Analysis of Test Log Information through Interactive Visualizations

Diego Castro
MARVEL-SE, Informatics and Comp. Science Dept.
Rio de Janeiro State University
Rio de Janeiro, RJ, Brazil
diegocbcastro07@gmail.com

Marcelo Schots
MARVEL-SE, Informatics and Comp. Science Dept.
Rio de Janeiro State University
Rio de Janeiro, RJ, Brazil
schots@ime.uerj.br

ABSTRACT

A fundamental activity to achieve software quality is software testing, whose results are typically stored in log files. These files contain the richest and more detailed source of information for developers trying to understand failures and identify their potential causes. Analyzing and understanding the information presented in log files, however, can be a complex task, depending on the amount of errors and the variety of information. Some previously proposed tools try to visualize test information, but they have limited interaction and present a single perspective of such data. This paper presents ASTRO, an infrastructure that extracts information from a number of log files and presents it in multi-perspective interactive visualizations that aim at easing and improving the developers' analysis process. A study carried out with practitioners from 3 software test factories indicated that ASTRO helps to analyze information of interest, with less accuracy in tasks that involved assimilation of information from different perspectives. Based on their difficulties, participants also provided feedback for improving the tool.

CCS CONCEPTS

• **Software and its engineering** → *Software defect analysis*; *Software testing and debugging*; Operational analysis;

KEYWORDS

Software testing, information visualization, test log analysis

ACM Reference Format:
Diego Castro and Marcelo Schots. 2018. Analysis of Test Log Information through Interactive Visualizations. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196321.3196345

1 INTRODUCTION

The increasing competition and interconnection in all kinds of markets have led organizations to shorten their time in launching new products. This is also true for software products. However, this search for speed in product creation should not affect its quality [21], since its failures may result in huge financial losses. A study performed by the US National Institute of Standards and Technology (NIST) estimates that software errors can end up costing about US$ 60 billion [26], reinforcing that assuring quality and reliability is crucial for the survival of software organizations.

To determine if the developed software product meets its specifications and works correctly in the environment for which it was designed, developers resort to software testing, whose overall goal is to increase the reliability of a program by checking whether it presents execution errors and revealing them for proper fixing [22].

The testing process can be organized into three main categories: *test planning* (involves the production of the test plan), *test monitoring and control* (verifies if the test progress is in accordance with the test plan), and *test completion* (aims to make useful test assets available for further use) [14]. There are also generic processes for performing dynamic tests (called *dynamic test processes*), comprising *test design and implementation*, *test environment setup and maintenance*, *test execution*, and *test incident reporting* [14]. The latter step deserves special attention, since it is the communication point between software testers and the other stakeholders.

The *test incident reporting process* results from events during the test execution, e.g., the identification of failures or an unusual or unexpected behavior. It involves assessing the results, confirming the incidents, generating the incident report, and communicating results to the stakeholders [14]. During this process, software testing tools usually produce (or can be configured to produce) auxiliary test files known as *log files*, i.e., documents that contain descriptive execution details of one or more test procedures, comprising data on their actual execution status, in chronological order [12].

According to Valdman [27], the information reported in log files is considered by developers as the most important and useful resource to monitor software quality and highlight failures, easing the debugging process. Such files usually describe a test description (if applicable), the test duration, errors identified in the test execution, incident statuses, context information, among others [15].

Some log files are small, so one does not take too long to understand them. However, due to their rich and detailed information, many log files are large, with a complex structure, making the understanding of their information a challenging task [27] [7]. As the volume of data gets larger, analyzing their contents becomes inefficient, hampering developers in properly understanding failures and their causes [27]. To overcome this, one can use software visualization techniques, i.e., methods developed to graphically represent different aspects of a software, facilitating its comprehension [8].

This work parts from the assumption that *the collection and abstraction of information from log files can ease and speed up the* *identification of problems identified in test executions*. In this sense, this paper presents ASTRO (Analysis of Software Test Results and Outputs), a tool that extracts information contained in log files and represents it through visual metaphors, aiming at supporting developers in understanding errors in their developed applications.

The remainder of this paper is presented as follows. Section 2 analyzes related works, Section 3 depicts ASTRO and its features, Section 4 shows an example of its use, Section 5 presents an evaluation of the tool, and Section 6 concludes with the final remarks.

2 RELATED WORK

Nowadays, many testing tools present some sort of visual interface trying to improve the understanding of the data contained in log files. Particularly, some tools (known as log file analyzers or log file visualization tools) can analyze a log file with a specific structure and produce human-readable summary reports. Some of these tools are analyzed as follows.

Jones et al. [17] analyzed test case results with their tool (named Tarantula), generating a colored view of the source code: each line of code is colored using either a discrete three-color mapping or a continuous level of varying color (hue) and brightness, to indicate how each source code statement participates in each passed/failed test case [17]. The tool provides mouse-based interactions with details on demand, besides controlling the grayscale brightness of some lines. For larger source code files, it can be hard to distinguish the severity of some failures if the colors of close lines are similar.

For improving the understanding of test logs of online applications, Din et al. [9] proposed a XML-based graphical tool. The graphic is a model (TraceViewer) formed by 5 different types of elements, each one representing a different task: an instance of an object, a test component, the start of an application, the finishing of an application, and a message exchange [9]. Their proposed tool shows a sequence of all the activities that were performed within an application test case, in a static visualization. The only interaction capability identified is the control of pages through arrows.

Gouveia et al. [11] created Gzoltar, a tool that aims to improve the representation of information generated by test cases. Using the program spectrum technique [1], it allows capturing the code snippets where the error may be. The program classifies the lines into 4 different types: low suspicion, medium suspicion, high suspicion, and very high suspicion. In the end, Gzoltar generates 3 types of graphs, all representing the same kind of information: Sunburst, Vertical Partition, and Bubble Hierarchy [11].

The Sunburst hierarchically organizes a project into packages, files, classes, methods, and lines. Due to that, its outermost layer is cluttered, making it hard to get a clear insight of the results. The Vertical Partition uses rectangles instead of arcs to represent each node so that sub-nodes of each node are drawn directly under it. The cluttering problem remains, and it is not possible to know where exactly each node begins and ends, also hampering to understand the meaning of the visualization and its information. Finally, the Bubble Hierarchy applies the concept of containment instead of adjacency, using circles to represent each solid node (each sub-node is drawn inside the area that represents the parent node) [11].

Lingo3G is a real-time text clustering engine created by Osinski and Weiss [23]. One of its uses includes a unit test coverage visualization[1], used with a search tool to visually demonstrate software test results. Lingo3G provides two views: a Treemap and a Polygonal Treemap. Besides lacking more explanatory legends for some visual attributes (e.g., for each color presented), the tool only presents results that failed or passed, not taking into account elements that were skipped and ignored. In addition, the tool self-description states that the size of polygons is related to the level of complexity of the class or package it represents, but the way how this complexity is defined it not clear. Finally, each polygon has a different shape in the tool, making it hard to perform comparisons.

