DeMark Project

Team DeFacto-UW:
Andrew Tran, Tony Vo, Tuan Ma, Jeff Xu, Lemei Zhang

June 7, 2018

1 Introduction

Developers want to understand what their code is doing, and usually, the simplest way to do this is to write extra, temporary code such as print statements or if statements to see the values of the variables and the overall flow of the program. This is also known as "printf" debugging. These lines of code are usually there for developers and are not meant for production. However, their removal is often missed and end up in production code, only to be removed at a later date in the future. Currently on GitHub there are over 573,000 commits with the message "removed unnecessary prints" [2].

Our solution to this problem is to provide developers with a tool to mark these temporary lines of code while they are developing and later delete them before committing. The primary goal for this is to increase programming productivity by facilitating the management of temporary code, and minimizing the time spent for searching and removing unintended code. This tool will be used as a plugin for IntelliJ IDEA because having this feature as part of an IDE enables programmers to mark lines as they code, rather than sift those lines out when they commit.

Currently, the tool can be found and installed in IntelliJ IDEA through the JetBrains Plugin Repository [3]. The implementation of the tool is open-source and can be found in the DeFacto-UW's GitHub repository [1].

2 Existing Solutions and Their Limitations

Currently, there are several ways of to do logging or transient logging. One way is by using logging levels and loggers [7]. An advantage of using logging levels is that the developer can have multiple levels of logging that can give information about program states. It also provides a way to log all the information that the developer may deem important. However, it may clutter the code, and it takes time for one to learn how to use it.

Another method is through the addition of the temporary, non-production lines of code (e.g print statements), where the developer will manually delete those lines later. This way is simple, quicker to do on the spot, and often requires no extra tools or learning to perform. It also gives the developer more control over what is deleted. However, this method is time consuming, especially when it is applied to a large code base. Along the same line as going through the code and manually deleting the lines, Git version control also has a patching system that allows the user to select which lines of code to be committed at the time of commit. This prompts an interactive mode for Git that allows developers to go through hunks of code and choose which lines to commit. This method, however, is only helpful in handling temporary code that is added to the code base when the developer wants to commit his or her changes. When debugging a program, one might want to let

only the temporary code for some specific methods to execute, and Git's patching system will not provide any help on this case because the developer will still need to go through the methods to comment or uncomment the temporary code manually.

Both manual deletion and Git patching require the developer to do a small mental reconstruction of what their code actually does to decide whether or not to add or remove specific lines of code, especially if the difference between a temporary code line and an actual line meant for production is not clear. This could become problematic as the time between commits become bigger which makes the recall time longer for each hunk of code. Furthermore, if the developer had already committed a previous version with temporary code, they won't show up as changes in the Git's interactive line selection. Because DeMark allows the developer to mark lines as they are typing or immediately after they are typing, it takes away the need to do any mental reconstruction of the code later on in development. This will prevent the oversight of lines that needed to be deleted, as well as reducing the recall time in general.

In terms of plugins, there is currently an IntelliJ plugin called LineOps that deal with marking lines and manipulating them. It allows the user to place a bookmark, which is a feature provided by IntelliJ that allows annotation with a given description, on lines that matches a certain string and perform actions like cut, copy, or delete on those bookmarked lines [5]. However, LineOps only allows the user to clear bookmarked lines, or clear the bookmarks completely. DeMark will also provide a toggle feature that allows the developer to comment out or uncomment the marked lines. The toggle feature will be separate from the clear feature, where the latter feature will allows the developer to actually remove the marked lines from a file. DeMark also allows the developer to undo their clear actions and restore their previously cleared marked lines. This can be useful in the case where the developer wants to remove their marked lines to commit and then bring the cleared lines back after the commit to continue working.

3 Description

DeMark current has mark, unmark, toggle, clear, unclear, and display as features. Mark allows the user to add a highlight and bookmark specific lines of code. Unmark removes any highlights and bookmarks on lines that were previously marked. Lines that are marked using the tool are kept track of as "marked lines". Toggle will comment out or un-comment marked lines, essentially turn them "off" while not removing them from the code. Clear deletes all marked lines from the code and users are allowed to perform multiple clears in a row. Unclear restores any lines that were removed by clear, starting at the most recent to the oldest. Finally, display opens a window that shows the user all the currently marked lines and their line numbers in the current file.

