time per AOI per person (in seconds).

AOI	I	II	III	IV	V	VI	VII
	Participant						
ic	446	839	258	332	435	167	442
mut	418	496	135	290	357	64	273
aux	5	97	18	12	23	44	91
cm	8	179	93	18	51	63	61
tc	720	554	365	690	746	380	364
tr	178	35	22	78	27	45	98

TABLE II: Total number of fixations per AOI per person

C. Measuring eye tracking within an IDE

Integrated development environments are complex beasts, where many interactions and events can happen which could distort measurements taken when combining eye tracking data with a gaze mapping. Most actions and tools within an IDE are overkill to perform studies like these. While a participant can have the comfort of having an IDE it knows and uses on a daily bases, for a researcher it introduces many potential problems which can interfere with the sampled data. During this experiment, any moving window within the IDE introduced a difficult to detect potential imprecision of data. Intellij has many hot keys which can open windows, and since the method used to perform gaze mapping between moments of motion on screen rendered objects relied on detecting scroll events, any changes in rendering would could not have been accounted for by the automatically created scenes. To account for that after creating the scenes automatically based on scroll events, the screen editor was used to manually scan the created scenes on errors and fix them. Participants were also asked before the start of the task to not open more windows than the ones which were already present: the default Intellij view, but with the implementation code on a vertically split window on the left, the test code on a vertically split window on the right and with test results below these windows. This however could raise the question why to use an IDE. Aside from familiarity to a participant, another reason is that all tools to perform unit testing activities are available from within a single window. While much can change within that window, using for example a terminal window to run the tests could introduce an at least even difficult mapping, as its position on the screen is neither static.

1) Static image gaze mapping: This study makes use of individual static image scenes to perform the gaze mapping process. This allows one to relatively easily create areas of interests on a scene. If no flattened image frames were used, the areas of interests should have been mapped to the video screen recording content. As a result, an area of interest would have to move with the content for any area of interest where the content on a previous screen would change the position of the area of interest in a next frame. Video recordings can of course be seen as sequences of individual images. Thus, creating a gaze mapping on the video of the screen recording can still be performed by, for each frame re-creating the mapping of the areas of interests. However, such a task would be unmanageable if done manually. Imagine having a screen recording of 30 frames per second, which goes on for 10 minutes. Then for a single participant, areas of interest would have to be mapped (or re-mapped) $30*60*10 = 18000$ times manually, once per frame. This amount can be decreased by reducing the amount of frames per second or by taking the difference of two consecutive frames and if there is none, using the same area of interest mapping (assuming the content on the screen allows for the same areas of interest). Still this remains a job of a large amount of impractical and error prone manual labour. Another option was to reduce the amount of frames by diving the frames for a participant by an integer x, and then only mapping the areas of interest for the frames the resulting amount of frames on equal time intervals. This has as advantage that the scenes have equal duration and as much resulting (static image) frames as viable could have been chosen. However, it does not take motion into account whatsoever. Since motion in a scene was the observed to be the most important divisor between scenes, this process was chosen.

A more practical solution would be to have an IDE (or whichever tool is used to present code on the screen) report where on the screen the areas of interest are rendered. This could introduce a problem with regard to synchronizing the positional data points of the areas of interests visible the screen with any other stimulus (for example because of induced latency) but at the clear advantage that you now would have a quite precise areas of interests for any moment in the recording. Little to no error prone manual work would be further required to perform a gaze mapping.

However in the event that such a tool is not available, we'll have to resort to other solutions, which perhaps also in effect reduce the probability on the correctness of the measurements.

The method described in the research method and used

during this study can also be further improved. Even if we can not report precise positions for our areas of interests within the IDE, we can still try to automatically create scenes for any event of motion by subscribing to any event fired by which involves potential motion.

2) Program complexity: As previously mentioned, we wanted to provide a participant with a natural developer environment. Tasks which a participant carried out are were no exception to this requirement. Regularly, studies which perform code comprehension related research use code samples from introduction algorithm and data structure courses or even simpler code samples such as a single if statement, a for loop and a few variables. While both can be interesting test cases, they (1) do not represent industrial written code (usually introductory algorithm and data structures are part of a standard library), (2) follow a certain structure which a computer science student is too much trained on, hence introducing a bias, and such samples will often be recognized by computer science students in advance and (3) might require specific prior knowledge on these specific topics (e.g. how heap sort or linked lists work). On the other part of the spectrum we could have chosen to use industrial code from, for example, open source industrial projects. We observed that these projects required too much domain specific knowledge to get up and running. This would substantially limited recruitment of participants and the time spend on gathering and processing data per participant would significantly increase. As mention in the research method, we chose to use a code practice task with a short introduction on the problem instead. This allowed us to present a single class of implementation code and a single class of test code to a participant, while also introducing the required complexities and code paths to have sufficiently complex unit test cases.