Cornelissen et al. present Extravis, which uses an execution trace for generating two views: a circular bundle view, showing the system's structural decomposition and interactions, and a massive sequence view, displaying consecutive calls between system's entities in chronological order. It provides rich interactions for exploring the execution trace. The circular bundle suffers from the same cluttering problem of Gouveia et al.'s Sunburst. Extravis does not provide a general overview of test results first, hampering a more immediate perception for prioritizing error handling.

All these approaches and tools analyzed intend to improve the interpretation of test data by visualizing them under a certain perspective. The approach we propose in this work is directed towards improving interaction capabilities, providing three different linked perspectives, with legends and/or tooltips for every visual attribute, seeking a more intuitive and user-centered analysis process. Similarly to other tools, our approach targets an intermediary level of granularity (classes and methods), so that details can be handled later using a debugger or another tool, while providing a more comprehensive report with information about passed, failed, skipped, and ignored tests. Finally, because most tools are limited to a particular file structure (a common limitation highlighted by Valdman [27]), we aim at increasing compatibility with other log file formats.

3 ASTRO: VISUAL ANALYSIS OF TEST LOGS

As stated in Section 1, log files have the main information about a particular test that was run, but such files can have a complex structure with large volumes of data, hampering the understanding of its information and making its analysis a time-consuming, difficult, and challenging task. A log file lists the problems that occurred within a program execution, which may end up not being verified and remain hidden if the log files are not properly analyzed.

The goal of ASTRO (*Analysis of Software Tests Results and Outputs*) is to facilitate the analysis and comprehension of the information contained in unit test log files, enabling a better understanding of this information in a quick, interactive, and practical way. To achieve such goal, we developed an integrated tool infrastructure that extracts data from log files generated by unit testing tools, providing a set of visualizations to facilitate the data analysis. The next subsections describe the methodology followed in this work.

3.1 Organization of Information Presented by Log Test Files

We performed a study to identify what kinds of logging information testing tools can provide. To restrict our search scope, the first cut-off criterion was that the testing tool should support software

[1]https://get.carrotsearch.com/foamtree/latest/demos/coverage.html

projects developed in the most popular programming languages and generate a log file, so that ASTRO could comprise more projects over time and increase the number of potential users[2]. Thus, we consulted two available rankings in order to identify the top-3 languages used by developers. The first ranking [13] listed JavaScript, Java, and Python, whereas the second one [10] listed Python, C, and Java. Therefore, the candidate testing tools to be evaluated should support projects developed in JavaScript, Java, Python, or C. We discarded tools that did not provide unit test logs.

Another cut-off criterion was the number of stars in the GitHub repository. The GitHub stars (assigned to a GitHub repository by developers) were used as a measure of popularity (as in other works, e.g., [2]). The average number of stars of all the candidate tools was ≈ 680, but the C-language tool project with the most stars had only 506. Taking this into account, the minimum number of stars was defined as 500. In total, 21 tools were selected for analysis.

We created a separate table for each selected testing tool to organize details on the log information it can generate. Fields include a metadata identifier, a description of its meaning, and an example of the data generated by the tool. Then, we created another table for mapping the whole set of tools and the information each can generate in their log files. From the 21 evaluated testing tools, we identified a total of 112 different types of log information. Based on this analysis, it became possible to identify what information is considered more relevant for the analysis of a test log.

Table 1 presents an excerpt of these data, showing the 15 kinds of information most frequently found in the analyzed test log files, among which we highlight the following: the given name for the test case, the date and time of its execution, the result of each test, the error message (if any), the number of executed tests, the total duration of tests, and the number of tests that passed/failed.

3.2 Choice of Visual Metaphors

We chose the visualization techniques based mainly in the nature of the data to represent. There are (and we intend to try) other visualization metaphors that could fit the target data (e.g., a treemap could replace the bubble chart), but our main concern was to find a visual arrangement that could properly depict the relationship between the data and communicate it effectively to the user.

To define what visualizations ASTRO should employ, we carefully mapped the data extracted from log files to visual abstractions that could properly represent them. As a support, we used a mapping structure [25] that presents a series of stages for matching specified goals with visualizations that potentially help achieving them.

When designing a visualization tool, one of the most crucial parts is to map how each datum must be visually presented to the user, since an inappropriate mapping can likely hamper the expected effectiveness of the tool. Figure 1 depicts the mapping of the main elements to their corresponding visual representations. In some cases, multiple visualization techniques map the same data attributes to better support different tasks. For instance, the bubble chart depicts better how different results are spread in the test cases, while the bar chart provides a more quantitative view of the same results. More details of the mapping can be found in [5].

[2]The analysis aimed at identifying log information (not tools) ASTRO should be compatible with. Each log format needs its driver/parser, not to be provided by the tool.

3.3 Elaboration of Visualization Perspectives

We organized visualizations into three perspectives – Overview, Callers & Callees, and History – according to the kind of information each one represents. These perspectives share the same status colors associated with the test case results: green for passed tests, red for failures, yellow for skipped tests, and blue for ignored ones. They are also interactive, with tooltips, filters and dynamic legends based on mouse interactions. Perspectives are described as follows:

3.3.1 *Overview.*

The Overview perspective (depicted in Figure 2) presents to the user a quick summary of the test case executions, providing a general report of the status of each test performed. It integrates a bubble chart, a bar chart, and a pie chart.

The bubble area is recursively divided into subareas, so that the bubble chart represents tests in a hierarchical way: the topmost hierarchy level (the biggest bubble) is the set of all test cases. Each element in the next level represents a test class (i.e., a test case), which in turn encloses its test methods, the lowest level of the hierarchy. For optimizing space, if a test class has only one test method, the method is drawn directly without its corresponding class (which still can be accessed through a tooltip). The test method execution time determines the bubble size.

The bar chart is organized into class groups, i.e., methods (tests) belonging to the same class (test case) stay close to each other. Each bar on the x-axis represents a test, and the y-axis displays how long each test took to run. The pie chart quickly summarizes the total number of run tests, including those with execution time close to 0.

3.3.2 *Callers & Callees.*

The Callers & Callees perspective (shown in Figure 3) aims at providing an understanding of what classes and methods were involved in an error, based on the stack trace data (which by default only presents methods that failed, but can also show skipped and ignored tests if the system is configured to do so). It shows classes and methods that were called before and after a selected method execution, as well as their status (consistently using the same color scheme of the other perspectives), helping the user to know where the error occurred and where it was spread.

This perspective comprises two visual abstractions: the *Method-Oriented View (MOV)* and the *Class-Oriented View (COV).*

The MOV shows all the methods that were run when a test happened. Its main function is to demonstrate the order of execution of a selected method. When a method of interest is selected, the links between the methods are highlighted, showing the methods that either were called in its execution or called it at some point of their execution. By selecting a method, the methods that were run before it are colored light blue, while the ones that were run after it are painted dark blue. The order of execution of the methods is displayed from left to right, i.e., the leftmost ones were executed first and the rightmost ones were executed last.