3.1 Graphical User Interface

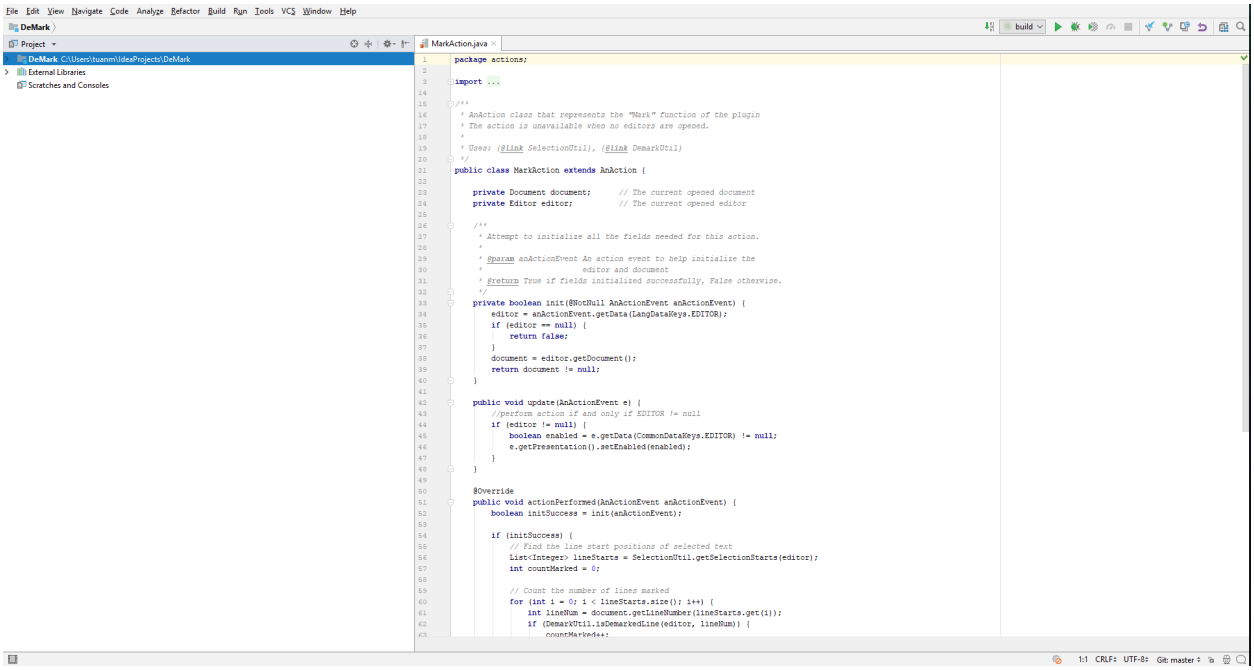

Figure 1: Original Interface

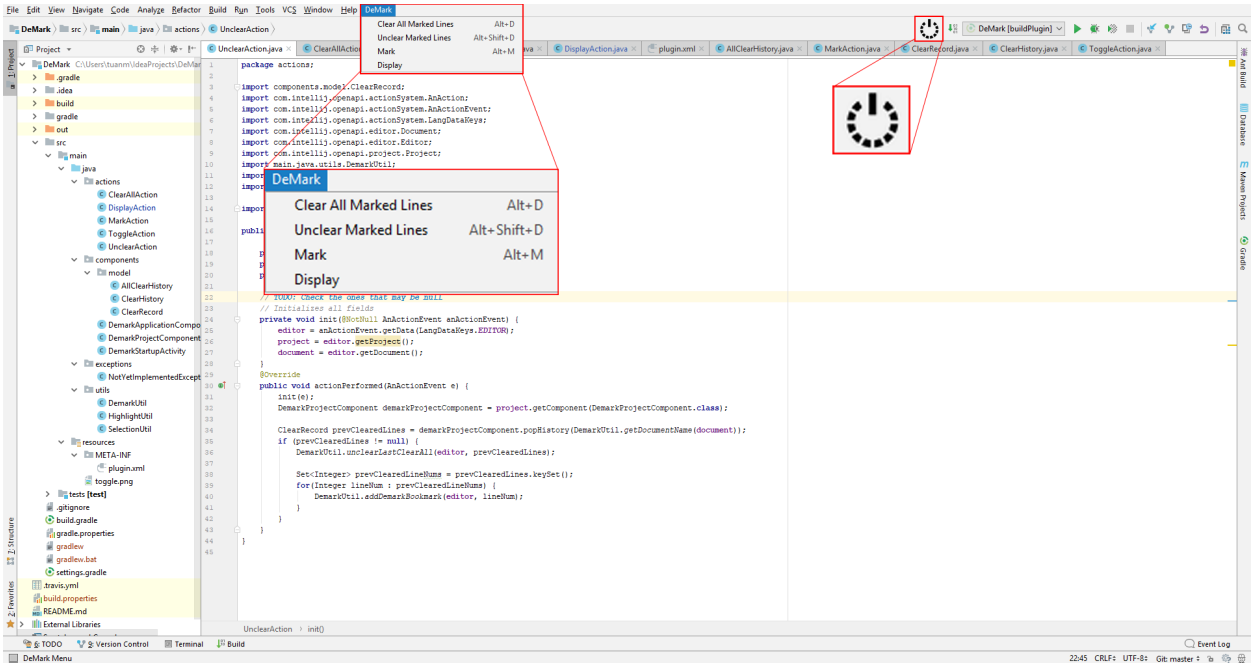

Figure 2: Plugin Installed

Figure 1 portraits the original IntelliJ display without our tool installed. With the tool installed, the developer is able to see all available options to them illustrated in figure 2 in the drop down menu.

The power button in figure 2 is the toggle button. It allows the developer the turn on and off the marked lines. When the toggle button is clicked, all the lines that were previously marked with a DeMark bookmark will be commented out, and another click on the button will uncomment those marked lines. The developer is allowed to mark and unmark new lines in both cases, as well us using the clear and unclear functions.

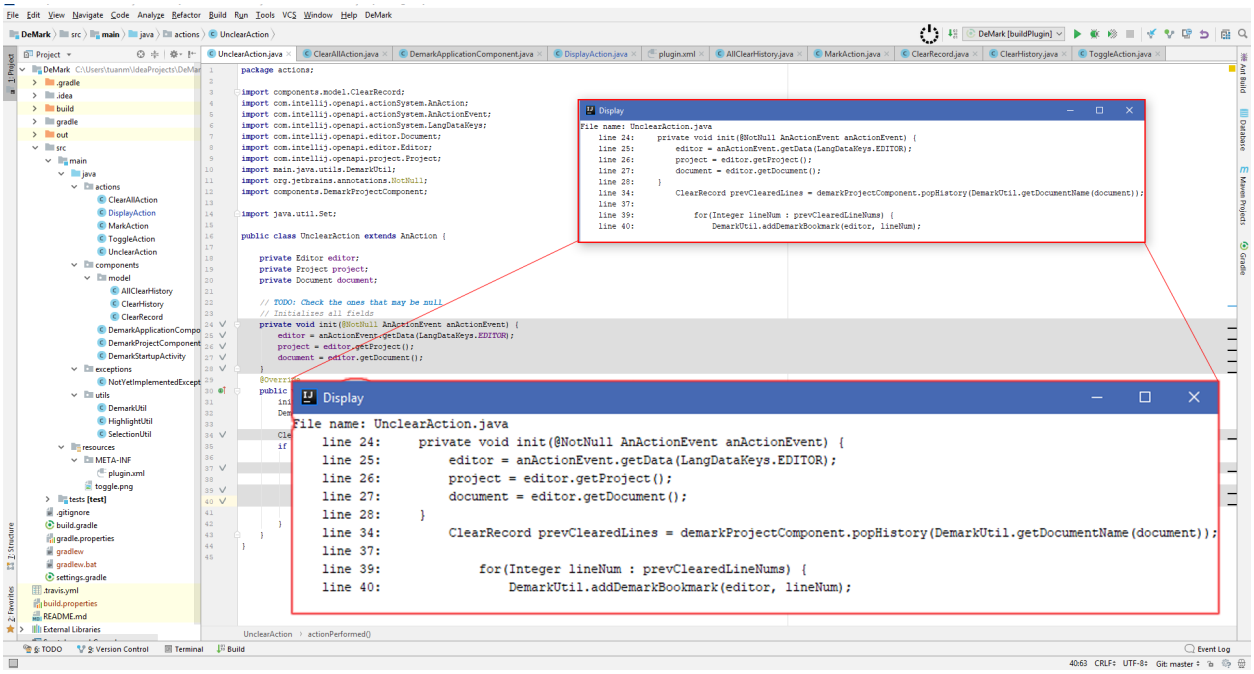

Figure 3: Display Function

The sections outlined in red represent the way DeMark marks lines. The developer is able to use a keyboard shortcut to mark the line and also be able to click on the line number to mark it as well. The marked lines will only be highlighted and marked within the IntelliJ IDEA. This means that lines that are marked in IntelliJ with our tool will still show up in other text editors and IDEs, but not marked or highlighted.

The "display" function creates tabs that are in display mode. Figure 3 shows what it looks like currently. Display mode allows the developer to see their marked lines in their immediate context. The goal of this mode is to allow the developer to see only their marked lines in context, allowing for a quick account of all the things that they have marked or unmarked, or a quick review of their code that they suspect is having issues. In the future, this mode will also allow the developer to be able to see an overview of every file that has marked lines in separate tabs, providing a good overall view of their temporary code across their project. The developer will not be able to make any changes to any lines while in display mode.

Visually, a marked line of code is highlighted in gray, paired with a check mark near the line number of the code. DeMark is designed to go beyond the simple marking of code. Internally, the tool keeps track of the marked lines of code via IntelliJ bookmarks with the description "DeMark". This allows us to implement functionality that allows the user to easily manipulate the marked lines of code, such as being able to comment and uncomment all the lines that are previously marked with a single action, being able to clear all the marked lines at once, as well as being able to restore all the previously cleared marked lines.

Figure 5 illustrates what the interface looks like after the developer deletes the previously marked

```java
public final class Dft {

    /*
     * Computes the discrete Fourier transform (DFT) of the given complex vec
     * All the array arguments must be non-null and have the same length.
     */
    public static void computeDft(double[] inreal, double[] inimag, double[]
        int n = inreal.length;
        for (int k = 0; k < n; k++) {  // For each output element
            double sumreal = 0;
            double sumimag = 0;
            for (int t = 0; t < n; t++) {  // For each input element
                double angle = 2 * Math.PI * t * k / n;
                System.out.println(angle);
                sumreal +=  inreal[t] * Math.cos(angle) + inimag[t] * Math.si
                sumimag += -inreal[t] * Math.sin(angle) + inimag[t] * Math.co
                System.out.println(sumimag);
                System.out.println(sumreal);
            }
            outreal[k] = sumreal;
            outimag[k] = sumimag;
        }
    }

}
```

Figure 4: Before clearing marked lines

lines of code. This is represented as a menu option labeled "Clear all marked lines" integrated into the IDE and can also be invoked using a keyboard shortcut. Marking and deleting lines of code together are some of the core features of DeMark. The tool also provides a way to restore all lines and marks that were previously deleted using DeMark.

```java
public final class Dft {

    /*
     * Computes the discrete Fourier transform (DFT) of the given complex vec
     * All the array arguments must be non-null and have the same length.
     */
    public static void computeDft(double[] inreal, double[] inimag, double[]
        int n = inreal.length;
        for (int k = 0; k < n; k++) {  // For each output element
            double sumreal = 0;
            double sumimag = 0;
            for (int t = 0; t < n; t++) {  // For each input element
                double angle = 2 * Math.PI * t * k / n;
                sumreal +=  inreal[t] * Math.cos(angle) + inimag[t] * Math.si
                sumimag += -inreal[t] * Math.sin(angle) + inimag[t] * Math.co
            }
            outreal[k] = sumreal;
            outimag[k] = sumimag;
        }
    }

}
```

Figure 5: After clearing marked lines

All of the marking, clearing, and unclearing of temporary code lines also have keyboard shortcuts associated with them.

3.2 Technologies

The tool we are developing is a plugin that adds new interactive components and new functionality to the IDE. We are using the IntelliJ Platform SDK to develop for IntelliJ. The platform provides the infrastructure to build IDEs that are composable. This means that the platform is "responsible for the creation of components, and the injection of dependencies into classes" [6]. This allows us to easily develop the tool as a plugin for IntelliJ using this platform.

