FootPrint

A variable history viewer

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

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

Motivation

Standard IDE debuggers such as those in IntelliJ and Eclipse do not offer the option of stepping backward throughout the debugging process. 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.

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. 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 2-3 gb 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].

Three prominent types of debuggers are: 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. 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:

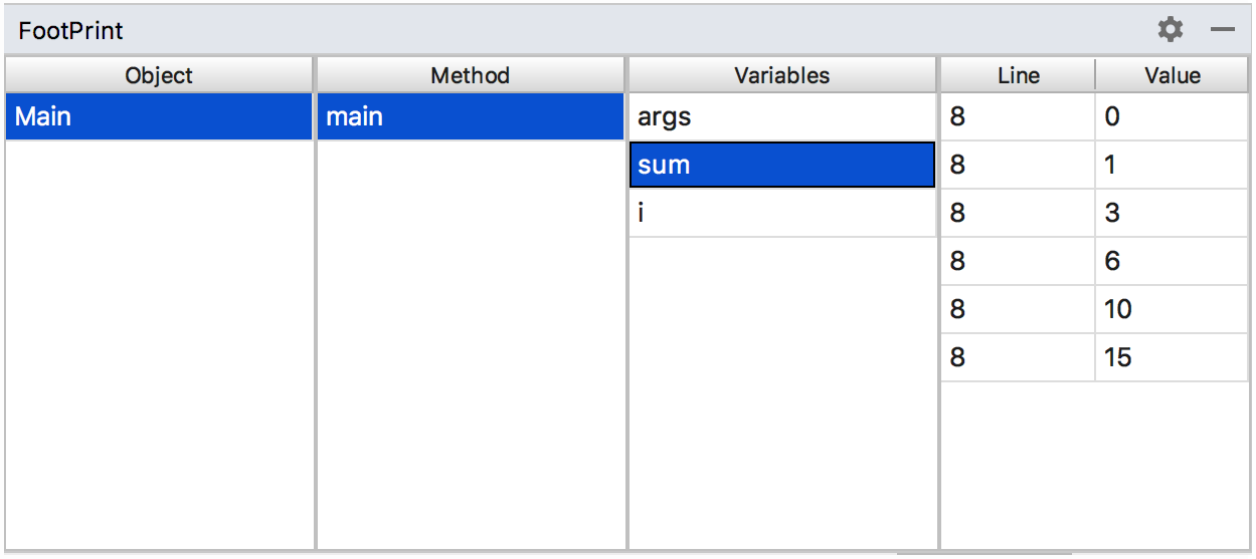
Figure 1.1: Example scenario

```
public class Main {

    public static void main(String[] args) {
        int sum = 0;
        for(int i = 0; i < 6; i++) {
            sum += i;
        }
        System.out.println(sum);
    }
}
```

The user wants to see how the variable “sum” changes throughout the for loop. The user then can debug as they would normally would (they can either set a breakpoint somewhere after line 12 or step through the loop). During this time, FootPrint records the different values that “sum” were previously assigned to create the following output:

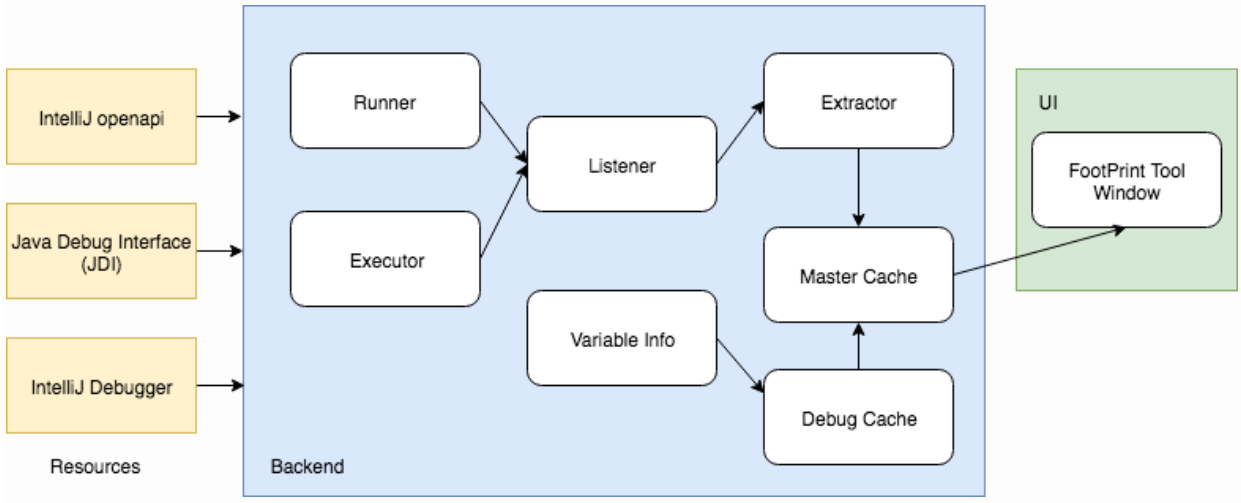
Figure 1.2: History output for “sum”



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 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 can get a quick summary of how things change throughout the program.

Figure 1.3 Architecture diagram illustrating the major components of FootPrint and their dependencies



Architecture & Implementation

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 through our custom Runner and Executor classes. Thus, our backend lives on top of the existing architecture for IntelliJ's debugger. The interactions between these components are illustrated in Figure 1.3 above in the Backend section.

Next, our Listener class works as follows. The custom runner will, as part of the initialization of the debug process, register with a virtual machine a number of breakpoints 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's backend will use JDI (Java Debug Interface), IntelliJ's Debugger, and IntelliJ's openapi. Debugger, 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. JDI is used to extract the variable data on the stackframe.

To summarize, FootPrint uses three resources: IntelliJ's openapi, IntelliJ's debugger interface, and JDI (Java Debug Interface), which are illustrated by the yellow boxes in Figure 1.3 above. 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 breakpoint events during the debugging process. Our Extractor class extracts contents such as local variables using JDI from the stackframe when the Listener indicates that the program is suspended. This data is stored in our caches that we implement. The Master Cache maps object instances and methods to a separate Debug Cache that stores the variable's histories in the

form of lists of Variable Info objects. There is a Debug Cache for each method. The UI then can access these histories through the Master Cache and display them to the user.

GUI

An overview of our UI is shown in figure 1.4. Since we want FootPrint to be an extension that enhances the user's experience with IntelliJ's built in debugger, our UI sits next to 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.

All local variables and fields in the loaded class are tracked automatically. In figure 1.5, our UI organizes them by object instances and methods that they were initialized in. The "Object" column displays both static and object instances. Instances of the same class are differentiated by a unique ID while static instances do not have one. Within each instances, we also track what methods were called in the "Method" column. From the selected method, users can view the history of all the local variables within it by selecting a variable from the "Variables" column and the UI would display its history altogether in the "History" column. 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.