The COV is organized as follows: its center contains the test methods as rectangles, surrounded by the classes that contain each method in the failed call stack. When an element is hovered, the associated test methods are painted according to their status, and their owning classes are highlighted. It is worth noting that an ignored or skipped method is not connected to any class but the one it belongs to. Besides, when a method fails and calls another method

Table 1: Log file information

Metadata identifier	Description	Example (taken from one of the tools, for illustration purposes)
Test name	Given a name for the test	test.number1
Test result	Result of the test	Failed
Test duration	Duration of the test	1900795
Passed test	Number of passed tests	3
Failed test	Number of failed tests	5
Test case name	Given a name for the test case	testCase.number1
Test case result	Result of the test case	Passed
Test case duration	Duration of the test case	140812247426
Passed test case	Number of passed tests case	2
Failed test case	Number of failed tests case	9
Test suite name	Given a name for the test suite	model.allure
Passed test suites	Number of passed tests suites	5
Failed test suites	Number of failed tests suites	1
Error message	Message that explains the error	AssertionError: Expected 10 less than 8
Stack trace	Program stack trace	Exception in thread "main" java.lang.NullPointerException at (...)

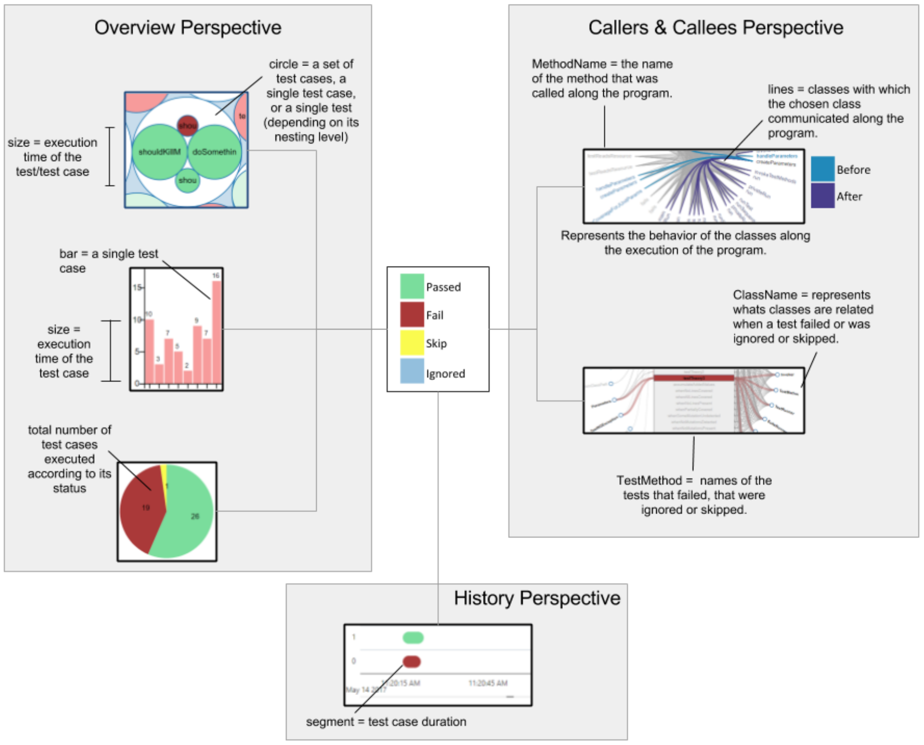

Overview Perspective
circle = a set of test cases, a single test case, or a single test (depending on its nesting level)
size = execution time of the test/test case
bar = a single test case
size = execution time of the test case
total number of test cases executed according to its status
Passed
Fail
Skip
Ignored
Callers & Callees Perspective
MethodName = the name of the method that was called along the program.
lines = classes with which the chosen class communicated along the program.
Before
After
Represents the behavior of the classes along the execution of the program.
ClassName = represents whats classes are related when a test failed or was ignored or skipped.
TestMethod = names of the tests that failed, that were ignored or skipped.
History Perspective
segment = test case duration

Figure 1: Mapping between data and visualizations

that did not fail, it is shown in the MOV (without its execution status), but not in the COV (due to the aforementioned reasons).

Both views organize methods according to the classes they belong to: lines between methods divide classes in the COV, whereas empty space gaps between methods separate classes in the MOV.

3.3.3 History. The History perspective (shown in Figure 4) aims at organizing the information of the test cases chronologically, providing the order of execution of each test method, how long they took to execute, and the exact date and time of their execution.

The log produced by running test cases of a software version show when bugs were introduced, and a repository analysis outside ASTRO can give additional context information, especially in a continuous integration scenario, producing test logs at each commit.

The History perspective is especially useful when the program under analysis suffers from *flaky tests*, i.e., tests with non-deterministic

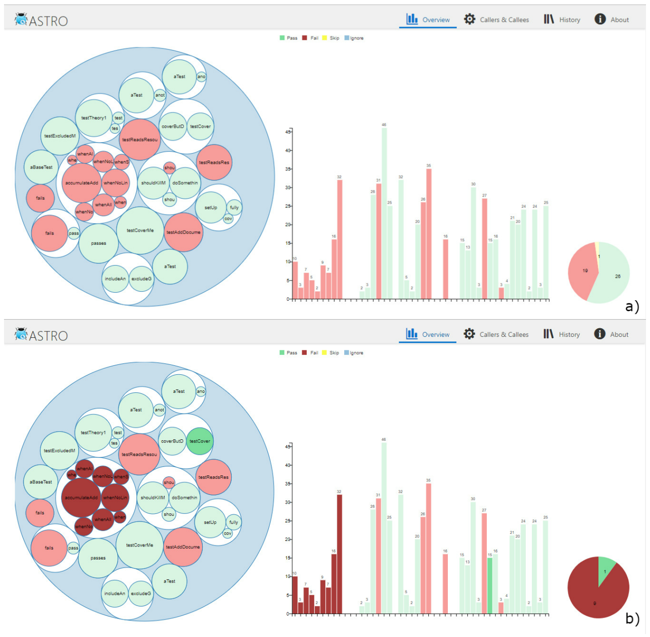


Figure 2: ASTRO Overview perspective, before (a) and after (b) selecting the test methods

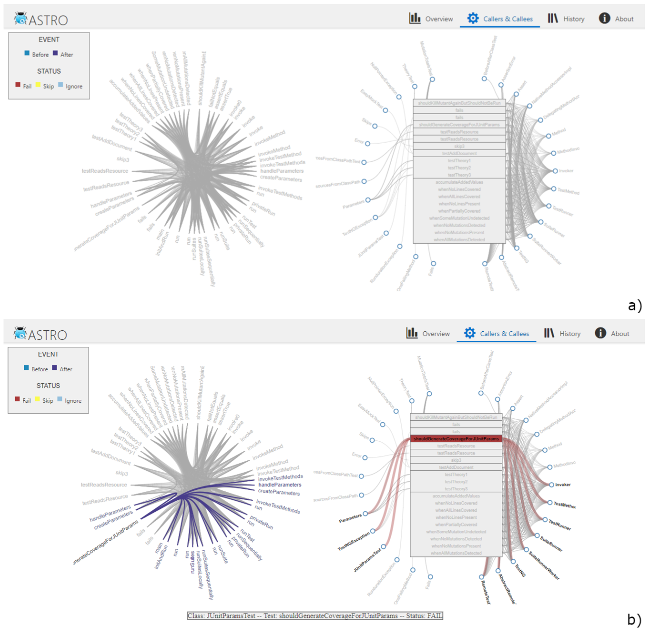
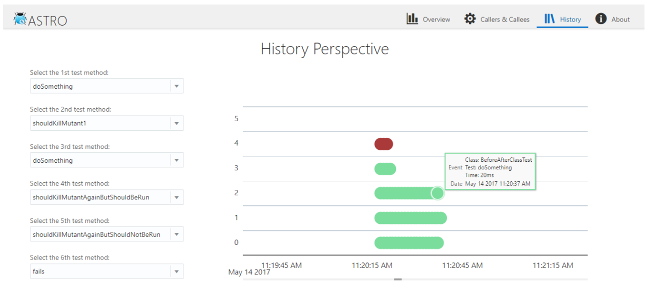