Additionally, we are using gradle-intellij-plugin as the work flow to build the tool. It takes care of the dependencies of our tool, including the base IDE as well as other plugins that our tool may depend on [4]. We are also using Java 8 for our development language since that is the language of the IntelliJ Platform.

3.3 Architecture

Each of our components will be extending and using the API provided by the Intellij SDK. The functions that our tool provides (mark, unmark, clear, unclear, etc.) are represented in IntelliJ as action classes that extends the IntelliJ `AnAction` class. To the developer these actions are represented as visual elements through a drop down menu or keyboard shortcuts. Then, when the developer clicks on "Mark" or enter the correct keyboard shortcut, the API will link the developer's input with the appropriate action class, which in this case will the Mark Action. The IntelliJ SDK also provides us a way to collect information from the IDEA itself, this includes the text editor, the documents, and projects, all of which are used to keep track of text, highlights, and bookmarks.

To maintain a cohesive and modular structure for our project, we decided to implement utility classes that work more directly with the components of the SDK mentioned above and are more specialized in specific functions. We made sure that the action classes themselves only interact with the SDK through our utilities.

More specifically, the marked line utility will handle most of the work on marking lines and keeping track of whether or not certain lines or code blocks are marked. This utility works closely with the SDK's `BookmarkManager` and the `MarkupModel`. The marked line utility allows a 1:1 correspondence between the IntelliJ's `Bookmark`s and `RangeHighlighter`s which makes for a cleaner implementation of a "marked line". The selection utility handles caret and selection positions, this includes getting line numbers and offsets. Selection will work closely with the SDK's `Document` for line information, and the `SelectionModel` as well. The text utility handles the writing and reading of actual line information, this includes writing to a line, finding certain words, or deleting lines and words. Because it handles writing to the IDEA itself, it uses IntelliJ's `Runnable` and `WriteCommandAction`.

With that, a use case is when a developer marks a line, IntelliJ gets it as `AnActionEvent` and passes it into the mark action. The mark action then calls the marked line utility, which gets the line information from the selection utility to determine whether a line is already marked. The marked line utility then passes said information to the action to decide whether to unmark a line or mark it by highlighting it and adding a bookmark with a "DeMark" description. Similarly, another use case is when the developer clears a line, an action event is passed into the clear action, which then gets information on if the lines are marked from the marked line utility, if the lines are selected from the selection utility, and then finally uses the text utility to delete the lines.

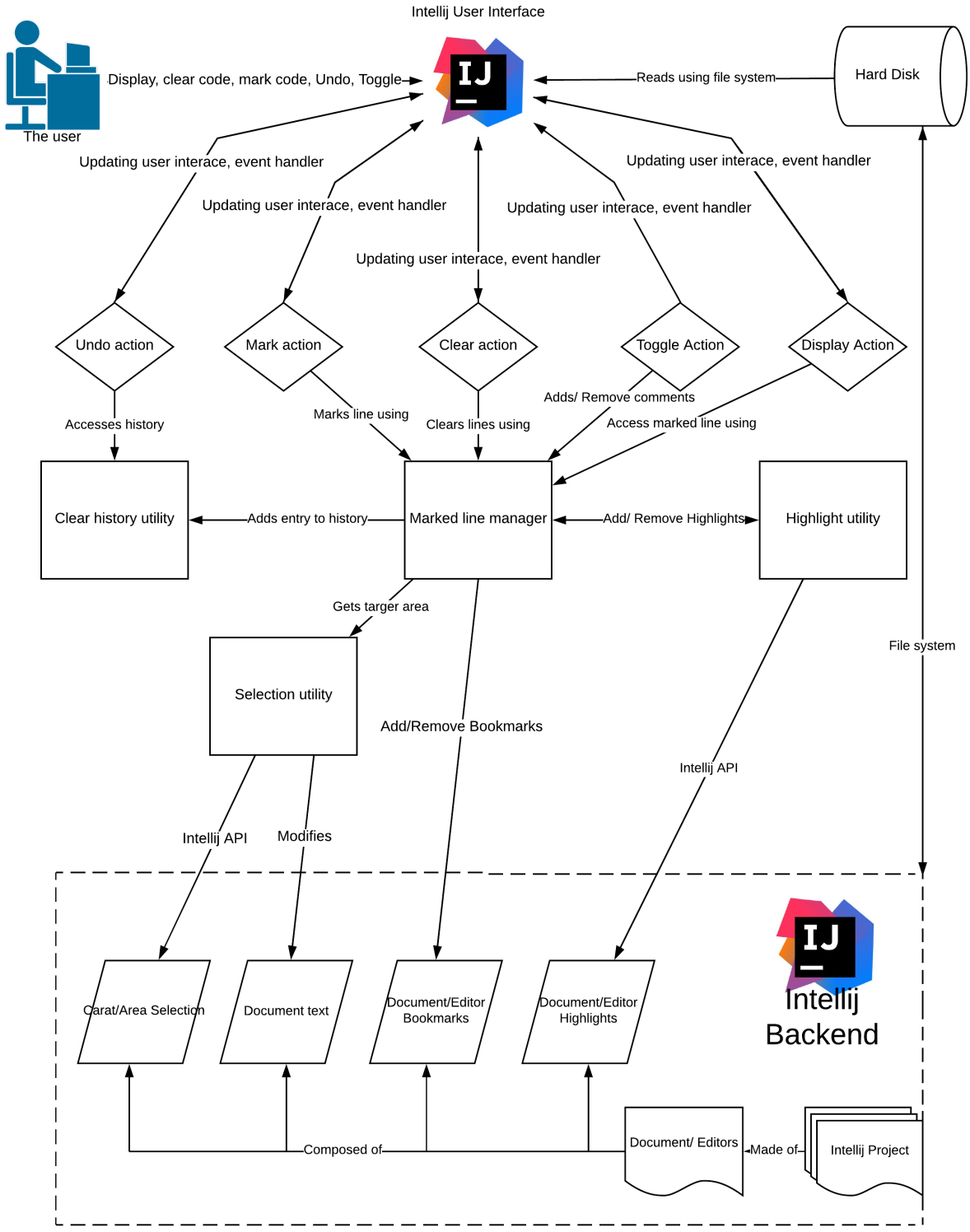

Figure 6: Architecture

An extensive list of different use cases and specific functionality interactions and behaviors in other text editors can be found in section 4 of our *user manual*.

4 Methodology of Experimentation and Testing

One of the main goal of the DeMark plugin is to increase productivity and make the best of developers' time. Development time is precious. Software engineers are highly paid, and if we can even save around 2-3 minutes of their time, it will dramatically save cost for the company due to the sheer amount of developers.

To access the tool's ability to accomplish this goal, we conducted multiple user trial experiments. We experimented on multiple versions of the tool, starting from the core functionality of marking and deleting/inserting temporary code, to having a persistent storage of temporary code. The data we gathered from the experimentation helped us evaluate our tool's contribution to the world of software development.

In addition, we also created an extensive test suite for DeMark to ensure that our implementation of the tool matches the specification that we created.

4.1 User Trial Experimentation

We had two stages of experimentation for DeMark. Each stage consisted of a case study where we distributed our tool to a group of users, and refine our tool based on their impressions and feedback.

The first goal of these experiments was to aid us in the development process of DeMark by using the data collected to further improve the tool in terms of function and user friendliness. Another goal of these experiments was to provide concrete data on the usefulness of the tool as well as its contributions to increasing developer efficiency.

The first stage of experimentation included the distribution of DeMark alpha version with the mark, unmark, clear, unclear, toggle, and display functions to specific developers who are friends and acquaintances of the DeFacto-UW team members. Along with the tool distribution itself, we also provided a user manual that contains preliminary instructions on how to install and use the tool.

We collected data on:

- Perceived improved productivity.
- Approximate of time saved.
- Ease of use.
- Thoroughness and clarity of instructions.
- Frequency of usage.
- Bugs and issues.

throughout the development of the alpha versions. A Google forms survey was provided to the users. The data helped the team determine the next stages of development for DeMark. The biggest challenge in this stage was the involvement of test users in reporting their experience as well as any bugs. The survey that was used for this stage can be found here: http://bit.ly/DeMarkPrelimSurvey along with questions and previous responses. The questions for the survey can also be found in the appendix section 7.1, and the results are discussed in section 5.1.1.

