Thuy Nhi Tran-Thien, Hang Bui, Eric Guo, Boya Cao

FootPrint

A variable history viewer

Repository: *https://github.com/cnhguy/FootPrint*

Motivation

The standard IDE debuggers such as those in IntelliJ and Eclipse do not offer the option of stepping backward through the execution. Occasionally, programmers set the breakpoint too far or step through the debugger too fast and miss the step they wanted to examine and have to restart. Having the option to view your variable's history throughout the debugging session would help solve these issues. This is what we are trying to achieve with FootPrint: a user-friendly, lightweight, and simple way for Java developers (specifically, students or beginning developers) to view the history of their variables.

For example, imagine a CS student who is trying to debug their program. They want to look at what is in their array at line 15, but they accidentally advance to line 16 when the array is discarded. With FootPrint, they could simply examine the state of the selected array by looking at the program state from line 15, which FootPrint would store for reference. Moreover, developers could have a better understanding of their code and the data structure by studying their variables' history.

Impact

FootPrint could be useful for Java programmers using IntelliJ. Even if someone doesn't overstep the debugger, viewing variable history would also allow for users to better understand what is going on in their code at a high level perspective. However, for beginner programmers who are more likely to overstep the debugger or who may not have as much intuition as senior programmers and would like to easily view what happened to their variables, FootPrint would be especially useful. Beginner-level programmers are also more likely to prefer using FootPrint over the more heavy-duty alternatives presented in the first section since FootPrint is very simple. Overall, this means less time and frustration spent on debugging. Since we are targeting beginner developers, we will focus on user-friendliness and practical use of our plugin and measure our success based on a user study.

Current Approach

Currently, time traveling debuggers and other plugins (e.g data visualization plugins) exist that allow one to go back and examine previous variable states, but they can be excessive for beginner programmers. Many existing programs store information such as stack frames, exceptions, method history, and logs. Chronon Time Travelling Debugger [1] is one such plugin also saves execution paths and time stamps for the program. Java Tutor is another tool visualizing the variables' states. Many existing plugins also store this information including JIVE [2] and UndoDB [3]. JIVE is a data visualization plugin for Java for Eclipse that also allows the user to see a history of their variables. While these tools may be useful for more sophisticated programs and experienced developers, we believe that intro-level programmers who simply wish they could look at their array from a line they stepped past would find these extra features and extra info unnecessary and perhaps overwhelming. It seems that recording the entire program with something like Chronon just to look at a variable a few steps back could be overkill in such cases.

Related works tend to take the following approaches: recording, using counters, or tracing. Recorders like Chronon, UndoDB, iReplay, etc., record the state of your program throughout the execution, and then allow you to go back and examine the states [4]. These have high overhead since the entire execution is recorded and can require gigabytes of logs per day [5]. For example, according to

Chronon's performance guide, the recorder "is greatly affected by the amount of memory allocated to it". At a minimum, it recommends users to allocate at least 1gb of memory during the recording process and 2-3gb during the unpacking or replaying process [13]. Kendo uses performance counters, which allow for a more low-level insight on program behavior in order to reproduce bugs for multithreaded programs without recording execution. However, Kendo only works for these multithreaded programs since the counters can only be used to reconstruct things such as lock acquisition order [5]. There is also the use of tracing in gdb's debugger and QIRA and strace. There is also the built-in feature of Watches and field watchpoints in IntelliJ. Watches allow you to monitor specific variables or expressions in the current stack frame, but do not store their history which is a large limitation; Watches is intended for evaluating expressions and not tracking variable history [11]. There are also field watchpoints, which allow the user to track a specific instance variable and suspends the debugger each time it changes. While these are useful, they do not actually store the history of the field so you cannot view the previous values of a field. Furthermore, these are only available for fields, not local variables [12].

There are three different types of debuggers: cyclic debuggers, record-replay debuggers, and reverse debuggers. As Engblom pointed out, reverse debugging has been discussed since the very beginning of computer programming. However, it was long ignored because of the difficulty in its implementations, such as the time management and reconstruction approach [8]. Since we are pursuing a similar feature of reverse debugging from the user end but with a simple and light-weighted backend, we came out the idea of extracting the information from debuggers and build a tool to track the footprint of variables [9].