V. Related Work

Increasingly biometric measuring devices such as eye trackers are being used for measurements related to program comprehension. For example, Busjahn et al. [14] looked into the linearity of source code reading and compared it to the linearity of natural language text. In their study, they found that novice participants follow a linear reading order (called "Story Order") more than expert participant developers did when reading source code. The authors think based on their results that experts more closely follow the "Execution Order" of a program (the order in which program statements and expressions are evaluated). Another study by Jbara and Feitelson [4] looked at the influence of regularity, the repetition of source code patterns, within a single function of source code. They make a case for the inclusion of the regularity of code to also be viewed as a metric of code complexity (which influences program comprehension), especially since simpler metrics of code complexity such as the amount of lines or cyclomatic complexity [15] do not consider type and context of the code and therefore do not actually represent how much effect the code has on actual the understanding of a developer, which relies on the context of other parts of the code. The study shows recurring patterns within the reading of a function, such as the initially scanning followed by reading or fixating or participants regularly looking ahead after a comprehension event took place.

Walters et al. [16] presented a plug-in for the Eclipse integrated development environment (IDE), which generates links between code segments and the eye gaze data and can also show these links from within the IDE. This has major advantages on scalability of eye tracking from within an IDE, has the i-trace tool will take take of capturing and calculating areas of interest from selections of code.

To get an impression where the idea of using fixations which often imply comprehension come from, one can read a paper by Just and Carpenter [13] which presented a model which aimed to explain comprehension during reading. The model was tested on scientific text samples and was conducted on college students. While source code is different from regular text because it does not necessarily follow the same execution order as more linear text samples [14], there are also similarities like how context within and between sentences are related to one another, which is similar in a way to how parts of code can also depend on other parts of code. The presented model joins eye fixation to reader comprehension based on two assumptions which the authors call the "immediacy" assumption and the "eye-mind" assumption. The "immediacy" assumption describes that words are being processed on the spot and the "eye-mind" assumption can be explained as the eyes not leaving a word for as long as is necessary for a person to process (which implies understanding) the word. The results show some confirmation which support both the "immediacy" and the "eye-mind" assumptions.

Exploratory work towards understanding of test code was done by Yu in his thesis titled *Towards Understanding How Developers Comprehend Tests* [17]. In this thesis, Yu decides on three factors which reflect the comprehension of test code: *reading time*, *the ability to identify the testing purpose* and the *ability to produce extra test cases*. To measure what part of the code a participant is reading, a sliding viewport is used. The side effect of this viewport is that participants can not jump backwards and forwards over the whole code, taking away the ability to scan ahead or re-scan the previously fixated areas. Participants are thus required to keep previously visited code clear in their mind during each test case. Yu observed that the reading time was not significantly dependent upon the measured experience of the participants with Java and with prior experience of using tests.

VI. Threats to Validity

Manual AOI mapping First, the manually mapping of areas of interest. Any human makes mistakes, and humans do not necessarily create exact equal areas of interest when using a drawing tool to select an area of interest. Areas of interest have been mapped to the contexts to the best of the authors ability. Additionally, the areas of interest are usually quite large (think a full code block like a method) to improve

precision, and thus allow for some imprecision just around the area.

AOI mapping with static scenes Secondly, the mapping of AOIs to static images should happen on all events where the mapped AOIs for a previous screen versus the the AOI mapping on the current screen could change based on changing content as result of actions or motions. The creation of scene around potential events which produce motions is not idea. The created scenes were however manually scanned after automatic creation based on scroll events, to correct the biggest mistakes.

Amount of participants The involvement of around eye tracking and requiring human participants to be available on site is sub optimal to enlarge your set of participants. The amount of data gathered is however offset by the duration of 10-30 minutes each participant spent on performing the code testing task.

VII. Conclusion

This study aimed to evaluate where developers fixated, to understand where developers spent the most time comprehending code while performing unit tests. Test code was found to be slightly more focused on compared to implementation code. While many program comprehension researchers have successfully used eye tracking devices to measure focus in program comprehension, it is mainly performed in a small, static environment where horizontal and vertical motion such as scrolling do not influence the gaze mapping of screen coordinates to areas of interest mapped over the screen content. Automated tooling is required to effectively and precisely map eye gaze to areas of interest on screen contents which are regularly in motion.