The second stage of experimentation involved distributing an improved beta version of DeMark to a larger set of developers. The improved version had most bugs fixed including compatibility issues. The data collected during this stage was similar to stage one's, with the tool having improved interface aesthetics and overall user experience. However, accessing a large user base who were willing to provide feedback and data was difficult. Additionally, how we collected data will once again have significant impact on the success of this stage. Because the tool was distributed to the public, we also collected data on each user's programming experience as well. The survey for the second stage can be found here: http://bit.ly/DeMarkStage2.

The survey for the first stage was conducted over the course of one day, with users only trying

out the plugin for several minutes. The survey for the second stage was conducted over the course of about 4 days.

4.2 Testing

Along with the user trial experiments, a test suite for DeMark was also designed and developed. The key in our tool is to have a notion of what our behavior should be and create tests to check for expected behavior based on the specification that we created for out tool. The majority of these tests were white box since our implementation is highly dependent on the IntelliJ environment, so it might be necessary to look at the code.

We also tested each individual component/module that we wrote for our tool. For example, we had a method that updates the line number given an action. We tested this method and made sure it updates the lines in the way that is correct. We also tested how the individual modules interacted with each other to form a system. For example, we tested how our system behaves when we both highlight code and delete code.

4.2.1 Branch and Path Coverage

We ran path/branch coverage tools such as JaCoCo with our test suit. These tools helped ensure that our test coverage executes a high percentage of code and enters a majority of branches. This is highly important because high coverage will increase confidence in the correctness of our program.

5 Initial Results of Experimentation and Testing

5.1 User Trial Experimentation

5.1.1 Stage 1

Stage one of the user trial experimentation included distributing an Alpha version of DeMark to a closed group of people known to the team. The data set that we have for stage one consists of *6* people who were not part of DeFacto-UW. The participants did not know about much about the plugin except a brief summary as seen in the appendix 7.1. Participants were then sent an email containing a ZIP archive of DeMark-Alpha-0.2 which includes the functions mark, unmark, clear, unclear, toggle, and display. Along with the distribution, participants were also provided with a set of instructions on how to install the tool as a plugin from disk onto IntelliJ, descriptions and instructions on how to use each of the function (which is taken directly from the user manual), and a link to a 5-section survey at the end.

The results gathered from stage one were mostly positive, with 5 out of 6 participants being able to install the plugin and test run it. Specifically, when asked about how often they would use DeMark, more than half of the participants responded with frequently to very frequently (appendix 7.1-Q3.a).

Moreover, when asked how useful they think DeMark would be for software development, 3 participants believe that the tool will be slightly useful, while 2 thought that it the tool will be very useful (appendix 7.1-Q3.b).

This result supports the idea that our tool would be useful and will be a tool that developers will want to make use of. Although the current data set is small, we expect that this trend will continue once we move into phase 2 of the user trial experimentation.

In terms of easiness to use, once again, all 5 people responded with “slightly easy” or higher for our tool (appendix 7.1-Q3.c). This may be because currently the only functions that are available

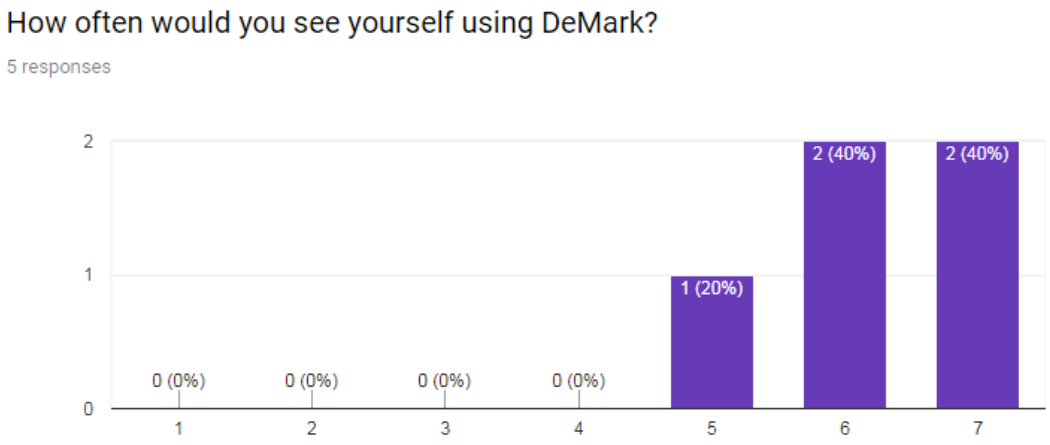

Figure 7: Question 3.a: 5 - "Slightly frequently", 6 - "Frequently", 7 - "Very frequently"

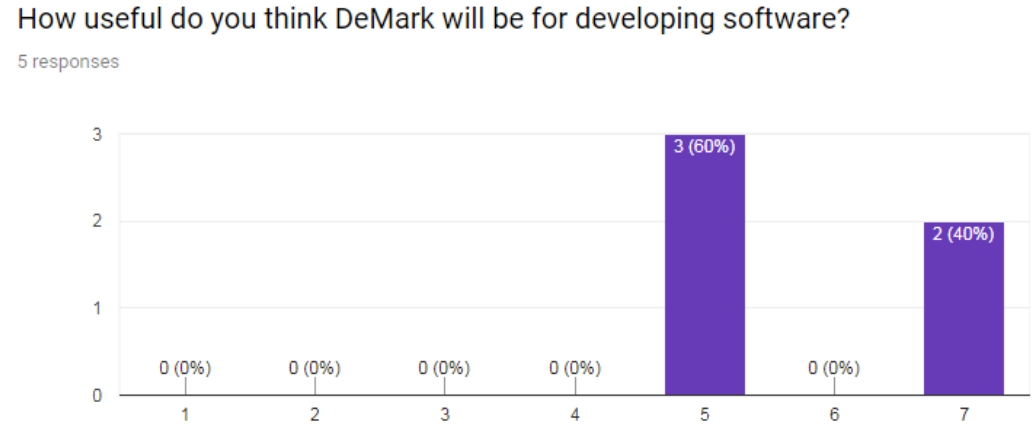

Figure 8: Question 3.b: 5 - "Slightly useful", 6 - "Useful", 7 - "Very useful"

are mark, unmark, clear, unclear, toggle, and display. Still, this results confirm that our current design for the tool in terms of function and flow has been successful, and users had no issues figuring out or understanding what each function or the tool as a whole does. It is necessary to note that while only 2 people rated the tool as only "slightly easy to use", 4 out 5 people found the keyboard shortcuts for DeMark to be not as intuitive as they want it to be (appendix 7.1-Q3.d).

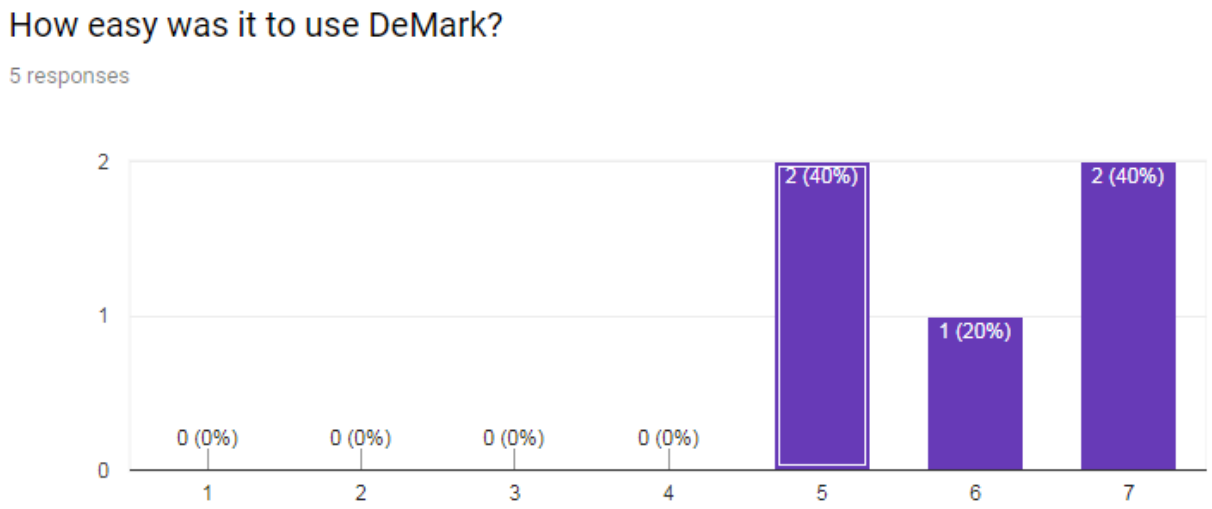

Figure 9: Question 3.c: 5 - "Slightly easy", 6 - "Easy", 7 - "Very easy"