Our Approach

FootPrint integrates with IntelliJ's built in debugger to provide a familiar, yet enhanced debugging experience for IntelliJ's users. To use, the user would set breakpoints and start a debugging session as normal. Then, in addition to the debugging window that pops up, a FootPrint UI would also appear. All local variables and fields in the loaded class will be tracked by default. For each variable that we are monitoring, FootPrint will store all changes that happened to it (from beginning of the tracking period to the end). Consider this scenario:

```
Line 1    int sum = 0    ← set breakpoint
Line 2    for (int i = 0; i < 6; i++) {
Line 3          sum += i;
Line 4    }
Line 5    System.out.println(sum);
```

The user wants to see how the variables sum and i change throughout the for loop. He or she will set a breakpoint beginning at the place where they are interested in the changes. The user then steps through their code and debug as they would normally. During this time, FootPrint records the different values that sum and i were previously assigned to create the following output for their histories:

History:

sum:

line	value
1	0
3	1
3	3
3	6
3	10
15	3

i:

line	value
2	0
2	1
2	2
2	3
2	4
2	5

Notice how sum = 0 only gets recorded once but in the code, sum actually took on the value “0” twice (once during initialization and once during i = 0) but we would only record when the variable changes. This prevents us from storing duplicate information. Once tracked, users can re-access these histories anytime, even if the variable becomes out of scope.

Furthermore, our approach uses less memory because we are only storing information about variables (specifically, values and the line number where it was changed) as opposed to extra data such as stack frames, exceptions, method history, and logs of the entire program like current approaches. Our users are only given information that are relevant to them and are able to get a quick summary of how things change throughout the program.

Architecture & Implementation

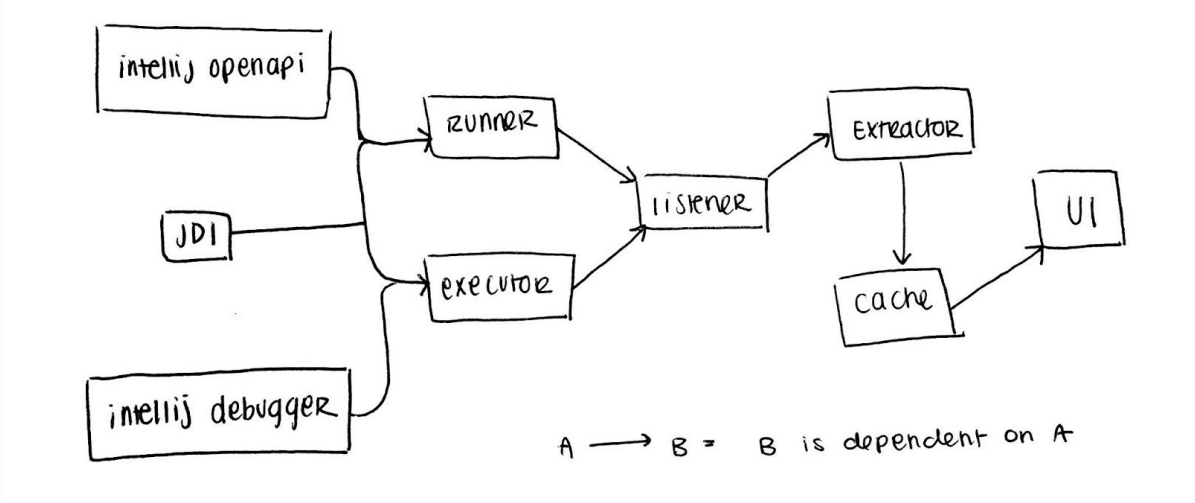


Figure 1.1 Architecture diagram illustrating the major components of FootPrint and their dependencies

IntelliJ's run actions for programs are as follows. First, a user will select a run configuration and an executor. From IntelliJ's documentation [10], a run configuration allows users to run certain types of external processes such as scripts. From IntelliJ's documentation [11], an executor is a specific way of executing any possible run configuration. IntelliJ has three built in executors: Run, Debug, and Run with Coverage. Footprint works with any run configuration, since we have created our own executor, extending the DefaultDebugExecutor. A program runner, which will actually execute the process, is then chosen from all registered program runners by asking whether they can run the given run profile with the given executor ID. Because we want access to the internal state of the running debug process and because we have created a custom executor, a custom program runner was necessary. Finally, ProgramRunner.execute() is called, starting the process. Thus, our custom executor and runner live on top of the existing architecture for IntelliJ's debugger. The interactions between these components are illustrated in Figure 1.2 above.

