DeMark Project

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

May 10, 2018

1 Introduction

More often than not, developers want to see 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 or the overall flow of the program. This is also known as "printf" debugging. These lines of code are usually there 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" [1].

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 lines of code that are meant to be temporary, 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.

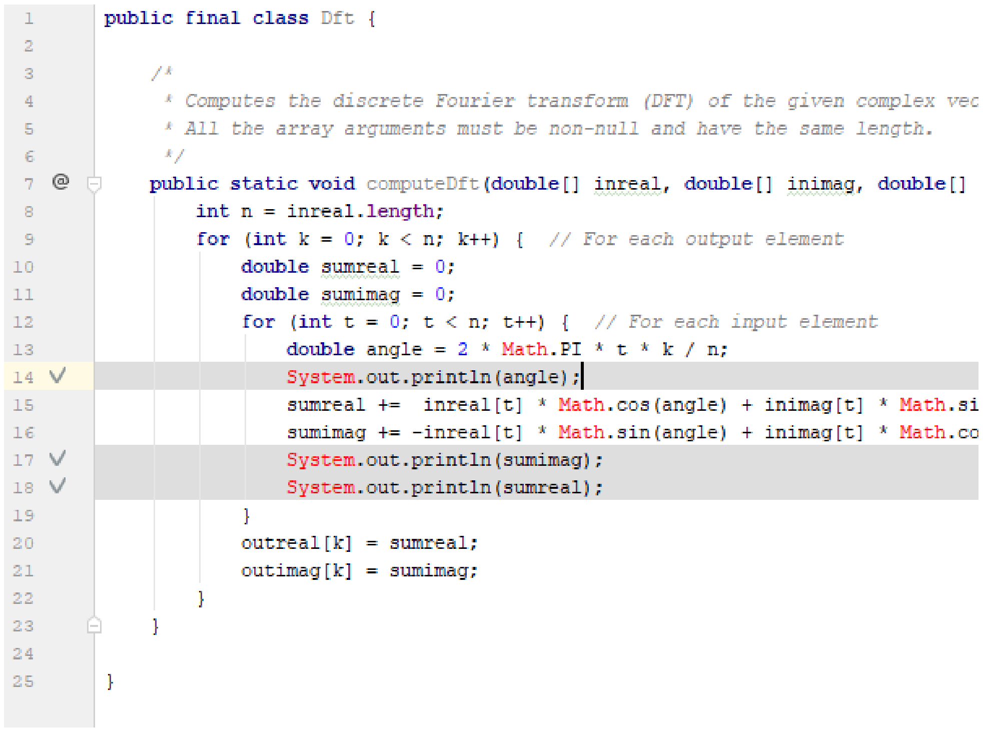
2 Description

Our tool allows developers to mark lines of code in their program through the IDE. When our tool is not in use and no lines of code are marked, the IntelliJ editor will look the same as its default state. Figure 1 illustrates what the interface will look like after the user has marked several lines of code.

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 through their description, which contains the word "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 2 illustrates what the interface looks like after the developer deletes the previously marked lines of code. This is represented as a menu option integrated into the IDE and can also be invoked using a keyboard shortcut. Marking and deleting lines of code together are the most basic and core features of DeMark.

Another important feature that we implemented is the ability to undo the lines of code that have just been cleared, essentially re-adding them to the code. There should be some semblance of an action history where developers can look at what has previously done and undo those actions if they desire. For example, lets say a user deletes a marked line. The tool should communicate to the developer that they took that action and allow them to undo the action.