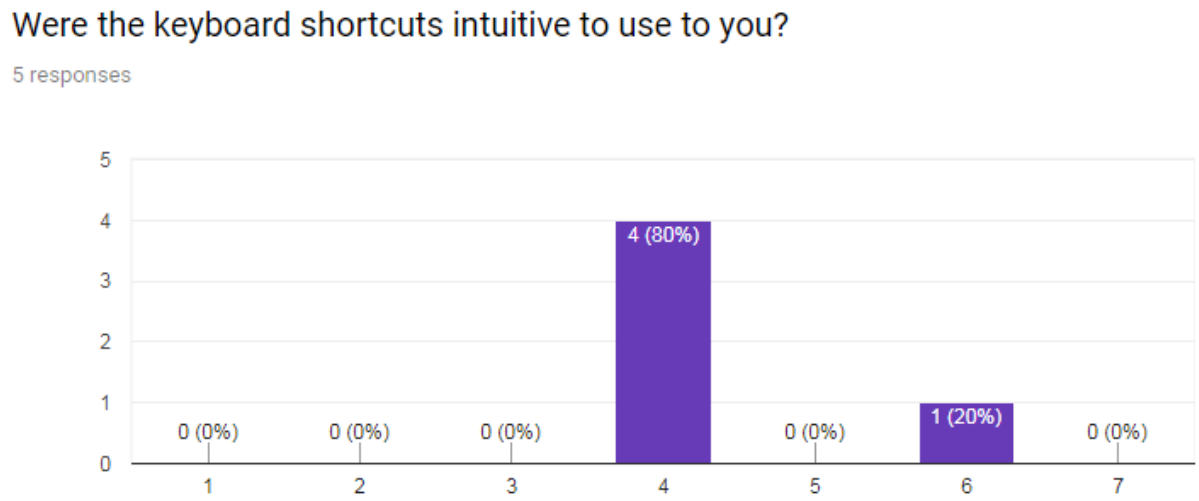

Figure 10: Question 3.d: 4 - "Not intuitive but not unintuitive", 5 - "Slightly intuitive", 6 - "Intuitive", 7 - "Very intuitive"

Currently, all of the default shortcuts starts with the `Alt` key. According to the feedback given after the intuitiveness question (appendix 7.1-Q3.e), starting the keyboard shortcut with the `Alt` can lead to some awkwardness in using the tool. We believe that this is normal as people are usually reluctant to learning new keyboard shortcuts but with time get used to them anyway.

Aside from collecting data on user experiences, this stage was also meant to help the team determine the status of DeMark's development. One of the main focus of this stage was determining whether or not a participant was able to install the plugin. This would indicate whether or not there were compatibility issue with different versions of IntelliJ. Compatability issues were resolved by the time stage 2 of experimentation started.

Additionally, section 4 of the survey (appendix 7.1-4) was focused on collecting participant's ideas on ways to improve DeMark as well as features that they would like the tool to have. Some of the feedback pointed out issues regarding the implementation of the tool itself (appendix 7.1-Q4.a), these will be the primary focus for development moving forward. Once we finish addressing those issues, we can move on to suggested features such as search and mark and different groups of marking (appendix 7.1-Q4.b).

5.1.2 Stage 2

Stage two of the user trial experimentation included distributing the Beta version of DeMark to the public through advertising at various sources, such as Facebook groups and Reddit. Participants were given a link to the survey, which contained links to the project repository, the installation manual, which includes steps on how to install the plugin, and the user manual, which includes instructions on how to use the each function of the plugin. At the end of the stage, we were able to gather responses from 11 people.

The participants were asked at the start of the survey about their experience in programming. Most responded with average experience in programming or higher, with the most specific being 3 years (appendix 7.2-Q1.a). Some of our users are most likely students in University as our plugin was also advertised on our University page and among our fellow peers. Then, the participants were asked about how often do they use print-f debugging, which all of them answered sometimes or more often.

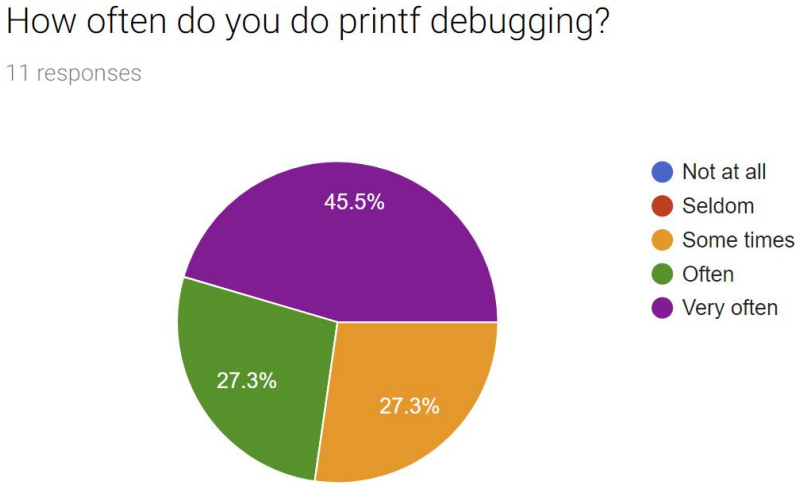

Figure 11: Question 1b

Overall, the results from stage two were once again mostly positive, with 10 out of 11 participants being able to install the plugin and use it. The survey was opened for 4 days such that the users were able to experience DeMark for a little bit without the pressure of filling out the survey. The results were split among the participants, with 4 participants only using the tool for less than 1 hour before the survey, 4 using it for more than 4 hours before the survey, and 2 using it for less than 2 hours (appendix 7.2-Q4.a). Even though usage time varies between all participants, the overall results were similarly positive.

One participant was unable to install DeMark (appendix 7.2-Q2.a), and one user was not able to install the tool from the JetBrains repository and had to install the tool from disk (appendix

7.2-Q2.c). This indicates that there were some issues regarding deployment of tool, although we could not reproduce these issues locally.

When asked about how often they would use DeMark in the future, more than half of the participants responded with every other day, with the remaining responses split between every week and every other week.

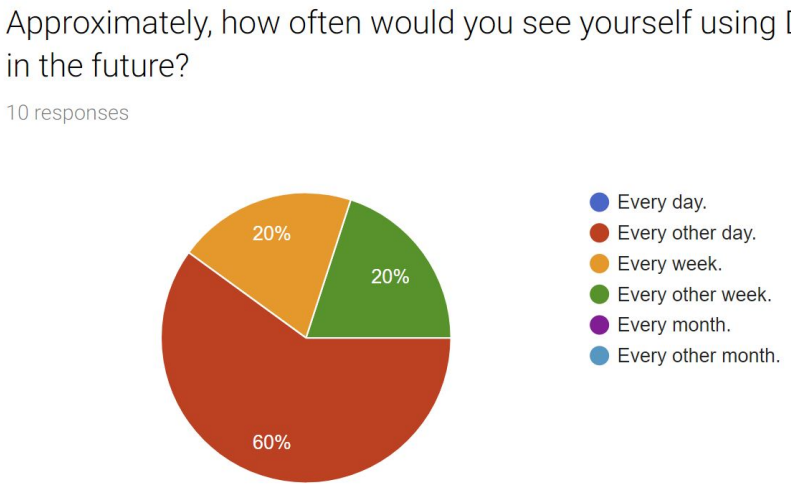

Figure 12: Question 4d

This data indicates that users will actively be using DeMark during development or debugging. Additionally, note that while 5 out of 10 participants says that they do print-f debugging very often (excluding the one that was unable to install) (appendix 7.2-Q1.b), 6 out of the 10 responded that they would use our tool every other day. This means that DeMark can be useful to some people who do slightly less print-f debugging.

Our users generally had positive feedback in regards to their experience using DeMarkȦll of users reported that they felt DeMark was an easy tool to use with 3 responding with "slightly easy", 5 responding with "easy" and 2 responding with "very easy" to use (appendix 7.2-Q4.e). The data collected shows that we have achieved our goal of being a simplistic tool that users could use to aid their everyday programming. This also shows that DeMark has a relatively low learning curve and can be learned quickly so the user can get started on using the tool as quickly as possible.