References

[1] N. Munassar, A. G. IJCSI, and undefined 2010, "A Comparison Between Five Models Of Software Engineering," *Citeseer*. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.402.8120&rep=rep1&type=pdf#page=115

[2] R. Black, E. Van Veenendaal, and D. Gramham, *Foundations of software testing : ISTQB certification*. Andover, Hampshire: Cengage Learning EMEA, 2012.

[3] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their IDEs," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. New York, New York, USA: ACM Press, 2015, pp. 179–190. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2786805.2786843

[4] A. Jbara and D. G. Feitelson, "How programmers read regular code: a controlled experiment using eye tracking," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1440–1477, jun 2017. [Online]. Available: http://link.springer.com/10.1007/s10664-016-9477-x

[5] B. Sharif, M. Falcone, J. M. P. o. t. S. on Eye, and undefined 2012, "An eye-tracking study on the role of scan time in finding source code defects," *dl.acm.org*. [Online]. Available: https://dl.acm.org/citation.cfm?id=2168642

[6] StackOverflow, "StackOverflow Developer Survey," https://insights.stackoverflow.com/survey/2019, 2019, [Online; accessed 02 April 2019].

[7] Jetbrains, "Intellij IDEA, a Java oriented Integrated Developer Environment (IDE)," https://www.jetbrains.com/idea/, [Online; accessed on May 10th 2019].

[8] Exercism authors, "Java exercise: Phone number," https://github.com/exercism/java/tree/master/exercises/phone-number, 2019, [Online; accessed on May 27th 2019].

[9] "Tobii X2-30 Eye Tracker," https://www.tobiipro.com/product-listing/tobii-pro-x2-30/, [Online; accessed on May 10th 2019].

[10] iMotions A/S, "imotions biometric research platform 7.1," https://imotions.com/eye-tracking/, 2018, [Online; accessed on May 27th 2019].

[11] A. Olsen, "The tobii i-vt fixation filter," *Tobii Technology*, 2012.

[12] K. Holmqvist, M. Nyström, R. Andersson, R. Dewhurst, H. Jarodzka, and J. Van de Weijer, *Eye tracking: A comprehensive guide to methods and measures*. OUP Oxford, 2011.

[13] M. Just, P. C. P. Review, and undefined 1980, "A theory of reading: From eye fixations to comprehension." *psycnet.apa.org*. [Online]. Available: https://psycnet.apa.org/record/1980-27123-001

[14] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, "Eye movements in code reading: relaxing the linear order," pp. 255–265, 2015. [Online]. Available: https://dl.acm.org/citation.cfm?id=2820282.2820320

[15] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, dec 1976. [Online]. Available: http://ieeexplore.ieee.org/document/1702388/

[16] B. Walters, M. Falcone, A. S. . t. I. . . . , and undefined 2013, "Towards an eye-tracking enabled IDE for software traceability tasks," *ieeexplore.ieee.org*. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6620154/

[17] C. S. Yu, "Towards Understanding How Developers Comprehend Tests," 2018. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid%3A3d251634-5499-4423-ab66-d9803a6ae877?collection=education

Appendix A
Appendix: implementation source code for NANPPhoneNumber

Listing 1 and listing 2 show the implementation code as provided to participants. The participants were tasked with testing the *getFormattedNumber* method.

Appendix B
Appendix: test code template for NANPPhoneNumber

Listing 3 shows the test code template as provided to participants.