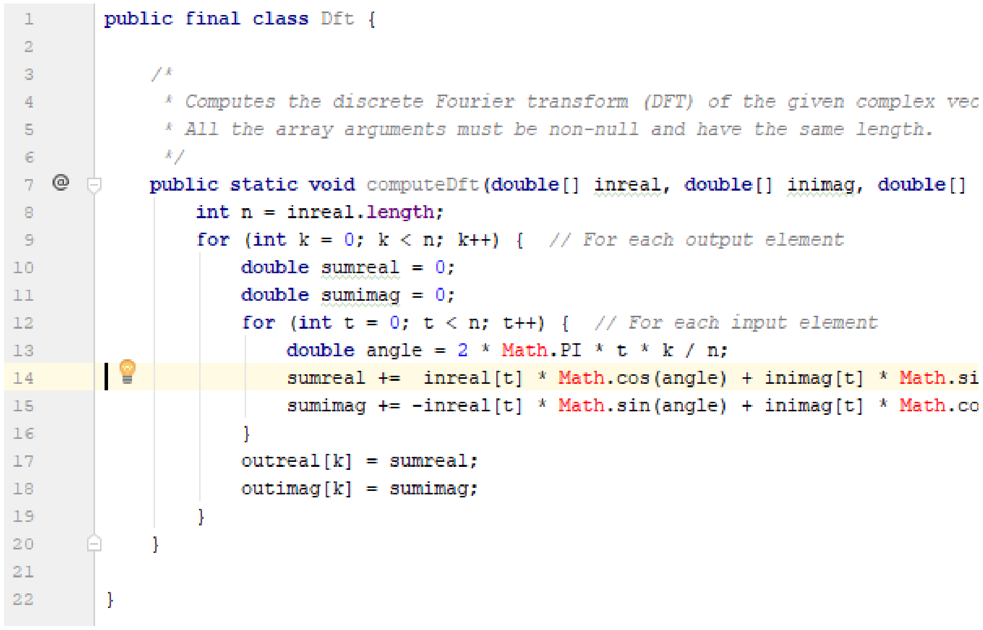


```
public final class Dft {

    /*
     * 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 1: Before clearing marked lines



```
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 2: After clearing marked lines

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

3 Existing Solutions and Their Limitations

Currently, there are several ways of dealing with finding bugs or understanding program states. One way is by using logging levels and loggers [5]. 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, this method requires integration into the source code. This could lead to difficulties in using the tools or improper usage that leads to inefficiencies, especially for beginner programmers. In theory, logging tools can provide the developer with a powerful tool that they can fine tune to their own needs. However, in practice logging tools can be more of a hassle than they are worth especially for short term debugging. Print statements have a much lower learning curve and are simpler to setup.

Another method is through the addition of the temporary, non-production lines of code, where the developer can search and manually delete those lines later. This way is simplistic and often requires no extra tools 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 bases. It also leaves room for "leftover" temporary code that the developer possibly missed. 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 line to commit. This method, however, is only helpful in handling temporary code that is added to the code base during the commit phase. 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. However, this method requires the developer to do a small mental reconstruction of what their code actually does to decide whether or not to add 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.

DeMark allows developers to mark lines as they are typing or immediately after they are typing, and then later delete those marked temporary code. DeMark is a tool that helps developers who use logging, print statements, or any programming style that produces transient lines of code. This tool has the potential to improve the experience of those who code in this style by streamlining the process of removing such lines of code.

In terms of plugins, there is currently a 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 [3]. However, LineOps only allows the user to clear bookemarked 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.

4 Architecture

Our architecture revolves around the IntelliJ Platform SDK. 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.

Along with providing a way to portray our functions as visual elements and classes that can be called upon and created, the IntelliJ SDK also provides us a way to collect information from the IDEA itself. Mainly, the SDK provides access to the IntelliJ text editor and files through an `Editor` class and a `Document` class. Both of these classes contains information on various editor elements like highlights, bookmarks, caret positions, selections, etc. The SDK also provides a `Project` class which allows the tool to gain information on all files in a particular project.

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 line.