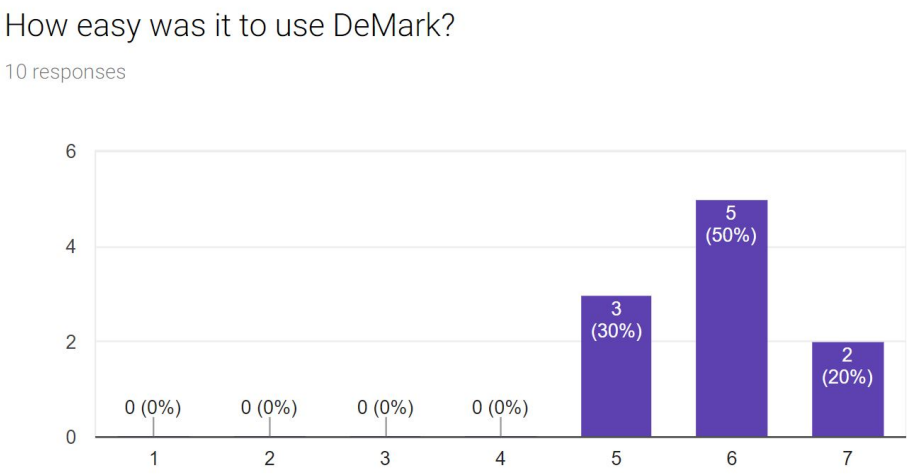

Figure 13: Question 4e: 5 - "Slightly easy", 6 - "Easy", 7 - "Very easy"

In terms of productivity improvement, all 10 of the participants responded that DeMark has improved their productivity slightly to quite a bit.

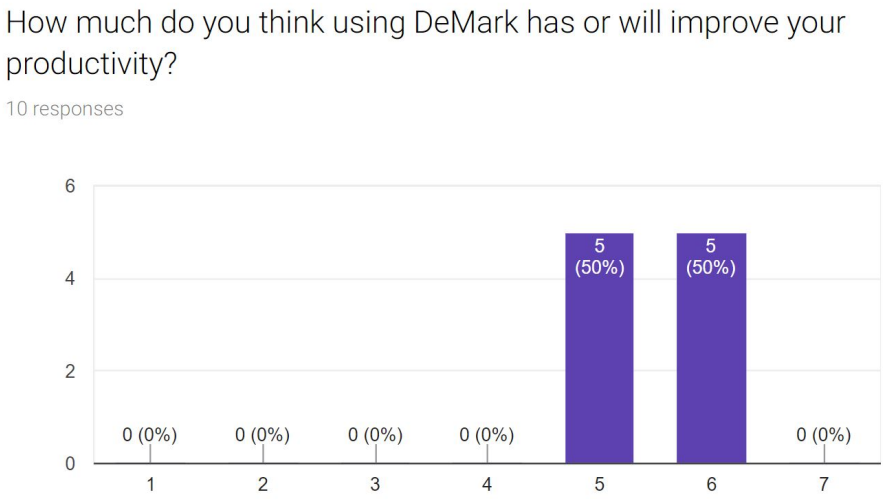

Figure 14: Question 4f: 5 - "Slightly", 6 - "Quite a bit", 7 - "A lot"

The usage data shows that people that use DeMark finds that it is a useful, easy to use tool that aided their productivity.

In addition to the usage data, participants were also asked for general feedback as well. Overall, the feedback given was mostly about features that users wanted DeMarkto have (appendix 7.2-Q4.b, Q5). For example, most users wanted a way to group marked lines and work with groups separately from each other. This feature was also critiqued for its absence in our developer case study (section 5.1.3). This will shift the priority of DeMark's development to support new features. However, we need to make careful design choices to keep our goal of making DeMark a simple tool.

5.1.3 Developer Case Study

Besides the distribution of the tool to a broader audience, we also had one of DeMark's developer extensively use the Beta version over the course of 1 week. Note that this developer was working on a distributed systems project implementing the Raft Consensus Algorithm. This means that he was using DeMark for an extensive course project. He was actively writing print statements to verify the correctness of the program flow.

The feedback from the developer was generally positive saying that the tool does what it advertises and works the way it should.

One interesting comment from our developer was that the work flow of our tool felt strange to him initially. He claimed that the mentality of actively debugging whilst coding is atypical. He believed that as developers we usually do not think too much about debugging until we know that our program has bugs or breaks. When we code, we usually just try to push out as much code as we can and then test it at a later time. He said that it took some work flow adjustment to make the best use of our tool.

Our initial intention was for the tool to be used during debugging, which will not introduce the work flow adjustment as mentioned above. However,

5.2 Testing: Branch and Path Coverage

Currently, we have successfully written tests for all of the classes in the `model.component` package in our project. The test suite also includes several tests for some of our utility classes, which includes `HighlightUtil` and `DeMarkUtil`. Furthermore, we have written use case tests for action classes that involves marking and unmarking (`MarkActionn`), clearing and unclearing (`ClearAction`), and commenting lines with toggling (`ToggleAction`). As a result, our current tests covers 61% of our code base.

To get this report, clone our repository and run this command "`cd DeMark/ && gradle check jacocoTestReport`", the report should then be in the folder `build/jacocoHtml` as "index.html". The coverage report can also be found in our `README.md` file on our repository [1]. As stated in section 4.2, we want a relatively high percentage of branch coverage for our tool, so part of finalizing DeMark's development will be writing tests for each of our component as well as use cases.

6 Total Hours Spent on this Assignment

We spent around 20 hours in total for this assignment.

References

[1] *Demark source code repository.* Available at `https://github.com/DeFacto-UW/DeMark/`.

[2] *Github commits removing print statements.* Available at `https://github.com/search?q=removed+unnecessary+print&type=Commits`.

[3] *Jetbrains plugin repository - demark.* Available at `https://plugins.jetbrains.com/plugin/10712-demark`.

[4] *Gradle Intellij Plugin.* Available at `https://github.com/JetBrains/gradle-intellij-plugin`.

[5] A. Hovmöller, *LineOps Intellij Plugin*. Available at `https://github.com/boxed/LineOps-intellij-plugin/`.

[6] *IntelliJ Platform SDK Guide*, Mar 2018. Available at `http://www.jetbrains.org/intellij/sdk/docs/intro/intellij_platform.html`.

[7] *Simple Logging Facade for Java (SLF4J)*. Available at `https://www.slf4j.org/`.

Appendix

7.1 Stage 1 User Trial Experimental Survey: Questions and Answers

---

The survey included this brief description of DeMark:

> The purpose of DeMark is to help provide a quick way to manage code that was meant to be temporary, with the biggest target being print-f debugging statements. The motivation is to provide a way to mark and clear those lines while debugging or coding, commit a clean version of your code, then unclear the lines and keep working.

The survey was conducted using Google Forms. As such, each question will have a *Type* of answer which will be noted. The questions were split into 4 sections comprising of 3 main section and a special case section.

The sections and questions are:

1. Installation
   (a) Did you successfully install the plugin? [**Type**: Multiple Choice]
   Installed just fine | **Could not install** - Leads to section 3 if first option is chosen, section 2 if second option is chosen.
   (b) How clear were the installation instructions? [**Type**: Linear Scale]
   0 - Not clear at all $\cdots$ **5 - Very clear**
   (c) If you rated a 3 or lower, please tell us how we could improve. [**Type**: Paragraph]
2. Installation Failed (Special case section, only reached if second of option of previous section is chosen)
   (a) If you could not install the plugin, please provide us with the following information:
   - IntelliJ build version. (This can be found by selecting [Help | About] in the IntelliJ menu.
   - The error message that was displayed.
   - Any addition information that you think will be useful.