Class: JUnitParamsTest -- Test: shouldGenerateCoverageForJUnitParams -- Status: FAIL

Figure 3: ASTRO Callers & Callees perspective, before (a) and after (b) selecting the test method

outcomes that can intermittently pass or fail even for the same code version [20]. Since time is a root cause of flakiness (e.g., relying on the system time, affected by time zones and in platform-dependent formats [20]), the History perspective can help comparing tests executed in different points in time and detecting such flaky tests.


History Perspective

Select the 1st test method: doSomething
Select the 2nd test method: shouldKillMutant1
Select the 3rd test method: doSomething
Select the 4th test method: shouldKillMutantAgainButShouldBeRun
Select the 5th test method: shouldKillMutantAgainButShouldNotBeRun
Select the 6th test method: fails

Class: BeforeAfterClassTest
Event Test: doSomething
Time: 20ms
Date May 14 2017 11:20:37 AM

Figure 4: ASTRO History perspective

Through the History timeline, one can visualize the entire project time frame and perform a post-mortem analysis, e.g., to identify and understand key moments in which errors occurred and, based on this, mitigate reasons to avoid such undesirable events.

3.4 Implementation

The back-end of ASTRO, built in Java, is primarily responsible for extracting the data from the test log files generated by the testing tools in their different formats. A dictionary-like structure maps the reserved words identified for each log file ASTRO can interpret (based on the study described in Section 3.1), fitting different words with the same meaning (e.g., `errored` and `errors`) into a single unique word. The mapping matches are organized into a JSON file structure and stored in a MongoDB database.

The current implementation of ASTRO only analyzes files produced by JUnit[3], TestNG[4], and Cucumber[5], covering all the information contained in test log files generated by these tools. However, it can be easily extended to support log files produced by other testing tools to increase the tool coverage. The front-end, built in JavaScript, uses D3.js [3] to generate the visualizations. D3.js contains several available examples compatible with different devices.

4 ILLUSTRATING EXAMPLE

To demonstrate a practical example of ASTRO, we applied it to the source code of Pitest (PIT Mutation Testing)[6], a Java mutation testing tool that works at the bytecode level, i.e., it creates mutants without touching the application source code. After PIT's execution has finished, it provides detailed information on mutants created and killed. It also creates an HTML report showing the line coverage and a mutation [24] [18]. We chose Pitest because it is a popular open source project[7] with available test cases.

Mutation testing measures how "good" tests are by inserting faults into the program under test. Each fault generates a new program, a mutant that is slightly different from the original. The idea is that the tests are adequate if they detect all mutants [4]. In mutation testing, faults (or mutations) are automatically seeded into the code, then the tests are run. If the tests fail, then the mutation is killed; otherwise, the mutation remains [24].

[3]https://junit.org/
[4]http://testng.org/
[5]https://cucumber.io/
[6]http://pitest.org/ – source code available at https://github.com/hcoles/pitest
[7]628 stars, 23 releases and 34 contributors on 8 February 2018.

Table 2: PIT types of mutation method

Class	Type
AnnotatedClassLevel	Inline Constant Mutator
AnnotatedBase	Increments Mutator
AnnotatedAtMethodLevel	Increments Mutator
BeforeAfterClassTest	Return Values Mutator
CoveredButOnlyPartiallyTestedTest	Return Values Mutator
CoveredByABeforeAfterClassTest	Inline Constant Mutator
EasyMockTest	Constructor Calls Mutator
Fails	Return Values Mutator
FullyCoveredByTestNgTesteeTest	Return Values Mutator
HasExcludedMethodsTesteeTest	Return Values Mutator
HasGroups	Return Values Mutator
HasOnePassingAndOneFailingMethod	Return Values Mutator
LoadsResourcesFromClassPath	Return Values Mutator
LoadsResourcesFromClassPathTest	Returns Values Mutator
MutationTotalTest	Inline Constant Mutator
PartCoveredTest	Inline Constant Mutator
Passes	Return Values Mutator
Skips	Void Method Call Mutator
StaticMethod	Return Values Mutator
TheoryTest	Inline Constant Mutator

Each Pitest class has a type of mutation test method, as listed in Table 2. Pitest offers mutation methods for different test types [24]:

- **Increments Mutator**: mutates increments, decrements, and assignment increments and decrements of local variables (stack variables), replacing increments with decrements and vice versa;
- **Return values mutator**: returns values of method calls; depending on the method return type, another mutation is used;
- **Constructor calls mutator**: replaces constructor calls with null values;
- **Inline constant mutator**: assigns a literal value (i.e., an inline constant) to a non-final variable; depending on the inline constant type, another mutation is used;
- **Non-void method call mutator**: removes method calls for non-void methods; their return value is replaced by the Java default value for that specific type.

We ran the project test cases[8] and used the TestNG framework to generate the test case logs. ASTRO extracted the data from the XML files for generating the visualizations, allowing to better explore and understand the data from the log files. The data depicted in Figures 2, 3 and 4 refer to the Pitest project.

Using the Pitest example, it is possible to obtain a large amount of information about test results simultaneously from the ASTRO tool perspectives, without having to analyze the whole test log. One can detect in a glance what methods that passed the tests, what methods have errors, and what methods were ignored, as well as checking their dependencies and the duration of tests.

[8]Based on revision 5a68857, committed on 12 May 2017.

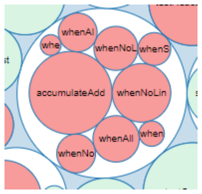


Figure 5: Bubble chart showing the `MutationTotalsTest` class

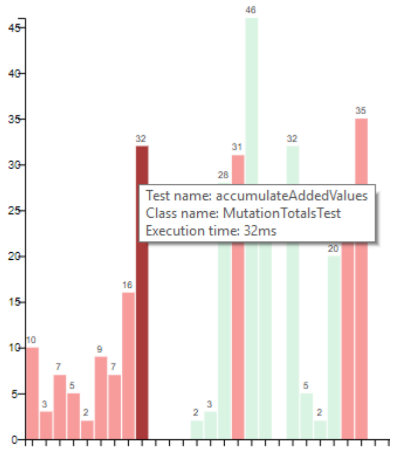


Figure 6: Bar chart showing the `MutationTotalsTest` class

The pie chart of the Overview perspective (Figure 2) shows that 19 (out of 46) test cases failed, 1 was ignored, and 26 executed smoothly (the bar and bubble charts do not show tests with execution time less than 1ms). Yet in this perspective, it is possible to notice that most of the tests that failed are located in the `MutationTotalsTest` class (as shown in Figure 5): 9 out of the 19 cases that failed belong to this class. This means that a large part of the application's problems seem to lie within this class, which can be a starting point for analyzing and fixing bugs.