Figure 1.4 Text-based GUI Overview

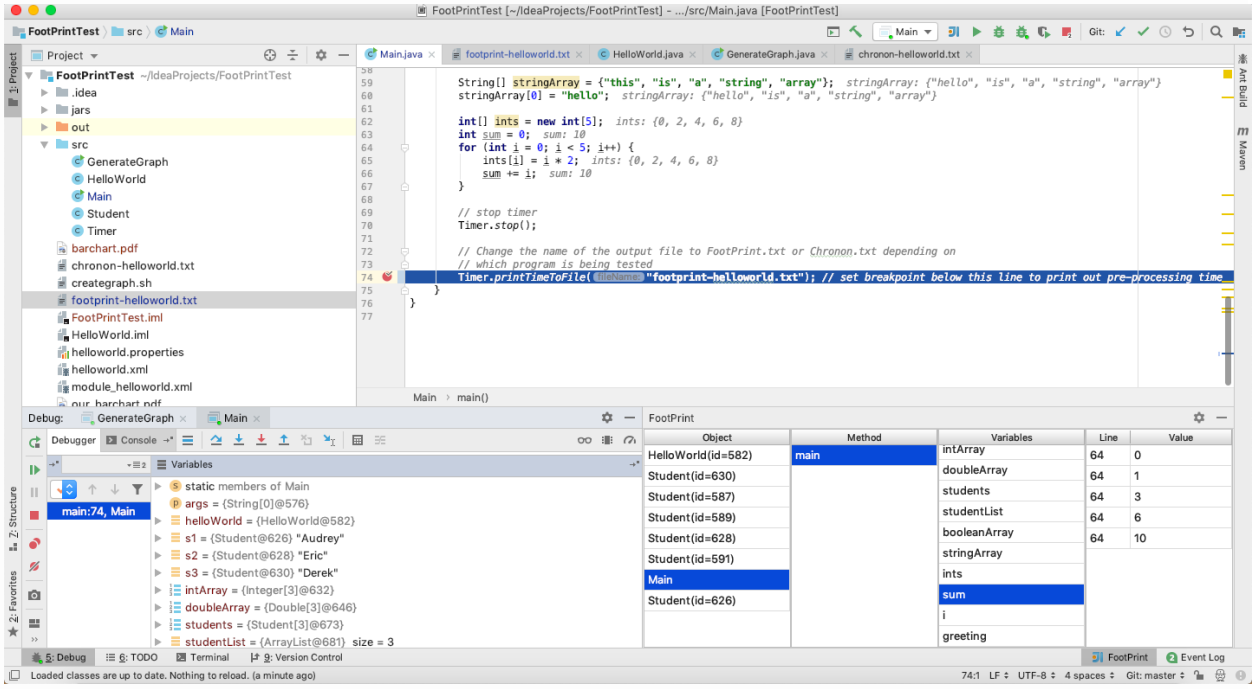
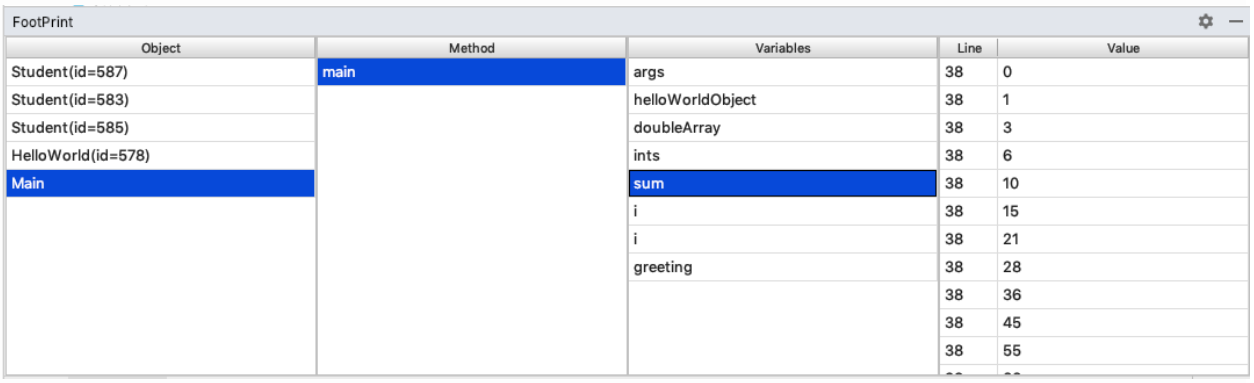


Figure 1.5 A more detailed look. Variables are separated by object instances and methods that they were initialized in. Note that static instances are not marked by an id number.