   [**Type**: Paragraph]
3. Usage
   (a) How often would you see yourself using DeMark? [**Type**: Linear Scale]
   0 - No usage $\cdots$ **7 - Very frequently**
   (b) How useful do you think DeMark will be for developing software? [**Type**: Linear Scale]
   0 - Not useful at all $\cdots$ **7 - Very useful**
   (c) How easy was it to use DeMark? [**Type**: Linear Scale]
   0 - Very difficult $\cdots$ **7 - Very easy**
   (d) Were the keyboard shortcuts intuitive to use to you? [**Type**: Linear Scale]
   0 - Not at all $\cdots$ **7 - Very intuitive**
   (e) If you answered 3 or less for any of the above questions, please indicate why? [**Type**: Paragraph]
4. Feedback

(a) What do you think will improve your experience using DeMark? [**Type**: Paragraph]

(b) What features, if any, would you like added? [**Type**: Paragraph]

(c) What features do you find unnecessary or lacking in purpose? If so, why? [**Type**: Paragraph]

(d) Do you have any other general feedback? [**Type**: Paragraph]

In total there were 6 people who were not part of DeFacto participating in the survey. The full data on responses are:

Question 1.a: Did you successfully install the plugin? 6 responses:

Installed just fine: 5

Could not install: 1

Question 1.b: How clear were the installation instructions? 6 responses:
0: 0 | **1:** 0 | **2:** 0 | **3:** 0 | **4:** 2 | **5:** 4

Question 1.c: If you rated a 3 or lower, please tell us how we could improve. 1 response:

- Adding a note to make sure that IntelliJ is up to date, and indicating which versions of intellij are supported, (e.g.: 2018.1.3+) in the install instructions would help prevent dumb user moments. I had dumb user moments.

Question 2: If you could not install the plugin... 1 response:

- IntelliJ IDEA 2018.1 (Ultimate Edition) Build #IU-181.4203.550, built on March 26, 2018
  Plugin 'DeMark' is incompatible with this installation

Question 3.a: How often would you see yourself using DeMark? 5 responses:
0: 0 | **1:** 0 | **2:** 0 | **3:** 0 | **4:** 0 | **5:** 1 | **6:** 2 | **7:** 2

Question 3.b: How useful do you think DeMark will be for developing software? 5 responses:
0: 0 | **1:** 0 | **2:** 0 | **3:** 0 | **4:** 0 | **5:** 3 | **6:** 0 | **7:** 2

Question 3.c: How easy was it to use DeMark? 5 responses:
0: 0 | **1:** 0 | **2:** 0 | **3:** 0 | **4:** 0 | **5:** 2 | **6:** 1 | **7:** 2

Question 3.d: Were the keyboard shortcuts intuitive to use to you? 5 responses:
0: 0 | **1:** 0 | **2:** 0 | **3:** 0 | **4:** 4 | **5:** 0 | **6:** 1 | **7:** 0

Question 3.e: If you answered 3 or less... 2 responses:

- For keyboard users, the keyboard shortcuts is intuitive. I tested DeMark on a laptop and the keyboard layout using the keyboard of the laptop isn't very intuitive as it is harder to hit `alt + m` on that layout due to the position of the arrow keys.
- Alt is an intimidating shortcut hotkey beginning, lol.
  I think that including a shortcut for the Toggle feature would be useful. Having a hotkey for delete, but not to comment is... interesting.

Question 4.a: What do you think will improve your experience using DeMark? 4 responses:

- I am not sure so far it seems pretty useful and cool.
- Make the plugin more compatible with `Crtl + Z`, make the toggle button smaller
- A little better interface would go a long way.
  I'm personally not a fan of the power button icon for the toggle feature... it's unintuitive, and doesn't clearly convey what the button does. Also, it's the largest icon on the menu row after my fresh install of IntelliJ.
- Give some useful usage cases for DeMark in instructions PDF (i.e. "use this to temporary toggle debugging/temporary code such as print statements").

Question 4.b: What features, if any, would you like added? 4 responses:

- I think it'd be cool if you could search for a line containing some text and automatically mark it that way
- I can't think of a feature that is nice to have because for a plugin that mark line and clear them it is doing quite well as is
- Grouped marks. Being able to toggle multiple sets of marks would be super useful: for example, if you had print statements spread across several functions, for different parts of bug testing, being able to toggle group 1 vs group 2 etc would be super useful.
- Add a feature where marked lines would print a unique number each time that line is reached (useful for catching off-by-one errors, dead code, etc.)

Question 4.c: What features do you find unnecessary... 2 responses:

- Nothing in particular
- n/a

Question 4.d: Do you have any other general feedback? 3 responses:

- No comment
- Clever concept! Good luck∼
- nope

7.2 Stage 2 User Trial Experimental Survey: Questions and Answers

The survey included this brief description of DeMark:

> The purpose of DeMark is to help provide a quick way to manage code that was meant to be temporary, with the biggest target being print-f debugging statements. The motivation is to provide a way to mark and clear those lines while debugging or coding, commit a clean version of your code, then unclear the lines and keep working.

The survey was conducted using Google Forms. As such, each question will have a *Type* of answer which will be noted. The questions were split into 4 sections comprising of 3 main section and a special case section.

The sections and questions are:

1. Experience
   (a) How experienced are you at programming? [**Type**: Short Answer]
   (b) How often do you do printf debugging? [**Type**: Multiple Choice]
   Not at all | **Seldom** | **Sometimes** | **Often** | **Very often**
2. Installation
   (a) Did you successfully install the plugin? [**Type**: Multiple Choice]
   Installed just fine | **Could not install** - Leads to section 3 if first option is chosen, section 4 if second option is chosen.
   (b) How clear were the installation instructions? [**Type**: Linear Scale]
   0 - Not clear at all $\cdots$ **7 - Very clear**
   (c) If you rated a 3 or lower, please tell us how we could improve. [**Type**: Paragraph]
3. Installation Failed (Special case section, only reached if second of option of previous section is chosen)
   (a) If you could not install the plugin, please provide us with the following information:
   - IntelliJ build version. (This can be found by selecting [Help | About] in the IntelliJ menu.
   - The error message that was displayed.
   - Any addition information that you think will be useful.

   [**Type**: Paragraph]
4. Usage
   (a) Approximately, how much time did you use DeMark before this survey? [**Type**: Multiple Choice]
   < 1 hour | **< 2 hour** | **< 3 hour** | **< 4 hour** | **> 4 hour**
   (b) What do you think DeMark could improve so you would use it more? [**Type**: Paragraph]
   (c) How clear were the instructions on the user manual? [**Type**: Scale]
   1 - Not clear at all $\cdots$ **7 - Very clear**

(d) Approximatelly, how often would you see yourself using DeMark in the future? [**Type**: Multiple Choice]
Every day. | **Every other day.** | **Every week.** | **Every other week.** | **Every month.** | **Every other month.**

(e) How easy was it to use DeMark? [**Type**: Linear Scale]
1 - Very difficult $\cdots$ **7 - Very easy**

(f) How much do you think using DeMark has or will improve your productivity? [**Type**: Linear Scale]
1 - Not at all $\cdots$ **7 - A lot**

(g) If you answered 3 or less for any of the above questions, please indicate why? [**Type**: Paragraph]

5. Feedback

(a) What do you think will improve your experience using DeMark? [**Type**: Paragraph]

(b) What features, if any, would you like added? [**Type**: Paragraph]

(c) What features do you find unnecessary or lacking in purpose? If so, why? [**Type**: Paragraph]

(d) Do you have any other general feedback? [**Type**: Paragraph]

In total there were 11 people participated in the survey. The full responses are shown below:

Question 1.a: How experienced are you in programming?

- A lot
- Experienced - but still terrible at it.
- Lot
- Moderately
- Experienced. CS major
- Okayish
- Pretty experienced
- 3 years
- average
- moderately experienced

Question 1.b: How often do you do printf debugging?
Not at all: 0 | **Seldom:** 0 | **Sometimes:** 3 | **Often:** 3 | **Very often:** 5

Question 2.a: Did you successfully install the plugin? **Installed just fine:** 10 | **Could not install:** 1

Question 2.b: How clear were the installation instructions?
1: 1 | **2:** 0 | **3:** 0 | **4:** 0 | **5:** 0 | **6:** 6 | **7:** 4

Question 2.c: If you rated a 3 or lower, please tell us how we could improve

- I couldn't find the plugin in the IntelliJ repositories so I had to download from disk

- You could try asking for my email?

Question 3: If you could not install the plugin, please provide us with the following information:

- IntelliJ build version. (This can be found by selecting [Help | About] in the IntelliJ menu.
- The error message that was displayed.
- Any addition information that you think will be useful.

No response.

Question 4.a: Approximately, how much time did you use DeMark before this survey?
< 1 hour: 4 | **< 2 hour:** 2 | **< 3 hour:** 0 | **< 4 hour:** 0 | **> 4 hour:** 4

Question 4.b: What do you think DeMark could improve so you would use it more?