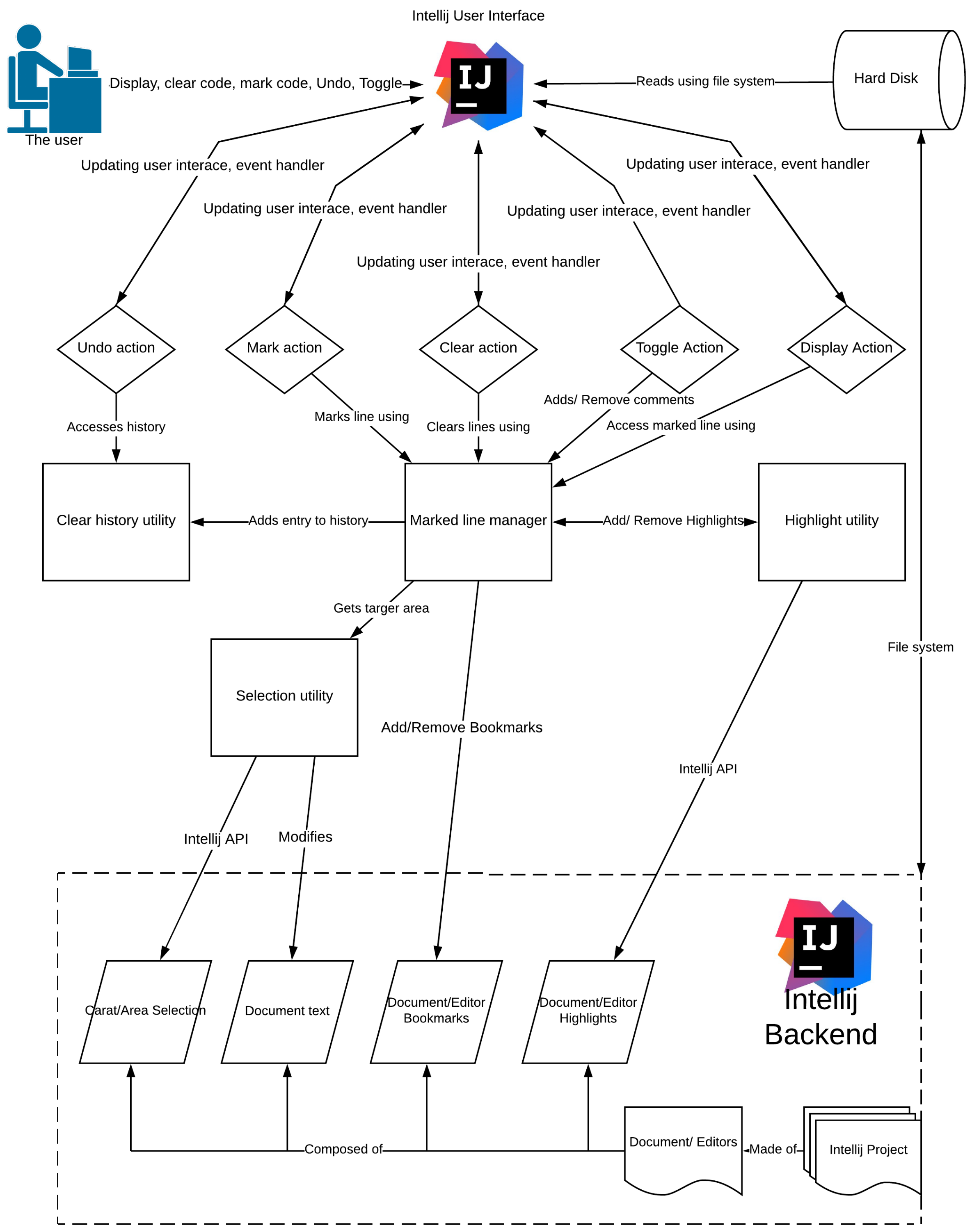


Figure 3: Architecture

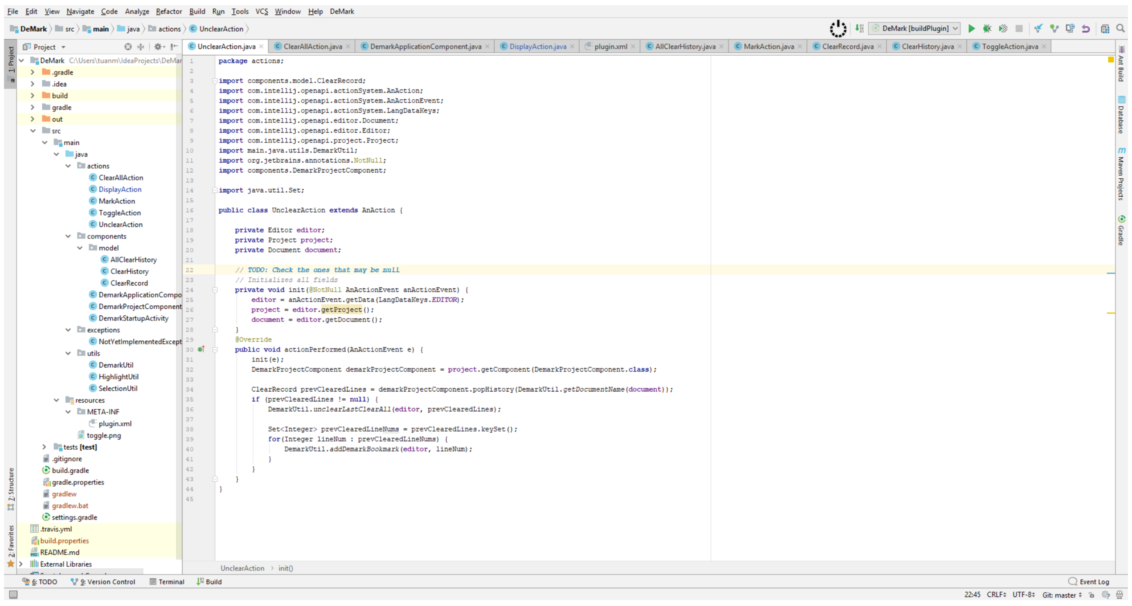
An extensive list of different use cases and specific functionality interactions and behaviors in other text editors can be found in section 5 of our *user manual*.

5 Interfaces

5.1 Extended System

The IntelliJ 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” [4]. This allows us to easily develop the tool as a plugin for IntelliJ using this platform.

5.2 Graphical User Interface



```java
package actions;

import components.model.ClearRecord;
import com.intellij.openapi.actionSystem.AnAction;
import com.intellij.openapi.actionSystem.AnActionEvent;
import com.intellij.openapi.actionSystem.LangDataKeys;
import com.intellij.openapi.editor.Document;
import com.intellij.openapi.editor.Editor;
import com.intellij.openapi.project.Project;
import main.java.utils.DemarkUtil;
import org.jetbrains.annotations.NotNull;
import components.DemarkProjectComponent;

import java.util.Set;

public class UnclearAction extends AnAction {

    private Editor editor;
    private Project project;
    private Document document;

    // TODO: Check the ones that may be null
    // Initializes all fields
    private void init(@NotNull AnActionEvent anActionEvent) {
        editor = anActionEvent.getData(LangDataKeys.EDITOR);
        project = editor.getProject();
        document = editor.getDocument();
    }
    @Override
    public void actionPerformed(AnActionEvent e) {
        init(e);
        DemarkProjectComponent demarkProjectComponent = project.getComponent(DemarkProjectComponent.class);

        ClearRecord prevClearedLines = demarkProjectComponent.popHistory(DemarkUtil.getDocumentName(document));
        if (prevClearedLines != null) {
            DemarkUtil.unclearLastClearAll(editor, prevClearedLines);

            Set<Integer> prevClearedLineNums = prevClearedLines.keySet();
            for(Integer lineNum : prevClearedLineNums) {
                DemarkUtil.addDemarkBookmark(editor, lineNum);
            }
        }
    }
}
```