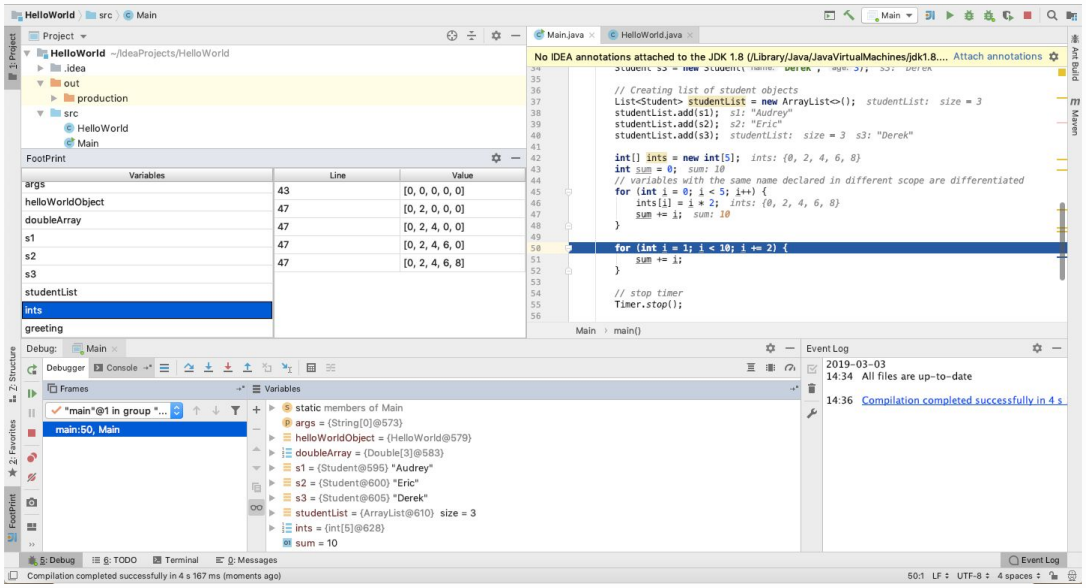
FootPrint uses three resources: IntelliJ's openapi, IntelliJ's debugger interface, and JDI (Java Debug Interface). Our own Runner and Executor classes use these resources in order to run IntelliJ's built-in debugger through our plugin. Our Listener class watches for changes in the program's variables. Our Extractor class extracts the contents of these variables when the Listener indicates a change. This data is stored in our Cache that we implement. The contents of this cache are then displayed through the UI.

The listener class work as follows. The custom runner will, as part of the initialization of the debug process, register with the virtual machine a number of breakpoints, either field or line, that will, when triggered, notify the listener class. The listener class will then notify the extractor class, passing along the necessary information including the stack frame, and will then resume the program. All of this will occur without the knowledge of the user, and will not affect the user's defined breakpoints.

In terms of API's and libraries, FootPrint will use JDI, IntelliJ's XDebugger, and IntelliJ's openapi. XDebugger, the built in underlying debug library used by IntelliJ, and openapi, IntelliJ's api for the editor, are used when implementing the runner and executor. Java Debug Interface (JDI) is used actually extract the variable data.

GUI

Figure 1.2 Text-based GUI



Our UI is text-based. The current state of our UI is shown above in Figure 1.2. All local variables and fields in the loaded class are tracked automatically. The user would essentially select a variable from the “Variables” column and the UI would display its history altogether. This is more useful if the user would like to see the variable history and line numbers all in one place to see how the variable changes. For objects, their histories will be displayed as their toString() definitions for consistency. This takes care of complex data structures such as Map and LinkedList since their toString() is already defined by default in Java. We are currently working on further organizing the variables within the “Variables” column by separating them based on method and object instance.

Since we want FootPrint to be an extension that enhances the user’s experience with IntelliJ’s built in debugger, our UI sits on top of the debugging window that the user can choose to minimize or maximize during their debugging session for easy reference. This placement integrates smoothly with the user’s debugging experience without clogging up the IDEA.

Previously, we were considering a UI that would feature a back button in the debugger window that would allow the user to “undo” the debugger, which would allow them to more easily see interactions between variables. However, we decided to go with our current UI because we found this option too difficult to implement within the constraints of this class. We also felt that our current UI would better accomplish the goal of allowing users to view the overall history of a variable in one place; by offering this feature, we’d also be offering a unique way of viewing variables that other solutions like Chronon do not.