- Instead of having a menu bar thing, I feel that it should be something like the bottom windows. Like how IntelliJ has the Terminal and Version Control things at the bottom (I don't remember what it is called). This is because I would rather like to have a button to mark these statements than to go to the menu bar again and again. Shortcuts are fine but this is something necessary in my opinion.
- I think it'd help if there was a way to make them all commented or uncommented. Sometimes I'd have lines commented out and some not. Then when I wanted them all commented out I'd have to go through and do it manually.
- Another way to toggle specific marked lines rather than all. More often than not, I want to disable just a portion of the prints because its flooding the console. It would be particularly useful to continue having the marked lines but disabled a portion of them.

  Sometimes the toggle become messed up because I might manually comment out some or forget to toggle one out. etc.
- MacOS default key mapping on manual
- More accessibility instead of just a menu option and keyboard shortcut
- Automatically mark lines with print statements would be better

Question 4.c: How clear were the instructions on the user manual?
1: 0 | **2:** 0 | **3:** 0 | **4:** 0 | **5:** 1 | **6:** 6 | **7:** 3

Question 4.d: Approximately, how often would you see yourself using DeMark in the future?
Every day: 0 | **Every other day:** 6 | **Every week:** 2 | **Every other week:** 2 | **Every month:** 0 | **Every other month:** 0

Question 4.e: How easy was it to use DeMark?
1: 0 | **2:** 0 | **3:** 0 | **4:** 0 | **5:** 3 | **6:** 5 | **7:** 2

Question 4.f: How much do you think using DeMark has or will improve your productivity?
1: 0 | **2:** 0 | **3:** 0 | **4:** 0 | **5:** 5 | **6:** 5 | **7:** 0

Question 4.g: If you answered 3 or less for any of the above rating questions, please indicate why.
No response.

Question 5.a: What do you think will improve your experience in using DeMark?

- Making it much more reachable than the menu bar.
- There is a small bug in the app actually. When you mark a line, the line gets coloured grey (I love this). But when you delete this line, the line is still coloured grey, so the grey colour does not go away basically. Should be a small fix imo. :)
- Some way to clear if-statements or loops without marking everything.
- Marking lines with print will make sure I don't forget to delete one line of print

Question 5.b: What features, if any, would you like added?

- Maybe finding all statements marked by DeMark if needed. Scrolling on a very big file can be a pain.
- A search that marks everything
- Selection Based Toggling.
- Being able to delete if's and loops.
- Automatically mark lines and more color schemes

Question 5.c: What features do you find unncessary or lacking in purpose? If so, why?

- Display was unused.
- no

Question 5.d: Do you have any other general feedback?

- I don't like the toggle icon. Maybe something smaller
- Great plugin overall. A little more features such as selection toggling and maybe even searching would greatly benefit.
- no

7.3 Schedule

We broke down DeMark's development by weeks. Within each week, there is a main goal and with each main goal there are sub goals to to keep us moving forward to achieve the main goal. The plan proposed here is subject to change as time goes along. Regardless, staying on schedule is still preferred. If not possible, we will adjust our plan and seek staff help to ensure things are going in the right direction.

Our plans for week 7 and on have changed slightly due to stage 1 experimentation being finished in week 7. We are currently on schedule in terms of development.

✓ **Week 2**: Initial project proposal writeup

✓ **Week 3**: Begin tool development and gain an understanding of IntelliJ API

- Each developer will familiarize themselves with the documentation layout.
- Each developer will do the quick start tutorial and do research on tool development.
- Revise algorithm to adapt to the API.
- **Project Due**: Architecture and implementation plan.

✓ **Week 4**: Push towards a draft implementation where user is able to mark and un-mark lines

- Finalize adaption of Intellij api and DeMark algorithm.
- Revise experiments and find sample group to present beta in for later weeks.

✓ **Week 5**: Continuation of development

- A working copy of the basic features we want.
- Code review each developer's code and make sure everyone is on the same page.
- **Project Due**: Revised project proposal.

✓ **Week 6**: Additional features if time. Otherwise continue basic implementation

- Features: Keyboard shortcuts, persistence of marked lines, switchable profiles, ability to undo actions
- Begin using the tool once the basic implementation is finished to gather data.
- **Project Due**: Revised project proposal.

✓ **Week 7**: Experimentation Design and Implementation and User Trial Experimentation Stage 1

- Alpha version fully functional.
- Experimentation framework set up for bug report and issue logging.
- User feedback framework set up.
- Data collected for stage 1.
- **Project Due**: Initial project result.

✓ **Week 8**: Issue Fixing, Debugging, and Tests

- Add unit tests to test suit, increase test coverage by at least 10%.
- At least three fourth of known issues are resolved.
- Fix compatibility issues with older versions of IntelliJ.
- Work on slides for project presentation.
- **Project Due**: Project presentation.

✓ **Week 9**: User Trial Experimentation Stage 2

- Implemented new features, if any.
- Less than 3 issues are known for tool.
- Beta version ready for forum distribution.

- Testing of all implemented functionality among a wider group of participants e.g. public forums.
- Collected and consolidated data and feedback from experiments.
- **Project Due**: Draft final report.

✓ **Week 10**: Analyze results and present our project

- Whether it be a large or small sample size, we will need to form an analysis of our results from the experiments.
- Practice presentation with group.
- **Project Due**: Repository review.

Week 11: Finalize project

- Practice presentation.
- **Projects Due**: final presentation slides, final project presentation, final report resubmission.

7.4 Discussion And Lessons Learned

7.4.1 Growth of The Team and Its Members

It is not an easy feat to create a piece of software from scratch and we have learned a lot from the process. Throughout the course of DeMark's development as a plugin, we have also developed as designers, technical writers, programmers, and teammates. DeMark taught us first-hand on how difficult the software development process can be.

As designers, we were able to take a vision of a tool and refined it into something more concrete and tangible, we transformed the ideas in our head into a real tool. One specific challenge that we faced was with the specification of our tool. As students, we are usually given specifications for a project and are supposed to implement it. With DeMark, there were no specifications given, we needed to design it on our own. We learned the importance of design in software development since a good design can lead to a better implementation and easier maintainability. We wanted our utilities to be as loosely coupled as possible and every function written to do one thing. So, we started with simple functions such as adding a highlight or adding a bookmark or checking if a line was marked. These functions were heavily used throughout DeMark's development. We realized that starting with the simple case and taking it slow can have a huge benefit for later development because the early stages are the foundation of the product itself.

With that, we grew as technical writers as well. We needed to be able to articulate our motivations, design, and ideas well to our users. Our growth as technical writers, then, is shown through each iteration of our report. By revising our report based on peer reviews and staff criticism regularly, we were able to make noticeable improvements through every submission. Furthermore, we were able to draw on past work and solutions to not only strengthen our own work, but also our motivation.

As programmers, we learned how hard it can be to deal with existing third-party software, as all our features revolved around the IntelliJ SDK, which was quite lacking in terms of documentation. However, we were constantly learning about the IntelliJ SDK platform, redesigning and improving our architecture so that it will fit well with the existing IntelliJ API. As a result, we were able to design and implement interesting algorithms and extend the existing IntelliJ feature set successfully.

Additionally, there were numerous challenges that we overcame as a team. We experienced a lot of the challenges of working in a larger group setting. None of our teammates had significant experience before in working in a group toward a common goal of developing a specific tool, as such we experienced a lot of challenges such as managing work division and communication. Over the course of developing the tool together, we established an efficient communication and work division system that significantly increased our productivity as a team. One strategy that we used was scheduling frequent meetings and discussing our vision of the project, which helped clear up any confusion about where we were heading in terms of development. This, along with having mini deadlines throughout the quarter allowed the team to stay on track for the project.

7.4.2 Learning About Tools and Best Practices

We also learned a great deal about real life software tools and practices. As students, we were not exposed to tools such as Travis CI, JaCoCo, and many advanced features of Git and GitHub before this class. As a team, we learned how important it is to maintain a clean and organized code base, and we learned how to setup and use software development tools like Travis CI and JaCoCo to ensure that our code base is continually well-tested. Another important practice we learned is the usage of the some of the more advanced features of GitHub such as issue tracking. Students are not

really exposed to these everyday parts of real world software development, and that is unfortunate as we feel that learning these tools are invaluable for actual software engineering.