When analyzing (in the Overview perspective) methods in terms of their execution time, the bubble chart provides some clues in terms of their hierarchical structure, while the bar chart presents a more time-oriented comparative view (see Figure 6, depicting (for instance) that (i) the two largest execution time were 46ms and 35ms, and (ii) the former passed the test, while the latter did not. The interaction reveals in a tooltip that such methods are `testCoverMe` and `testReadsResource`, respectively. Depending on domain- or application-specific performance requirements, one could assume that the obtained execution time for the first method is prohibitive, and take action in this regard.

In the Callers & Callees perspective, one can identify what methods and classes are in the sequence of calls that resulted in a test fail. For example, Figures 7 and 8 illustrate the `accumulateAddedValues` method (from the `MutationTotalsTest` class), which failed when called by `failNotEquals`, which was called by `assertEquals` (both methods from the `Assert` class), called in turn by `invoke0` (from `NativeMethodAccessorImpl`), and so forth, up to the starting call

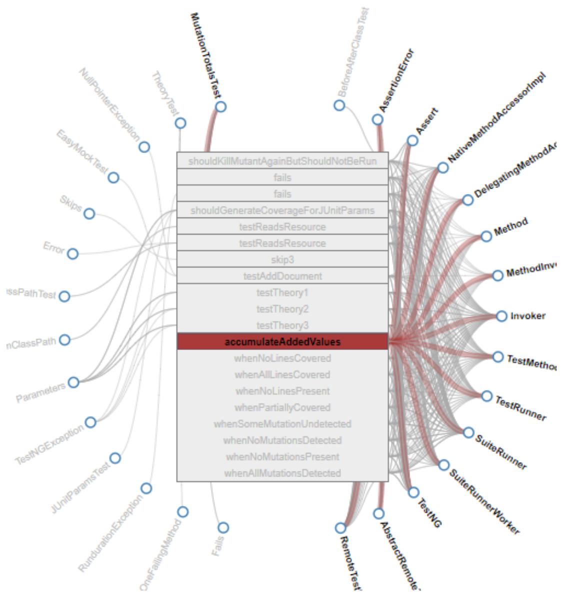


Figure 7: Class-Oriented View (COV), highlighting the `accumulateAddedValues` method

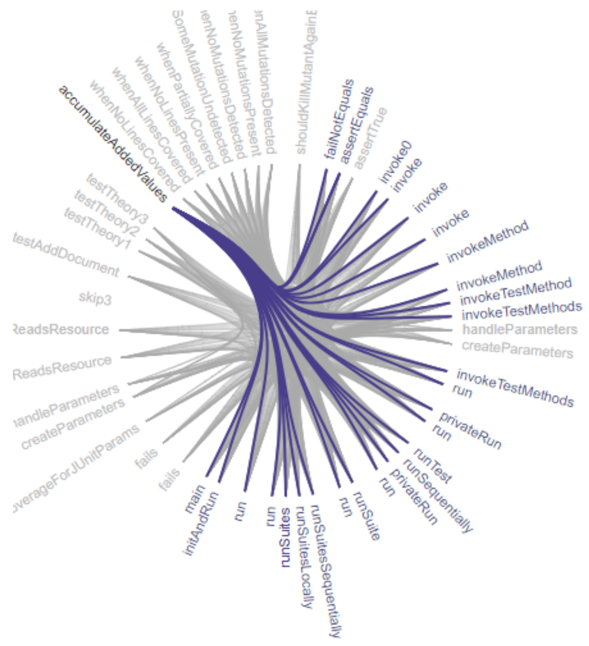


Figure 8: Method-Oriented View (MOV), highlighting the `accumulateAddedValues` method

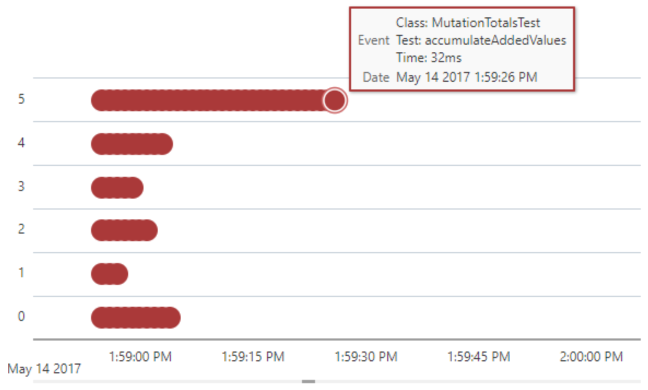

Event Class: MutationTotalsTest
Test: accumulateAddedValues
Time: 32ms
Date May 14 2017 1:59:26 PM

Figure 9: Timeline showing the execution of methods from the `MutationTotalsTest` class

(`initAndRun`, from `RemoteTestNG`). Individual status information is shown on demand in a separate text area, as shown in Figure 3.

By analyzing the sequence of methods that have been called, it is possible to identify a wrong method invocation or an unexpected execution order, leading to further investigation. The example depicted in Figure 8 shows homonymous and/or similar methods that could have been mistaken during the implementation (e.g., `invoke`, `invokeMethod` or `invokedTestMethods`).

Finally, the History perspective shows the execution of methods from the `MutationTotalsTest` class, in chronological order. For illustration, 6 methods were selected in the timeline shown in Figure 9. This visualization shows that the method `accumulateAddedValues` took 32 ms to execute, starting on 14 May 2017 at 1:58:54 PM and finishing some milliseconds later, at 1:59:26 PM.

By navigating into this perspective, based on each segment's horizontal position and length, one can assess whether the intended execution order actually happened, and whether there is a "hidden" pattern between failed tests and the order or time of execution (e.g., due to synchronization issues). Taking a hypothetical example, a failed method may have accessed information that should have been previously set by another method at some point, but based on the timeline information, the latter method had not set it yet.

5 EVALUATION

The goal of this evaluation is to understand to which degree the target user could benefit from the use of ASTRO, in terms of its usefulness, by performing day-to-day tasks correctly and in a short time. Because the goal of this study is a characterization, we did not perform any comparison with another tool. To perform the evaluation, we elaborated some tasks that should be answered using the information visually provided by the tool, reusing the Pitest example presented in Section 4. These tasks were divided into three levels of difficulty (as proposed by Knodel et al. [19]):

- *Filtering Tasks (FT)*: due to their simplicity, every target person should be able to perform them (otherwise, there might be general problems hampering the person's successful participation in the study, so the other results can be bypassed);
- *Basic Tasks (BT)*: involve a basic understanding of the tool mechanisms, not requiring much rationale; and
- *Assimilation Tasks (AT)*: require reasoning about different kinds of information provided by the tool to find an answer.