Figure 4: Original Interface

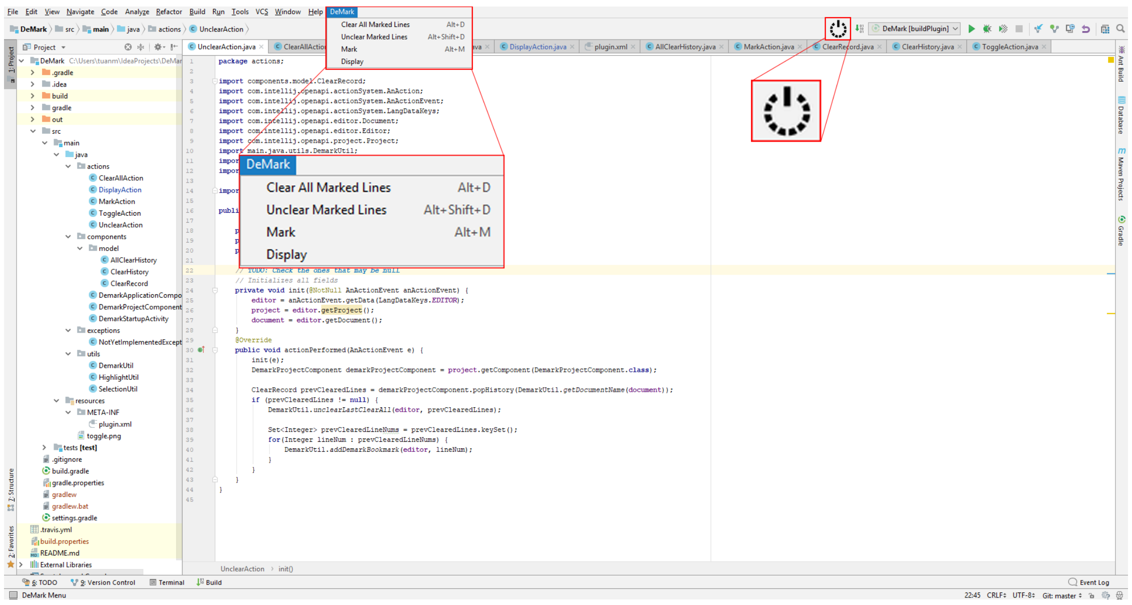
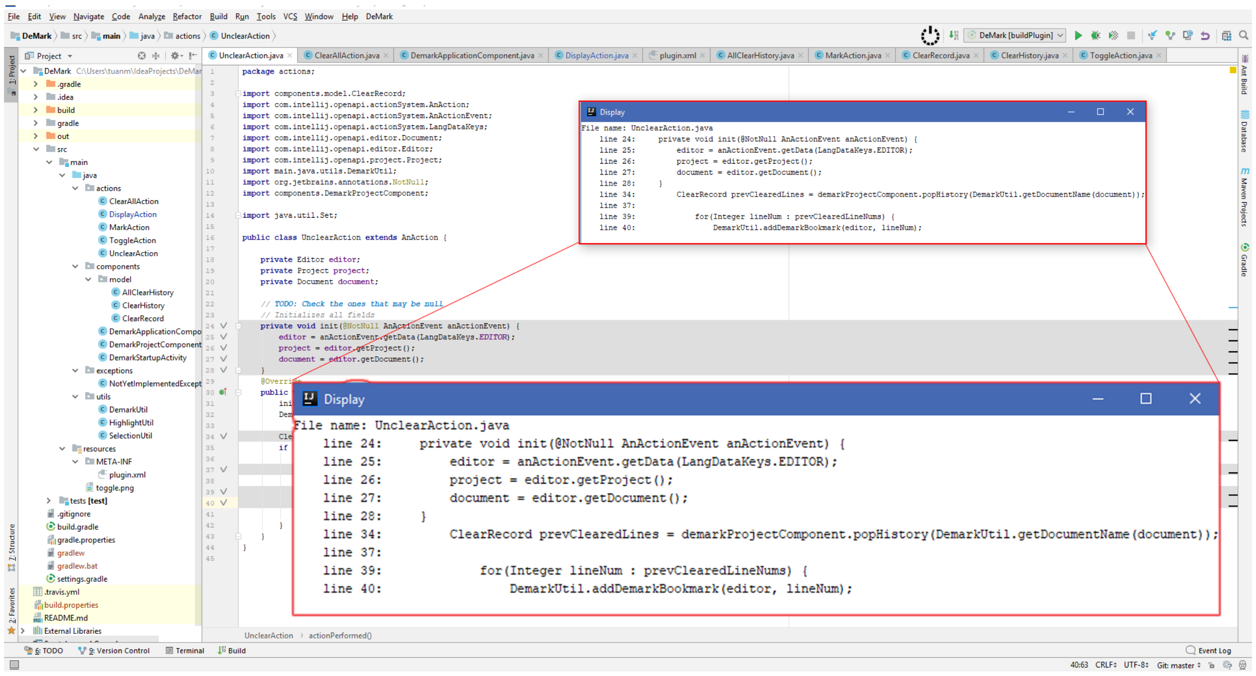