Risks and Rewards

We expect the biggest challenge of this project is detecting when a variable is changed up until the breakpoint set by the user. In order to extract information about a variable, Java debuggers need an "event", such as a breakpoint, to pause the program's execution and get the information from the variable [6]. If we want to track the any changes that happened, we would need to set our own breakpoints at every line before the user's breakpoint in order to monitor those changes. However, doing so would defeat the purpose of creating a fast and lightweight plugin. An alternative is to use watchpoints instead, as a way to track specific variables which sounds much more efficient. We just have to figure out how to incorporate watchpoints into our implementation now. A major limitation is that there are only watchpoints offered for fields. There is also the alternative of just extracting all local variables each time the debugger is suspended, which we have already figured out how to do. However, there is still the issue of only getting data during suspension which we will have to work around. If we cannot get around this issue, then FootPrint will only store values during suspended states. We believe this could still be useful though, since it still can solve the issue of overstepping the debugger so long as you previously had the debugger paused on the line of interest.

Despite this, we think that our solution is feasible since IntelliJ does provide open APIs that allows us to interact with its built in debugger [7] . We will take advantage of this and build our solution upon existing resources. To minimize risks, we will implement basic features first and then add more as time permits.

Cost and Time

Since FootPrint will build on some of the functionalities from IntelliJ's debugger (like extracting variable states), we expect to be able to finish this project within 6 weeks. The first four weeks will be spent on building and testing the backend of extracting and caching information while the remaining two weeks will be spent on building and testing a user interface.

Checks for Success

The midterm to measure FootPrint project was to finish the building process of FootPrint's backend. At this point, we will have fully implemented and tested the storage of debugging information. We are successfully able to extract and store values from objects and primitives, and to display them by invoking their toString() method (when applicable). However, we have not completely met the midterm check since we have yet to figure out how to set additional breakpoints under the hood of FootPrint. To compensate, we have been making headway on the UI. Currently, we are working on making our UI more readable by separating variables while also trying to figure out how to set breakpoints to get a full history of variables.

We have the final exam to check for success will be after the UI design of FootPrint and the launch of the project. We will do experiments with the users for feedbacks to further improve FootPrint, as well as run a performance experiment that focuses on runtime.

Experiments

Pre-processing Experiment:

We first ran FootPrint and Chronon on our sample program in debugging mode. We measured how long each plugin takes to record/prepare for user interaction. For Chronon, this means we measured the recording time. For FootPrint, we set a breakpoint at the end of the program and measured how long it takes for FootPrint to collect and display the variable histories. Since FootPrint set breakpoints under the hood in order to get a full record of variable history (in addition to user breakpoints), measuring this preprocessing time is analogous to Chronon's recording time since both processes have to be complete before the user can begin debugging. We used the results to gauge FootPrint's performance and efficiency based on the program that had the faster pre-processing time. We chose to run FootPrint against Chronon because Chronon seems to be the most popular Java rewind debugger that is compatible with Intellij. We also considered testing JIVE which is also made for Java, but it is focused on data visualization so we thought it was too different to be compared. Since our project is aimed towards students, we will choose example problems or homework solution from an intro CSE class here at UW to most accurately reflect what our target audience will use FootPrint on.

Results:

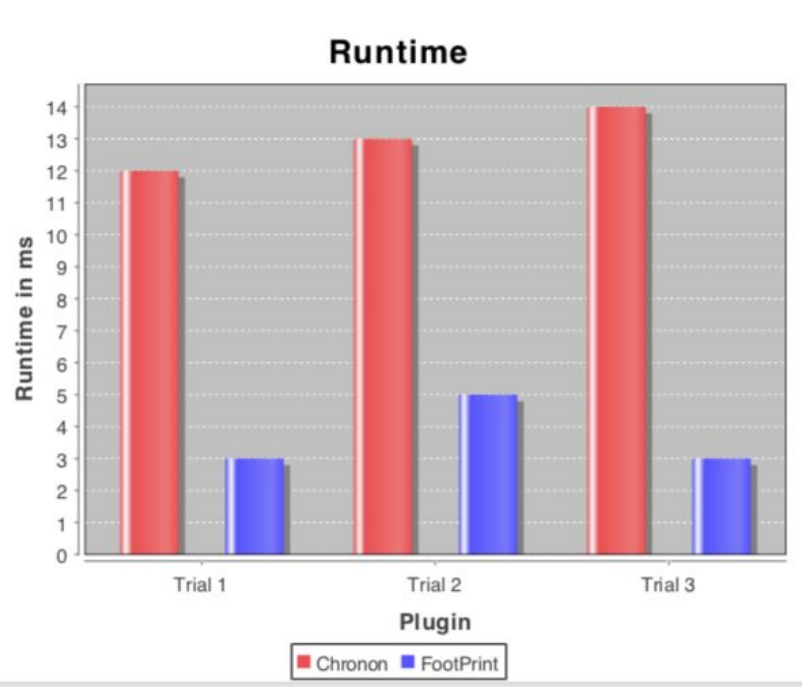


Figure 1.3: Runtime graph comparing preprocessing time for Chronon and FootPrint

The first program we experimented on (results shown in Figure 1.3) was a simple HelloWorld program made by our team. HelloWorld contains common elements that would be used by a student in the intro CSE classes such as for loops, arrays, lists, primitives, and custom objects. We ran FootPrint on its main() method which uses all the above elements. Main() is 70 lines long, of which 38 lines contain code and 32 are comments. Our goal with this program is to show that FootPrint can accurately display various types of data and their changes.

However, as these are just the preliminary results, we plan to run more trials with other programs (not made by us) to reduce bias. The current FootPrint runtimes are inaccurate since FootPrint has not been completed yet. Because we are not currently collecting the full variable history, it's likely that FootPrint's final runtime will be higher. However, the Chronon runtimes are accurate. Furthermore, we realize that only testing on programs that our team has made raises concerns, so in our final experiment we will make sure to test on multiple programs pulled from other sources. We will also select programs of varying sizes to cover more use cases.

Follow this link to find out how to reproduce the tests and figures above:
https://github.com/hangbuiii/FootPrintTest

User Study:

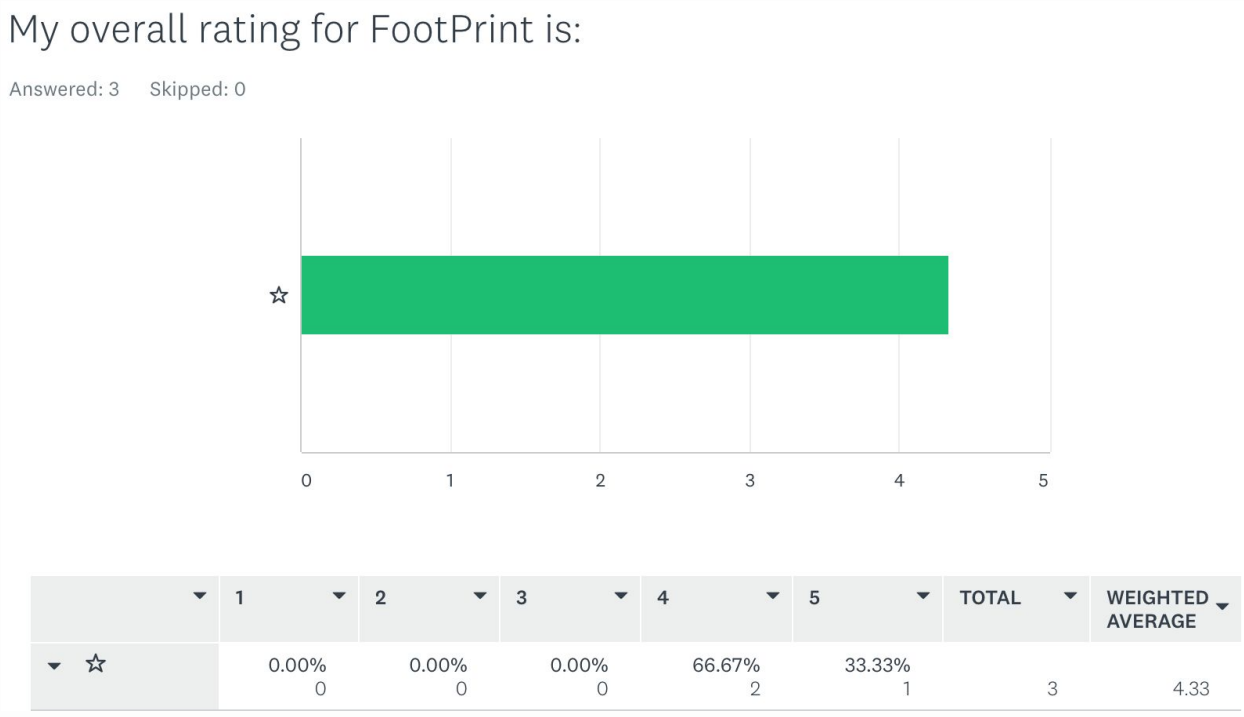
We will gather a small sample of beginner developers (e.g. our friends) who have never been exposed to Chronon and have them test use Footprint and Choronon on the same program (the one from the performance experiment). Afterwards, we will gather data via surveys and measure our goal of being "user friendly" through their feedback to see which one they think offers the most intuitive and understandable UI. Essentially just letting people we know test out FootPrint and seeing if they think our UI is reasonable. Through the results of the feedbacks, we can see if FootPrint achieves its goal of being more user-friendly than current solutions on the market. As suggested by Mortensen, we will create a well-designed user study by reducing bias and ask explorative questions [14]. In order to reduce an bias, we will have users record their answers in an environment where FootPrint team members are not present. We want to ensure that participants do not feel pressured to answer in certain ways due to our influence (e.g. the tone in which we ask questions). We included follow-up sub-questions such as "why?" or "which ones?" to prompt the participants for more info. We made sure to use some Likert questions for better data collection and analysis, as well as still have some open-ended questions so we can collect quotes from participants.