Experiments

Pre-processing Experiment:

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. Although these processes are different, we feel that measuring this pre-processing time best simulates the user's experience using both plugins, since they will have to wait for recording/collection to complete before they are able to start debugging. Note this represents the "worst case" for FootPrint since we are setting breakpoints at the very end of programs. We gauged FootPrint's performance and efficiency by comparing the pre-processing times of each plugin. 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.

Figure 1.6: List of test programs

Program	Lines of executable code
HelloWorld (created by FootPrint team)	159
Prime Ex (open source [15])	103
Assassin Game (from CSE 143)	435
Anagram Solver (from CSE 143)	166
Simpledb (from CSE 444)	5294

Since our project is aimed towards students, we have chosen example problems and homework solutions from intro CSE classes here at the University of Washington or relatively simple open source programs to most accurately reflect what our target audience will use FootPrint on. A larger program was also tested on for reference on how FootPrint would perform on a larger scale. A list of these programs and their respective lines of executable code can be found above in Figure 1.6.

Results:

As shown in Figure 1.7, FootPrint's runtimes are consistently much higher than Chronon's. Although we do store less information than Chronon, the slower runtime is likely attributed to the additional breakpoints at every line. Despite this, we still think FootPrint is useful because it is aimed towards students who are likely to not be debugging very complex/lengthy programs where the pre-processing time would be more noticable. Furthermore, the main appeal of FootPrint is its user-friendliness as well as how it "records as you go" as opposed to Chronon which requires users to record the entire program before they begin debugging. In the average case, the user will likely not be setting breakpoints at the very end of their program very often so the pre-processing times illustrated above are higher than what users would usually see if they are using FootPrint as they go through their programs. However, we realize that there is room for improvement in FootPrint's performance and would like FootPrint to eventually be useful for more experienced programmers debugging large programs as well. FootPrint's lack of scalability is especially visible in Figure 1.8, where we tested it on a ~5,000 line program. We will look into reducing the pre-processing time in the future (explained further in the "Future Steps" section below).

Figure 1.7: Runtime graph comparing the average preprocessing time of 5 trials for Chronon and FootPrint on various beginner-level programs