DeMark
Clear All Marked Lines Alt+D
Unclear Marked Lines Alt+Shift+D
Mark Alt+M
Display
DeMark Menu

Figure 5: Plugin Installed

Figure 4 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 5 in the drop down menu.

The power button in figure 5 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.



```
package actions;

import components.model.ClearRecord;
import com.intellij.openapi.actionSystem.AnAction;
import com.intellij.openapi.actionSystem.AnActionEvent;
import com.intellij.openapi.actionSystem.LangDataKeys;
import com.intellij.openapi.editor.Document;
import com.intellij.openapi.editor.Editor;
import com.intellij.openapi.project.Project;
import main.java.utils.DemarkUtil;
import org.jetbrains.annotations.NotNull;
import components.DemarkProjectComponent;

import java.util.Set;

public class UnclearAction extends AnAction {

    private Editor editor;
    private Project project;
    private Document document;

    // TODO: Check the ones that may be null
    // Initializes all fields
    private void init(@NotNull AnActionEvent anActionEvent) {
        editor = anActionEvent.getData(LangDataKeys.EDITOR);
        project = editor.getProject();
        document = editor.getDocument();
    }
    @Overri
    public
        ini
        Dem

        Cle
        if


            }
    }
}
```

```
File name: UnclearAction.java
    line 24:     private void init(@NotNull AnActionEvent anActionEvent) {
    line 25:         editor = anActionEvent.getData(LangDataKeys.EDITOR);
    line 26:         project = editor.getProject();
    line 27:         document = editor.getDocument();
    line 28:     }
    line 34:         ClearRecord prevClearedLines = demarkProjectComponent.popHistory(DemarkUtil.getDocumentName(document));
    line 37:
    line 39:             for(Integer lineNum : prevClearedLineNums) {
    line 40:                 DemarkUtil.addDemarkBookmark(editor, lineNum);
```

Figure 6: 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 6 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.

6 Technologies

The tool we are developing is a plugin that adds new interactive components and new functionality to the IDE. Since we develop for the IntellijJ IDE, we are using IntelliJ Platform Plugin SDK, which provides the most important APIs we need to build our tool. We are using gradle-intellij-plugin as the workflow 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. [2] Additionally, we are using Java 8 for our development language since that is the language of the IntelliJ Platform.

7 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 are conducting multiple user trial experiments as well as creating an extensive test suite for DeMark. We plan to experiment 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 are gathering from the experimentation will display our tool's contribution to the world of software development.

7.1 User Trial Experimentation

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

The first goal of these experiments is to aid us in the development process of DeMark as the data that we collect will allow us to further improve the tool in terms of function and user friendliness. Another goal of these experiments is 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 closely connected with the DeFacto-UW team. Along with the tool distribution itself, we also provided preliminary instructions on how to use the tool. Data was collected 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 who tested the alpha version of the tool as a way to measure our initial results. The data helped aid the team determine the next stages of development for DeMark, specifically whether or not to add addiitonal features and functionality. The biggest challenge in this stage was the involvement of test users in reporting their experience as well as any bugs. Designing and implementing an accessible reporting system as well as error tracking system for DeMark was essential to the success of this initial stage. This is the link to the survey used in the first stage of experimenting: `http://bit.ly/DeMarkPrelimSurvey`.

The second stage of experimentation will involve distributing an improved beta version of DeMark to a larger set of developers. The improved version will have most bugs fixed as well as some new functionality included. The data that will be collected during this stage will be similar to stage one, with the addition of possible features, interface aesthetics, and overall user experience. However, accessing a large user base who are willing to provide feedback and data will be difficult.