Table 3: Pilot study task results by participant

	Participant 1		Participant 2	
Questions	Time	Answer	Time	Answer
FT1 - a)	0.4s	Wrong	3.3s	Right
FT1 - b)	1.3s	Wrong	4.1s	Right
FT1 - c)	1.3s	Right	2.1s	Right
FT2	1.2s	Right	1.3s	Right
BT1 - a)	1.5s	Right	0.9s	Right
BT1 - b)	1.0s	Right	1.2s	Right
BT1 - c)	4.2s	Right	5.3s	Right
BT1 - d)	5.1s	Wrong	4.1s	Right
BT1 - e)	1.3s	Wrong	1.0s	Right
BT1 - f)	1.2s	Wrong	1.1s	Right
BT1 - g)	1.0s	Wrong	1.2s	Right
BT2	1.4s	Right	1.6s	Wrong
BT3	2.0s	Right	1.2s	Right
AT1	2.8s	Right	1.8s	Right
AT2	2.3s	Right	2.7s	Right
AT3	3.6s	Right	3.1s	Wrong
AT4	3.3s	Wrong	2.3s	Right

Participants were asked to fill out a characterization questionnaire and a follow-up questionnaire, before and after performing the tasks, respectively. The former is used to help us understand potential confounding factors that may justify unexpected results.

5.1 Pilot Study

A pilot study was conducted with two Computer Science undergraduate students, none of which involved with the development of ASTRO. Their evaluation sessions were audio recorded with their consent. They were instructed to express themselves according to the think-aloud protocol [16].

The pilot study sessions were performed in a room free of interruptions. The study was run on a 14-inch notebook, with a Core i7 processor, 2.40GHz CPU, 6GB RAM, with Windows 10, with the maximum brightness level (to facilitate color differentiation). All study sessions were run on the same machine to avoid problems in differences in hardware performance. In addition, all participants used the same initial display settings. The average execution time by each participant was around 37 minutes, which allowed to estimate an average time for the participants of the actual study. The execution time for each task by participant was also accounted to know in average how long it would take to perform each of the proposed tasks. This is shown in Table 3, along with the results.

After the pilot study, participants informed that the wording of some questions was too verbose, and needed to be shortened. Besides, some questions were also considered ambiguous. With respect to the accomplishment of the tasks, the first participant pointed out that some visualizations used very similar colors, making it difficult to differentiate between them. This participant also asked for more explanatory legends. The other participant asked for a faster method to transition between perspectives. Finally, both participants pointed out a problem in integrating the menu into

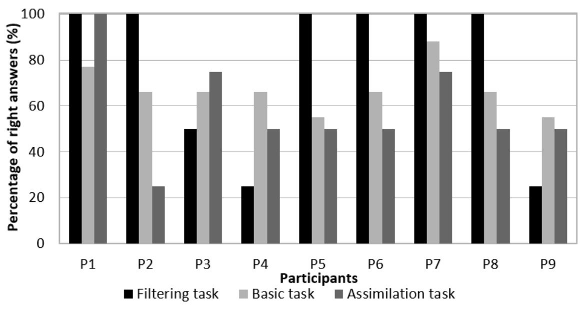


Figure 10: Percentage of right answers by each participant, grouped by task levels

the History perspective. The suggested modifications were incorporated to ASTRO and the questionnaires before executing the actual study. The final task list can be found in Appendix A.

5.2 Study Conduction

The study was carried out with 9 employees from 3 software development companies that work as test factories. Again, none of them were related to the development of ASTRO. As in the pilot study, all participants were instructed to express themselves according to the think-aloud protocol [16]. Among the participants, 2 preferred not to be audio recorded. In this case, the audio recording was replaced by note-taking, and the time was not accounted.

All 9 participants filled out the characterization questionnaire. 7 of them had completed an undergraduate course and 2 are in the final stages. 6 participants reported having planned/performed tests as part of a team in the industry, 7 have at least read some material on information visualization, 4 claimed to have good knowledge about both unit testing and Java, and 5 declared good knowledge about software inspection and object orientation. Only 5 participants stated they use log files to identify errors in their programs, 6 consider log analysis a difficult task, and yet, 6 consider the log file extremely relevant to finding problems in a particular program.

After answering the characterization questionnaire, participants executed the tasks (not simultaneously), using the same configuration used in the pilot study, but in a different place. The results are summarized in Figures 10 and 11, and detailed in Table 4[9].

It can be observed that 3 out of the 9 participants did not answer all the Filtering Tasks correctly. Although this can be considered as a cut-off criterion ([19]), we decided to keep their results, due to the low number of participants. For Basic Tasks, one participant had around 89% of correct answers, one achieved around 77%, other 5 had 66% of correct answers, and 2 achieved 55%. For Assimilation Tasks (the most difficult ones), one participant answered 100% correctly, 3 had 75%, 4 had 50%, and one had 25% correct answers. The total time spent varied between 14.4 (minimum) and 27.6 minutes (maximum), with an average of 20.03 and a median of 19.9.

Most Filtering/Basic questions were answered correctly. Only one participant gave a right answer to BT4 (related to the History perspective) – which agrees with the difficulties reported for this

[9]There is no time and accuracy information for questions FT3 and BT4 because they were only used for confirming the visualizations used.