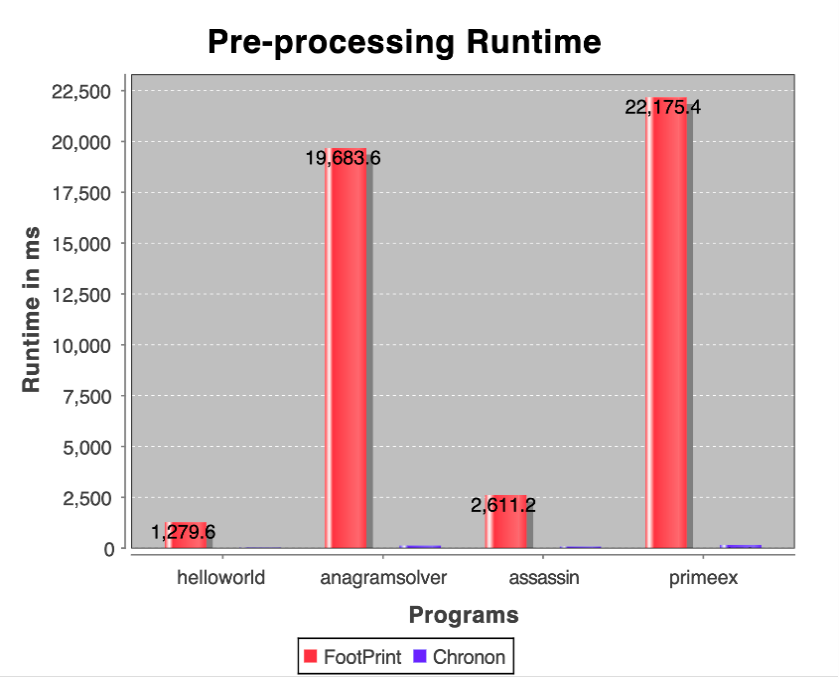
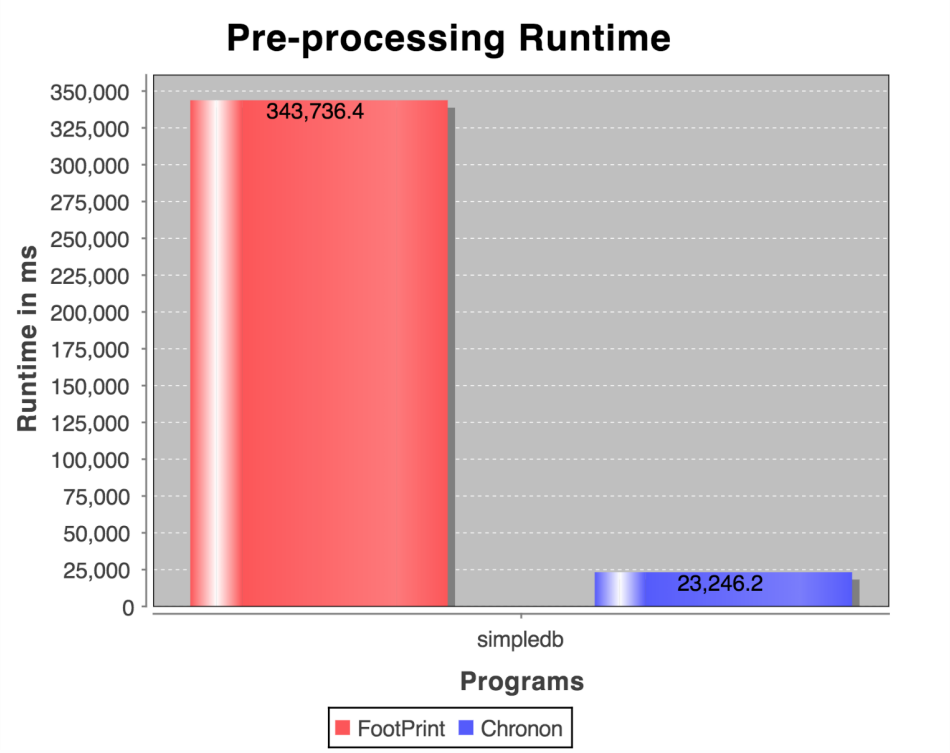


Figure 1.8: Runtime graph comparing average preprocessing time of 5 trials for Chronon and FootPrint on a larger program



The following repo includes instructions on how to reproduce our experimental results above: https://github.com/hangbuiii/FootPrintTest

User Study:

We gathered a small sample of beginner developers (e.g. our friends) who have never been exposed to Chronon and have them test use Footprint and Chronon on the same program (the one from the performance experiment). Afterwards, we gathered 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. We aimed to create a well-designed user study by reducing bias and ask explorative questions [14]. In order to reduce any bias, we have users recorded their answers in an online survey where FootPrint team members were 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.