Additionally, how we are going to collect data will once again have significant impact on the success of this stage.

We must note that in order for both stages of experimentation to provide impactful data, each stage will have to be conducted over at least one week with enough time between the alpha and beta versions for there to be significant program improvements. As a result, another challenge for the experiments will be allocating time for both developing and experimenting. Finding users who are willing to test our product will also be another challenge in the experimental stage and so we will try to plan ahead and find people who willing to use our tool in its early stages.

The data for both phases will be collected through surveys that will ask about user experience, GitHub's issue tracker, the IntelliJ plugin download tracker to measure how many people has download.

7.2 Testing

Along with the user trial experiments, a test suite for DeMark will also be 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. Of course, expected behavior is a very subjective term and will depend on the specification that we create for ourselves. The majority of these tests might have to be white box since our implementation will be highly dependent on the IntelliJ environment, so it might be necessary to look at the code.

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

7.2.1 Branch and Path Coverage

Once we have written our test suite, we will run path/branch coverage tools such as JaCoCo using our tests suites. These tools will help to ensure that our test coverage executes a high percentage of code and enters the majority of branches. This is highly important since high coverage will increase confidence in the correctness of our program.

8 Initial Results of Experimentation and Testing

8.1 User Trial Experimentation

8.1.1 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 10.1. 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 10.1-Q3.a).

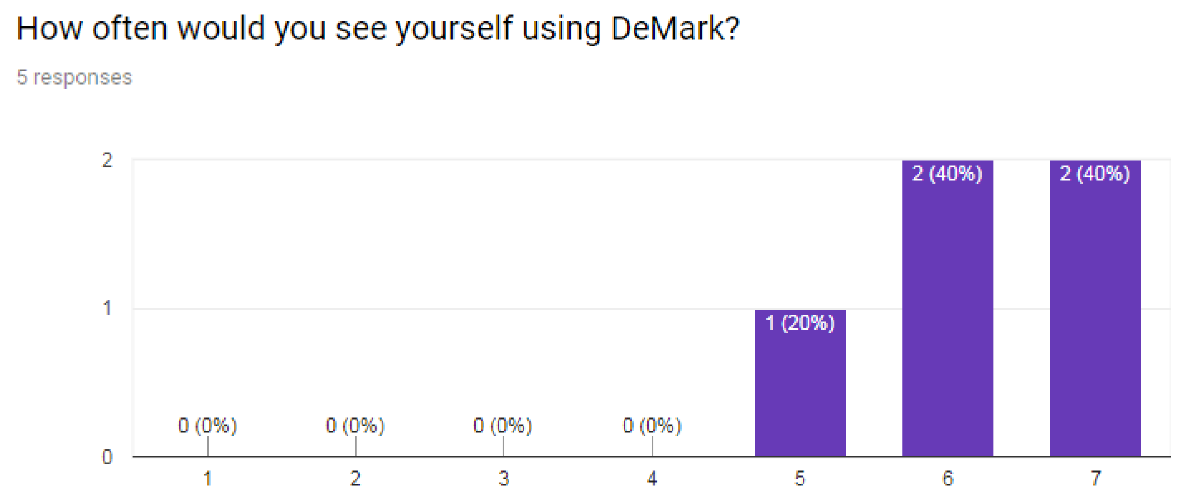


Figure 7: Question 3.a: 5 - "Slightly frequently", 6 - "Frequently", 7 - "Very frequently"

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 10.1-Q3.b).