Table 4: Study task results by participant

	Participant 1		Participant 2		Participant 3		Participant 4*		Participant 5		Participant 6		Participant 7		Participant 8		Participant 9*	
Questions	Time	Answer	Time	Answer	Time	Answer	Time	Answer	Time	Answer	Time	Answer	Time	Answer	Time	Answer	Time	Answer
FT1 - a)	0.5s	Right	1.1s	Right	1.1s	Wrong	N/A	Wrong	1.4s	Right	1.0s	Right	1.0s	Right	1.3s	Right	N/A	Wrong
FT1 - b)	0.7s	Right	0.8s	Right	1.0s	Right	N/A	Wrong	1.2s	Right	0.8s	Right	1.0s	Right	1.4s	Right	N/A	Wrong
FT1 - c)	0.5s	Right	0.9s	Right	1.1s	Right	N/A	Wrong	1.4s	Right	0.8s	Right	0.7s	Right	1.1s	Right	N/A	Wrong
FT2	0.5s	Right	0.8s	Right	1.0s	Wrong	N/A	Right	1.6s	Right	0.9s	Right	1.1s	Right	1.3s	Right	N/A	Right
BT1 - a)	0.7s	Right	0.8s	Wrong	1.0s	Right	N/A	Wrong	1.6s	Right	1.0s	Right	1.2s	Right	1.5s	Right	N/A	Wrong
BT1 - b)	0.6s	Right	0.9s	Right	0.9s	Right	N/A	Right	1.0s	Right	1.0s	Right	0.9s	Right	1.1s	Right	N/A	Wrong
BT1 - c)	2.1s	Right	1.5s	Wrong	2.3s	Wrong	N/A	Right	4.1s	Wrong	1.0s	Wrong	3s	Right	2.3s	Wrong	N/A	Wrong
BT1 - d)	2.3s	Wrong	1.8s	Right	2.1s	Wrong	N/A	Right	4.1s	Wrong	2.1s	Wrong	2.9s	Right	2.7s	Wrong	N/A	Right
BT1 - e)	0.7s	Right	0.9s	Right	0.9s	Right	N/A	Right	1.1s	Right	0.9s	Right	1.1s	Right	1.2s	Right	N/A	Right
BT1 - f)	0.7s	Right	1.0s	Right	0.8s	Right	N/A	Wrong	1.0s	Right	0.8s	Right	0.7s	Right	1.1s	Right	N/A	Right
BT1 - g)	0.7s	Right	0.8s	Right	0.6s	Right	N/A	Right	1.1s	Wrong	0.8s	Right	0.6s	Right	1.2s	Right	N/A	Right
BT2	0.7s	Right	0.8s	Right	0.6s	Right	N/A	Right	1.1s	Wrong	0.7s	Right	1.1s	Right	1.0s	Right	N/A	Right
BT3	0.7s	Wrong	0.9s	Wrong	0.9s	Wrong	N/A	Wrong	1.1s	Right	1.1s	Wrong	1.1s	Wrong	1.0s	Wrong	N/A	Wrong
AT1	0.6s	Right	0.7s	Wrong	1.1s	Right	N/A	Right	1.1s	Right	1.0s	Right	0.9s	Right	1.0s	Right	N/A	Right
AT2	0.5s	Right	0.8s	Right	1.1s	Right	N/A	Right	1.1s	Right	1.1s	Right	0.8s	Right	1.1s	Right	N/A	Right
AT3	0.5s	Right	0.9s	Wrong	1.1s	Wrong	N/A	Wrong	1.5s	Wrong	1.1s	Wrong	1.1s	Right	1.2s	Wrong	N/A	Right
AT4	1.4s	Right	1.1s	Wrong	2.3s	Right	N/A	Wrong	2.1s	Wrong	1.6s	Wrong	1.4s	Wrong	2.1s	Wrong	N/A	Wrong

[1]Note: Participants indicated with asterisk (*) did not have their time accounted.

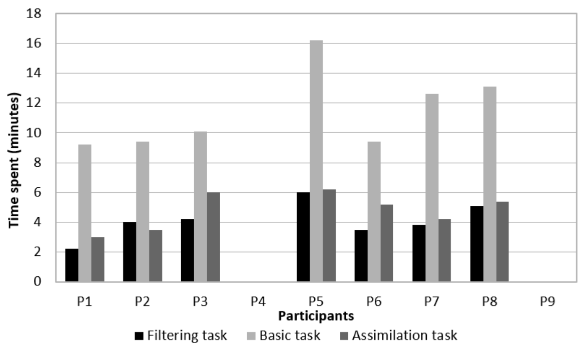


Figure 11: Task execution time by participants, grouped by tasks' level of difficulty

perspective. Participants also had problems answering AT3 and AT4, which required combining information from different perspectives.

The follow-up results showed that most of the participants considered the tool useful and easy to use: "it presents in an aggregated way a large amount of useful information that would be difficult to obtain from the analysis of an extensive textual log file". One of the participants who works with test reports stated, "the tool would help a lot in the company activities". Yet, it can be observed that some participants who only had knowledge about manual tests and little knowledge with visualization had more difficulties.

The perspective that was considered the most relevant was the Overview, by 5 participants. The reasons include "a good integration between the elements within the visualization" and "the fact that the visualization can exhibit the information of all the tests at once, allowing to know in an easy and intuitive manner how many methods achieved each status". The second most important was the Callers & Callees, by 4 participants. The main reason was "its innovative way to demonstrate the stack trace generated by the test methods, from being something textual to something visual and intuitive". The History perspective was considered the least relevant among the three, and participants with little knowledge about programming languages and object orientation had more difficulty. Some participants would like to have viewed all test cases at once, but the version used in the study only shown 6 at a time.

Among the improvements pointed out, the main ones were (i) to create more captions explaining better each information inside the visualizations and (ii) the modification of the History visualization so that it displays all the methods simultaneously. These suggestions are already being implemented in the ASTRO tool.

Among the 3 participants who did not achieve good results, 2 reported having little knowledge about object orientation and visualization, which is a crucial point for understanding how ASTRO works. Besides, 2 out of the 3 were working with testing for only 1 year. This may have impacted their results in the evaluation.

From the study results, it was possible to observe that ASTRO supported a general interpretation of the errors identified by the tests. The Overview was the most praised perspective, which provides the results of all tests at a glance, informing the status of each one, their execution time, and possible indicators for further actions.

5.3 Threats to validity

Some aspects may have influenced the study. Since it was executed inside the companies, some problems were observed that did not occur in the preliminary study.

- Two participants did not authorize the audio recording; this may have hampered identifying relevant information.
- In real workplaces, external factors can distract participants while performing the tasks. Some study executions recordings needed to be paused and then un-paused, because the participant had to speak with another person in the company. While the tool is expected to be used in these scenarios, this may have added confounding factors to the study.

- The preoccupation by participants with the time of the study may have hampered the execution of the study. In some cases, it seemed that they answered questions in a hurry, in order to finish the study and return to their work.
- The number of participants in the study can be seen as low, especially regarding the generalizability of results. Since the study was performed within companies, we counted on each employee's availability to carry out the study, which was not easily achieved. Regardless of this problem, the nature of this study makes it costly to replicate to several subjects.

6 FINAL REMARKS

In an attempt to reduce the number of delivered errors, it is estimated that most companies spend between 50% - 80% of their software development effort on testing [6]. Besides, locating errors can be considered one of the most difficult development tasks [28]. Some techniques can be of great help to reduce the time needed for identifying problems, such as the use of visualizations. The reducing of this location time can have a significant impact on cost and quality of software development and maintenance.

Traditionally, stack traces of the test cases execution are stored in textual format, and their analysis is done after the test case execution. These files contain the most important information about the tests performed, so they cannot be left aside, and it is necessary to have a good understanding of them. ASTRO helps this by providing three different perspectives with interactive visualizations that can support the execution of this understanding task.

The main contribution offered by ASTRO is to support developers in quickly finding the main points of failure and their causes by facilitating the analysis of the textual log files. ASTRO is highly interactive, providing awareness on the status and execution time of each test, showing the relationship between code and test methods and presenting a history of test executions for further analysis.

The study carried out to evaluate the use of ASTRO with software testers and developers from industry provides indications that ASTRO can facilitate the analysis of log file information. Participants also provided important insights and improvement suggestions.

We conducted informal tests with a large amount of data, and it showed that the tool perceptively decreases its performance, due to the sizes of the generated visualizations. While the timeline, the pie chart, and the bar chart visualizations are less affected by this problem, we noticed that the bubble chart and the MOV did not scale well. We want to try other visual formats for the data to compare the effectiveness and scalability of each representation. Another limitation we're working on is that the Callers & Callees perspective does not currently account for methods that call each other or the number of times one method calls another.