Here is a link to our survey: https://www.surveymonkey.com/r/GSRQTWZ

We also plan to ask TA's and/or instructors their thoughts on FootPrint and whether or not they think it would be useful to their students, but we would like to discuss more with the course staff before proceeding.

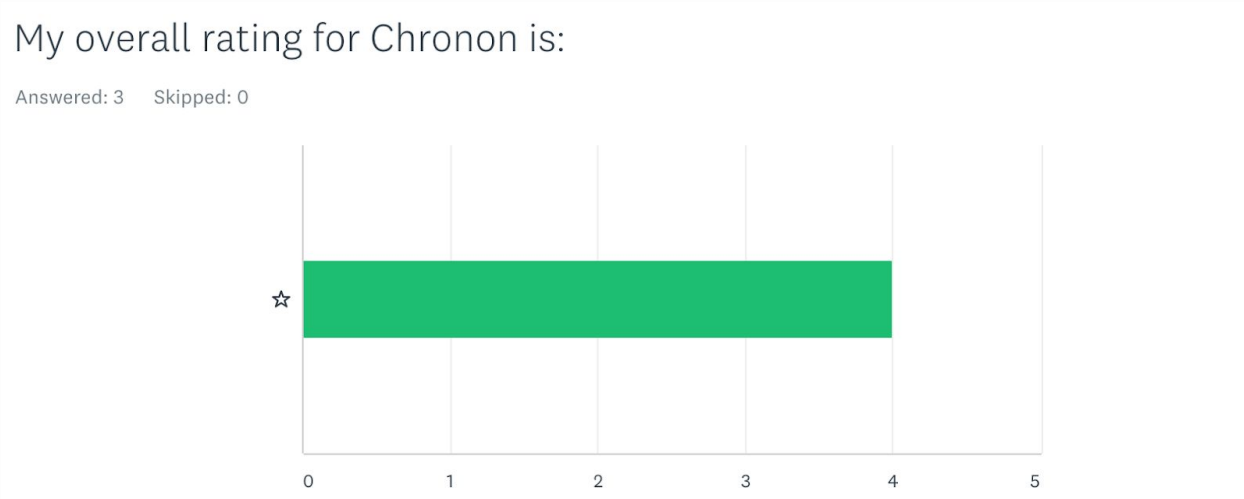
Results:

Figure 1.4: Average rating for FootPrint



	1	2	3	4	5	TOTAL	WEIGHTED AVERAGE
☆	0.00% 0	0.00% 0	0.00% 0	66.67% 2	33.33% 1	3	4.33

Figure 1.5: Average rating for



	1	2	3	4	5	TOTAL	WEIGHTED AVERAGE
☆	0.00% 0	0.00% 0	33.33% 1	33.33% 1	33.33% 1	3	4.00

Currently, we have run a few people (3) through the survey to get some initial results. Please note that FootPrint is not yet completed, so we will properly conduct the survey when it has been finished. The average rating for FootPrint was 4.33/5 as shown in Figure 1.4. The average rating for Chronon was 4.00/5 as shown in Figure 1.5.