Our survey can be found here: https://www.surveymonkey.com/r/GSRQTWZ

Figure 1.9: Future use of FootPrint

Figure 1.10: Average rating for FootPrint

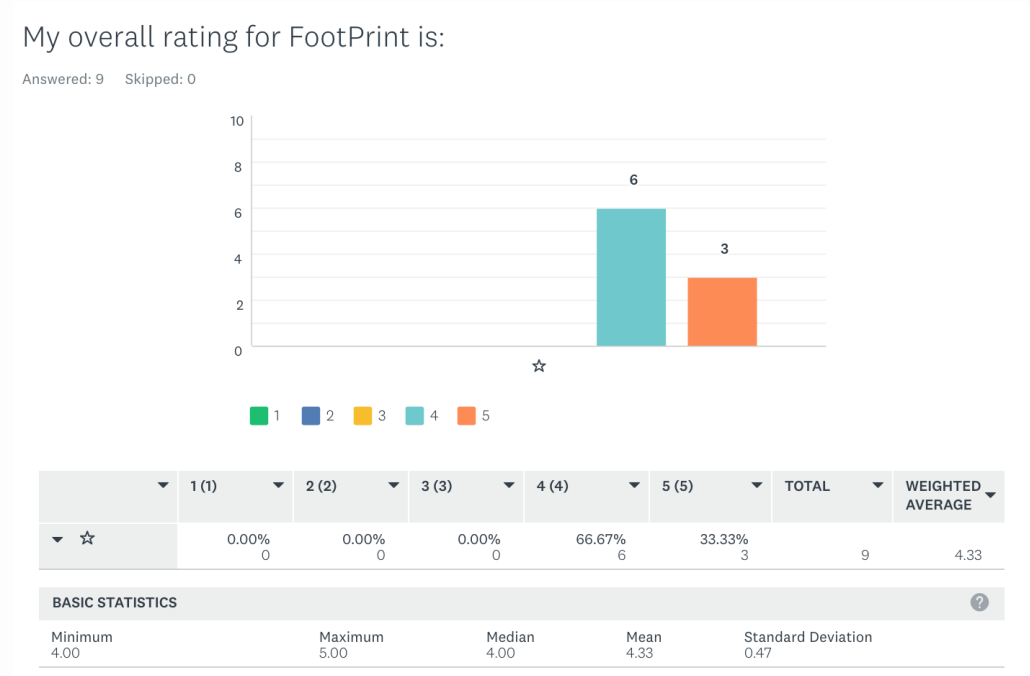
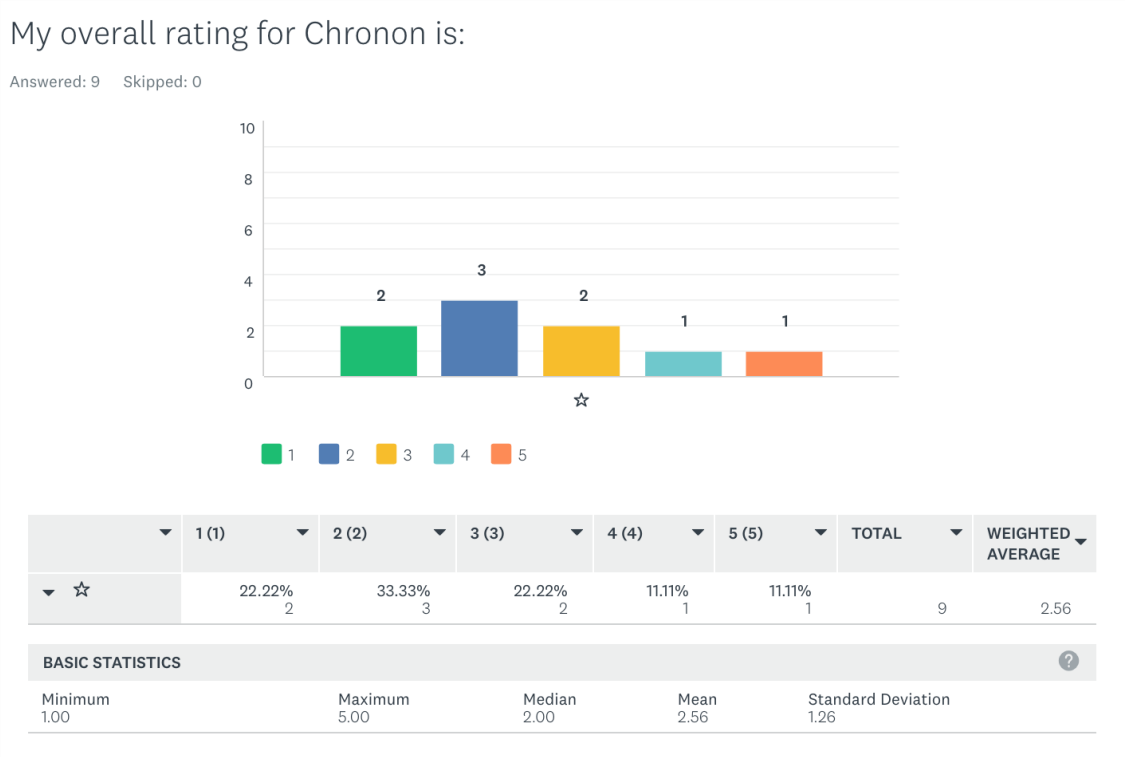