Although log files are used mainly for analyzing errors and their causes, they can provide interesting insights in software development. For instance, combining their information with the information provided by other sources (e.g., version control or issue tracking systems), it is possible to assess if tests are being performed according to the plan or schedule, identify which errors are introduced more frequently (providing clues on software development topics that may need more training), reveal team configurations that are more or less error-prone etc. This is subject for future work.

Finally, we intend to integrate ASTRO with Integrated Development Environments (IDEs), so that developers can use the tool while working in their projects, in a faster and more practical way. This would allow users to see the error information through the visualization and be directed towards the error in the source code.

ACKNOWLEDGMENTS

We would like to thank the participants of the study for their useful contributions, as well as Natália Schots and the anonymous reviewers, for the insightful comments for improving this work.

A APPENDIX A - LIST OF STUDY TASKS

Filtering Tasks (FT)

FT1. Please, provide the following information for the following test methods ("testTheory1", "testCoverMe", "aBaseTest" and "skip3"): (a) How long did each test method run? (b) What is the result of each execution? (c) In which class is each test method located?

FT2. Considering the execution of all the tests demonstrated in the tool, how many test methods failed?

FT3. Where did you find the information to answer these questions?

Basic Tasks (BT)

BT1. For the "shouldGenerateCoverageForJUnitParams" and "skip3" test methods, answer the following information: (a) Which classes own these test methods? (b) What was the result of each test execution? (c) Which classes were called when executing each of these test methods? (d) Which methods were executed before and which were executed after each of these test methods? (e) How long did each test method take to run? (f) When (in time) did each of these test methods start and end executing? (g) Is there any test method that did not call any method? If so, which one and why?

BT2. Which test method took longer to run and still returned an error? How long did this test method take to run? Which class own this test case?

BT3. Are there one or more test methods that have been run in the morning (until 11:30 AM) on 14 May 2017? If so, what is their status and which class own it/them.

BT4. Where did you find the information to answer these questions?

Assimilation Tasks (AT)

AT1. For the "Parameters" class, indicate what methods it contains, their test results, and the execution times of each of them.

AT2. Based on the results provided by ASTRO, what could you indicate/infer about the execution of this set of tests? What conclusions can be drawn from this implementation?

AT3. Some methods do not appear in the bubble view or in the bar chart. Which ones? What classes own these methods? What was the possible reason why they did not appear in these views?

AT4. The project manager requested information on the execution of the tests. A co-worker said that the methods "testTheory1", "testTheory2" and "testTheory3" passed, indicating that this information is in the Overview perspective (in the bubble chart and in the bar graph). However, another co-worker said that the tests actually failed, indicating that this information is in the Callers & Callees perspective. Considering these answers, analyze these perspectives and their visualizations, and indicate what may have led to this divergence of interpretations.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007.* IEEE, 89–98.

[2] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the factors that impact the popularity of GitHub repositories. In *32th International Conference on Software Maintenance and Evolution (ICSME, 2016).* Raleigh, North Carolina, USA, 334–344.

[3] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D3: Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (2011), 2301–2309.

[4] Mattias Bybro and S Arnborg. 2003. A mutation testing tool for Java programs. *Master's thesis, Stockholm University, Stockholm, Sweden* (2003).

[5] Diego Cardoso Borda Castro. 2018. *Análise e Compreensão de Arquivos de Log Gerados por Ferramentas de Teste de Software.* Undergraduate final project. Rio de Janeiro State University, Rio de Janeiro, Brazil.

[6] James S Collofello and Scott N Woodfield. 1989. Evaluating the effectiveness of reliability-assurance techniques. *Journal of Systems and Software* 9, 3 (1989), 191–195.

[7] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J. van Wijk. 2008. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software* 81, 12 (2008), 2252 – 2268.

[8] Stephan Diehl. 2007. *Software visualization: visualizing the structure, behaviour, and evolution of software.* Springer Science & Business Media.

[9] George Din, Justyna Zander, and Stephan Pietsch. 2004. Test Execution Logging and Visualisation Techniques. In *17th International Conference Software and Systems Engineering and their Applications, Paris, France.*

[10] githut. 2014. githut.info. http://githut.info/. (2014).

[11] Carlos Gouveia, José Campos, and Rui Abreu. 2013. Using HTML5 visualizations in software fault localization. In *1st IEEE Working Conference on Software Visualization (VISSOFT 2013).* 1–10.

[12] IEEE. 2008. Draft IEEE Standard for Software and System Test Documentation (Revision of IEEE 829-1998). *IEEE Unapproved Draft Std P829/D11, Feb 2008* (2008).

[13] IEEE. 2016. ieee.org. http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages. (July 2016).

[14] ISO/IEC/IEEE. 2013. ISO/IEC/IEEE International Standard for Software and systems engineering – Software testing, Part 1: Concepts and definitions. *ISO/IEC/IEEE 29119-1:2013(E)* (Sept 2013), 1–64. DOI: http://dx.doi.org/10.1109/IEEESTD.2013.6588537

[15] ISO/IEC/IEEE. 2013. ISO/IEC/IEEE International Standard for Software and systems engineering – Software testing, Part 3: Test documentation. *ISO/IEC/IEEE 29119-3:2013(E)* (Sept 2013), 1–138. DOI: http://dx.doi.org/10.1109/IEEESTD.2013.6588540

[16] Riitta Jääskeläinen. 2010. Think-aloud protocol. *Handbook of translation studies* 1 (2010), 371–374.

[17] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *24th International Conference on Software Engineering (ICSE 2002).* Orlando, FL, USA, 467–477.

[18] Tomek Kaczanowski. 2013. *Practical Unit Testing with JUnit and Mockito.* Tomasz Kaczanowski.

[19] Jens Knodel, Dirk Muthig, and Matthias Naab. 2006. Understanding Software Architectures by Visualization–An Experiment with Graphical Elements. In *13th Working Conference on Reverse Engineering (WCRE 2006).* Benevento, Italy, 39–50.

[20] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014).* Hong Kong, China, 643–653.

[21] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing.* John Wiley & Sons.

[22] Kshirasagar Naik and Priyadarshi Tripathy. 2011. *Software testing and quality assurance: theory and practice.* John Wiley & Sons.

[23] Stanislaw Osinski and Dawid Weiss. 2005. A concept-driven algorithm for clustering search results. *IEEE Intelligent Systems* 20, 3 (2005), 48–54.

[24] pitest. 2017. pitest. http://pitest.org/. (2017).

[25] Marcelo Schots and Cláudia Werner. 2015. On Mapping Goals and Visualizations: Towards Identifying and Addressing Information Needs. In *3rd Workshop on Software Visualization, Evolution, and Maintenance (VEM 2015).* 25–32.

[26] Gregory Tassey. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project* 7007, 011 (2002).

[27] Jan Valdman. 2001. Log file analysis. *Tech. Rep. DCSE/TR-2001-04* (2001).

[28] Iris Vessey. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies* 23, 5 (1985), 459–494.