Here are some quotes from participants:

"I prefer footprint in a simpler assignment. Chronon is overwhelming and way slower." - Participant 1

"I would use FootPrint because it was simpler to use for a small homework assignment and I could just debug like normal and get the extra feature of seeing previous values that I stepped over. " - Participant 2

It seems that even though FootPrint isn't complete, users still liked the simplicity of it and still found it useful for beginner homework assignments (and preferred it over Chronon for such uses) which is what we intended. However, some participants brought up issues regarding the lack of full history. In this aspect, Chronon is more useful, but we are planning to fix this when we finish FootPrint.

(Note that because the above graphs are included in SurveyMonkey's site, we have not provided instructions on how to generate them since they are automatically created by SurveyMonkey)

Current Status

Currently, we have implemented the basic extraction and storage of data as well as a simple text-based UI. We have unit tests for instantiable classes such as the DebugCache and VariableInfo. At the moment, FootPrint will automatically load/track any fields of the current object, and any local variables within the current method. For objects, their values are displayed through their respective toString() method. Our UI currently displays each value in a single line, which can be a bit hard to read, especially when displaying multiple fields within an object. We are working on refining the UI to make the values more readable. Our biggest hurdle is still finding a way to track all changes, and not just the ones that occur when the debugger is suspended by the user. Right now, we are looking into utilizing IntelliJ's Program Structure Interface (PSI) in order to parse the source file to find assignments, constructors, and mutators so we can set additional breakpoints at these lines. We are also trying to research the implementation of field watchpoints so that we can possibly mimic this with local variables.

Because FootPrint's main purpose of displaying variable's full histories has yet to be achieved, we have put off our full user study and performance experiment until it is complete since we feel that such experiments will be of less use at this point. Instead, the FootPrint team has been testing FootPrint ourselves in order to find bugs, test the UI, etc. The major issue that we encountered was clutter in our variable display. Because FootPrint retains a complete history of all variables encountered so far, the UI can get very busy when stepping into multiple methods and confusing when variables have the same name. Although this can be solved by looking at the line numbers, we still think this leads to the UI being cluttered and would like a clearer way for users to differentiate and organize their variables. We are currently working on this by refactoring our cache to also store object and method names and have the UI use this info in order to group variables. Another issue we found was that the toString() method of an array object was not overridden so it would simply display the memory address. In order to remedy this, we decided to make our own toString() for array references since arrays are quite a commonly used data structure.

Discussion

Throughout this project, we've learned a lot about teamwork. Since our members had previously only worked on projects solo or with one other person, having to cooperate with a total of four members was a valuable learning experience. We had to divide roles, properly communicate, and do code reviews. In the beginning, sometimes we would be confused on whether or not an assignment had been submitted or an email had been sent because we had not clearly communicated, but now, we make sure to notify everyone in our Slack channel when one of us does something.

A major section of this class also involved planning and design, which none of us had done to this extent before. Oftentimes, we are just given a specification in other classes and told to implement it. In this class, we had the opportunity to build something from scratch, design our own specification, choose which technologies to use, how to display our info, etc. We also had the challenge of having to evaluate our project. Having that much freedom was a bit daunting at first. Doing a CS assignment where there wasn't a "correct" answer was also foreign, but we have learned a lot about design, planning, and experiments. For example, we had to refactor more than halfway through the project. Initially, our design just put all the variables displayed in one place. After testing FootPrint ourselves, we realized how cluttered and confusing our UI could get, especially with variables with the same name. This led to us having to refactor many parts of the project, including the cache, extractor, and UI. However, we believe that this was worth it since it makes our UI much more user friendly which is the main appeal of FootPrint. In future projects, we will be sure to more deeply consider the user experience in our designs.

Regarding the technical components of this project, we gained the experience of diving into documentation and online forums in order to understand how JDI and the IntelliJ Debugger api worked. It was quite difficult at first, especially since no one on the course staff had dealt with such topics before, but now when we have issues or questions, we know where to search, what to search, etc. Finally, we also gained experience using version control (Git, in our case). Many of us had never contributed to an open source project before, so we learnt a lot about code reviews, branches, pull requests, CI, and merge conflicts. Overall, we feel that these were all valuable skills for us to learn and to continue working on for our careers as software developers moving forward.