Figure 1.11: Average rating for Chronon



Results:

Our full result data can be found here: https://www.surveymonkey.com/stories/SM-BXYPHN9L/

We had 9 people take our survey. All participants were student developers who were either in 14X classes (introductory Java programming) at the University of Washington or had recently finished the 14X series (within the last year).

As shown above in Figure 1.9, almost 90% of users would use FootPrint again, with the remaining 10% being neutral. This neutral participant took the survey with an earlier version of FootPrint and their main issue was that variable values were only collected when the debugger was stopped. Since then, we have rectified this issue so that full variable histories are collected and displayed. Some of these early testers (3 participants) also complained about the UI being cluttered. To solve this, we now separate variables by object/class and method. Since these fixes, the more recent testers tended to give FootPrint higher ratings.

The main complaints for Chronon were that its debugger interface is inconsistent with what the users are used to. They had to learn Chronon's debugger interface whereas FootPrint works directly with IntelliJ's debugger. Features that usually accompany IntelliJ's debugger are missing from Chronon's. For example, many users mentioned the absence of the in-line display of variable values (e.g. IntelliJ will display the contents of your list alongside its declaration when in debugging mode). Another complaint is Chronon failed to display any information on several occasions for reasons that are unknown.

The average rating for FootPrint was 4.33/5, with ratings ranging from 4-5 stars (Figure 1.10). Compared with Chronon's average rating of 2.56/5 (Figure 1.11), with ratings from 1 - 5 stars, FootPrint performed significantly better in our user study. We calculated a 2-tailed t-test with .05 significance and

our results showed that the difference in rating between FootPrint and Chronon was statistically significant, with a p-value of 0.001765.

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

"Chronon was really confusing to use. I liked how FootPrint was just like the regular debugger. I could use it without even looking at the manual." - Participant 3

"it was annoying having to set the run configurations for Chronon it was much easier to use Footprint" - Participant 4

Users liked the simplicity of FootPrint and found it useful for homework assignments (and preferred it over Chronon for such uses) which is what we intended. The participants especially liked how FootPrint could be used with the regular IntelliJ debugger, whereas Chronon could not. Overall, we conclude that we have accomplished our goal of creating a variable history viewer that is simple and easy to use.

Future Steps

Although the basic features of FootPrint are working, there is definitely room for improvement. We focused on making FootPrint for students/beginners, so when run on larger, more complex programs, FootPrint's performance can be lacking when compared to current solutions like Chronon (as demonstrated in our performance experiment above). This is mostly due to the fact that breakpoints are set at every line in our current implementation. In the future, we would like to look into code analysis so that we can only set breakpoints at lines where values could possibly be changed. Furthermore, FootPrint does not currently support multi-threaded programs. We also believe the UI could be improved. Ideally, FootPrint would allow users to select what variables they would like monitored, thus reducing the need to set breakpoints everywhere and ultimately improve the backend performance.

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, designing the UI was rather challenging. 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. We also had to refactor more than halfway through the project. Initially, our UI 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.

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
[15] Tcondit. (2015, November 16). Tcondit/PrimeEx. Retrieved from https://github.com/tcondit/PrimeEx

Hours Spent: 7