Listing 1 Code Task: provided implementation code (1/2)

```java
import java.util.ArrayList;
import java.util.List;

/**
 * The North American Numbering Plan (NANP) is a telephone numbering
 * system used by many countries in North America like
 * the United States, Canada or Bermuda.
 * All NANP-countries share the same *international country code*: 1.
 * <p>
 * NANP numbers are ten-digit numbers consisting of a three-digit
 * Numbering Plan Area code, commonly known as *area code*,
 * followed by a seven-digit local number.
 * <p>
 * The first three digits of the local number represent the *exchange code*,
 * followed by the unique four-digit number which is the *subscriber number*.
 * <p>
 * Both the *area code* and the *exchange code* should not start with either a 0 or a 1.
 * <p>
 * Valid examples (which all produce "6139950253") are:
 * - "+1 (613)-995-0253"
 * - "613-995-0253"
 * - "1 613 995 0253"
 * - "613.995.0253"
 * <p>
 * Invalid numbers include (but are not limited by):
 * - "123456789" (too short)
 * - "+2 (613)-995-0253" (invalid country code)
 * - "112 345 6789" (area code should be in range 2-9)
 * - "212 045 6789" (exchange code should be in range 2-9)
 */
public class NANPPhoneNumber {

    private final String input;

    public NANPPhoneNumber(final String input) {
        if (input == null) {
            throw new ValidationException(ValidationException.NULL_PTR);
        }
        this.input = input;
    }

    /**
     * 1) First we filter out any character which is not a digit.
     * 2) Then we check whether we have 10 or 11 digits, any other amount of digits is invalid.
     * <p>
     * We are left with a number which can be represented by:
     * --------------
     * A BCD EFG HIJK
     * --------------
     * A       represents the optional country code 1 (often displayed as +1).
     * BCD     represent the 3 digit area codes.
     * EFG     represents the 3 digit exchange code.
     * HIJK    represent the 4 digit subscriber number.
     * <p>
     * 3) If we are working with a 11 digit number starting with a +1, we remove the country code.
     * <p>
     * 4) The area code and the exchange code must not start with a 0 or a 1.
     * We check for both codes if they start with a 0 or a 1 and signal failure if they do.
     * <p>
     * 5) The phone number is valid. We join the digits together as a string and return it.
     *
     * @return The 10 digit North American Number Plan phone number formatted
     * to only contain digits. The country code is NOT included.
     */
    public String getFormattedNumber() {
        char[] chars = this.input.toCharArray();

        // All tokens
        ArrayList<Character> tokens = new ArrayList<>(10);

        for (char currentChar : chars) {
            if (Character.isDigit(currentChar)) {
                tokens.add(currentChar);
            }
        }

        // Valid amount of number characters is 10 or 11.
        final boolean tenDigitsWithoutCountryCode = tokens.size() == 10;
        final boolean elevenDigitsWithCountryCode = tokens.size() == 11;
        if (!tenDigitsWithoutCountryCode && !elevenDigitsWithCountryCode) {
            throw new ValidationException(ValidationException.INVALID_LENGTH);
        }

        // remove the country code if we have eleven digits
        if (elevenDigitsWithCountryCode) {
            tokens.remove(0);
        }

        // check the area code: it can start with any digit except: 0 or 1.
        final int AREA_CODE_START_INDEX = 0;
        // check the exchange code: it can start with any digit except: 0 or 1.
        final int EXCHANGE_CODE_START_INDEX = 3;

        if (tokens.get(AREA_CODE_START_INDEX) == '0'
                || tokens.get(AREA_CODE_START_INDEX) == '1') {
            throw new ValidationException(ValidationException.AREA_CODE_INVALID_CHAR);
        } else if (tokens.get(EXCHANGE_CODE_START_INDEX) == '0'
                || tokens.get(EXCHANGE_CODE_START_INDEX) == '1') {
            throw new ValidationException(ValidationException.EXCHANGE_CODE_INVALID_CHAR);
        }

        return this.joinCharacters(tokens);
    }
```

Listing 2 Code Task: provided implementation code (2/2)

```
    /**
     * Join a list of characters together (without a separator) and format it as a String.
     *
     * @param sequence The list of characters which should be joined together and
     *                 formatted as a String.
     * @return The formatted String.
     */
    private String joinCharacters(List<Character> sequence) {
        StringBuilder sb = new StringBuilder();
        for (char c : sequence) {
            sb.append(c);
        }
        return sb.toString();
    }

    /**
     * An exception which represents failure to process a NANP number.
     * Should only be thrown when a NANP number is considered invalid.
     */
    public class ValidationException extends RuntimeException {

        static final String NULL_PTR =
                "A NANP phone number can't be null.";

        static final String INVALID_LENGTH =
                "A NANP phone number must have a length of 10 digits without" +
                        " the prefixed country code.";

        static final String AREA_CODE_INVALID_CHAR =
                "The area code must not start with a 0 or 1 character.";

        static final String EXCHANGE_CODE_INVALID_CHAR =
                "The exchange code must not start with a 0 or 1 character.";

        ValidationException(String message) {
            super(message);
        }
    }
}
```

Listing 3 Code Task: provided test code template

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

public class NANPPhoneNumberTest {

    @Test
    public void validNumberWithPunctuation() {
        String expectedNumber = "2234567890";
        String actualNumber = new NANPPhoneNumber("(223) 456-7890").getFormattedNumber();

        assertEquals(expectedNumber, actualNumber);
    }

    @Test
    public void invalidEmptyNumber() {
        assertThrows(NANPPhoneNumber.ValidationException.class, () -> {
            new NANPPhoneNumber("").getFormattedNumber();
        });
    }
}
```