Appendix

Team Assignments

UI: Eric, Derek
Cache & Extracting information: Audrey, Hang

Week-by-week schedule (Underlined items are complete):

Week 5: Architecture and Implementation plan
We will outline the architecture plan for Footprint. We will decide on a UI and give a mockup. Furthermore, since we are extracting info from the built-in debugger, we will research the architecture of that. We will also decide on what data structures to use for our backend.

Week 6: User manual + begin implementation
User manual - ReadMe.MD (Sections: About, How to Download, How to Use)
Implementation - Finish and test module that will extract variable states from the debugger. Finish and test module that will store the variable states.

Week 7: Build and Test
We will complete and do testing of the UI. Further, we will create a user survey to determine usability and stability of FootPrint. At this point, FootPrint's basic features should be usable (basic data extraction and storage).

Week 8: Initial Results
We will do some basic preliminary testing of FootPrint ourselves and fix bugs that we find. We will continue to work on FootPrint's features, particularly, tackling the setting of breakpoints so that we can acquire the full histories of variables.

Week 9: Draft Final Report
We will run a preliminary version of user feedback survey and experiments. Based on these, we will add to the test suite and fix any apparent bugs. We may add more example programs to run FootPrint on, but otherwise coding should be complete and fully tested. - UPDATE: at this point we have the basic implementation of UI and backend complete but code is not complete and fully tested, currently working on this

Week 10: Finalize Project Report
We will fully run the outlined experiments and user study. Finish final commits on Git and finalize the report.

Feedback

Our peer reviewer suggested moving the discussion of our performance experiment into the "Initial Results" section. However, we feel that it is more clear to have it directly after our experimental methodology. Accordingly, we have renamed this section "Experiments" so we include both the methodology and the results and we have renamed our previous "Initial Results" section "Current Status" to better reflect the contents of that section.

References

[1] Chronon Time Travelling Debugger. (n.d.). Retrieved January 27, 2019, from http://chrononsystems.com/products/chronon-time-travelling-debugger

[2] About JIVE. (n.d.). Retrieved, January 27, 2019, from https://cse.buffalo.edu/jive/

[3] UndoDB. (2019). Retrieved January 27, 2019, from https://undo.io/products/undodb/

[4] Mozilla. (n.d.). Mozilla/rr. Retrieved February 4, 2019, from https://github.com/mozilla/rr/wiki/Related-work

[5] Olszewski, Marek & Ansel, Jason & Amarasinghe, Saman. (2009). Kendo: Efficient Deterministic Multithreading in Software. International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS. 44. 97-108. 10.1145/1508244.1508256.

[6] Package com.sun.jdi.event. (2018, October 06). Retrieved from https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/package-summary.html

[7] JetBrains. (2019, February 04). JetBrains/intellij-community. Retrieved from https://github.com/JetBrains/intellij-community/tree/master/platform/editor-ui-api/src/com/intellij

[8] Engblom, J. (2012, September). A review of reverse debugging. In System, Software, SoC and Silicon Debug Conference (S4D), 2012 (pp. 1-6). IEEE.

[9] Run Configurations. (2018, September 14). Retrieved February 4, 2019, from https://www.jetbrains.org/intellij/sdk/docs/basics/run_configurations.html

[10] Execution. (2017, October 30). Retrieved February 4, 2019, from https://www.jetbrains.org/intellij/sdk/docs/basics/run_configurations/run_configuration_execution.html

[11] Adding, Editing, and Removing Watches. (2017, October 30). Retrieved February 18, 2019, from https://www.jetbrains.org/intellij/sdk/docs/basics/run_configurations/run_configuration_execution.html

[12] Breakpoints. (2019, February 01). Retrieved February 18, 2019, from https://www.jetbrains.com/help/idea/using-breakpoints.html#field_watchpoint

[13] Prashant, D. (2012, December 1). Chronon Performance guide. Retrieved February 19, 2019, from https://chronon.atlassian.net/wiki/spaces/DOC/pages/525186/Chronon Performance guide

[14] Mortensen, D. Best Practices for Qualitative User Research. (2019, February 10). Retrieved February 24, 2019, from https://www.interaction-design.org/literature/article/best-practices-for-qualitative-user-research

Hours Spent: 2