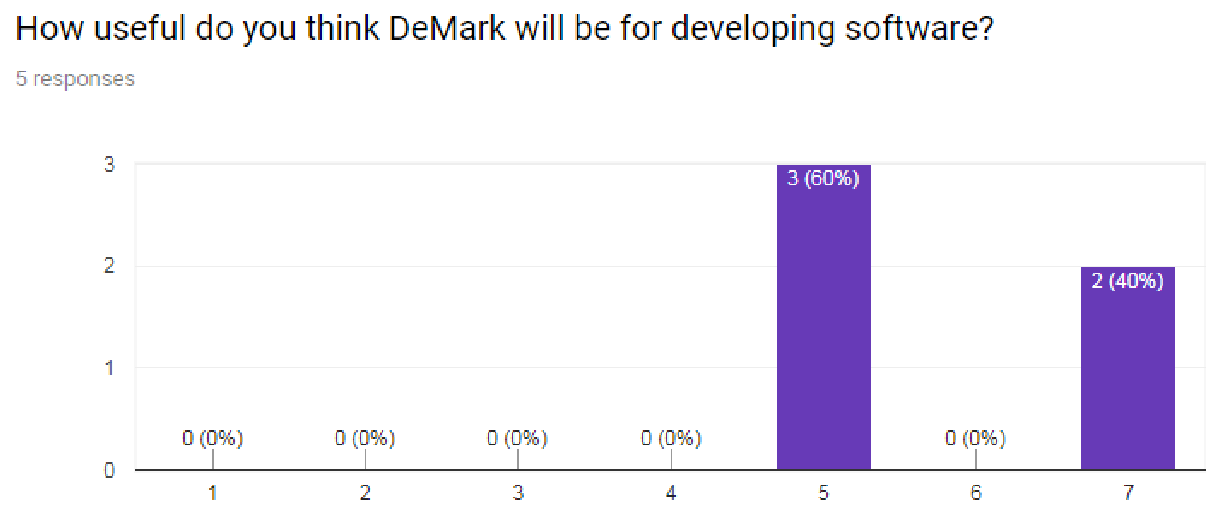


Figure 8: Question 3.b: 5 - "Slightly useful", 6 - "Useful", 7 - "Very useful"

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 10.1-Q3.c). This may be because currently the only functions that are available 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 10.1-Q3.d).

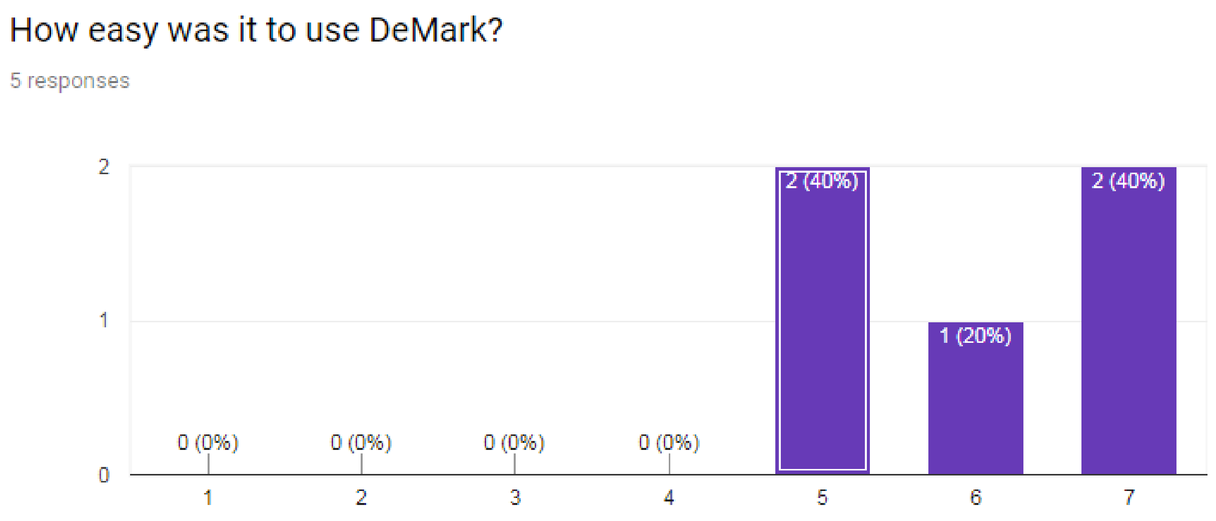


Figure 9: Question 3.c: 5 - "Slightly easy", 6 - "Easy", 7 - "Very easy"

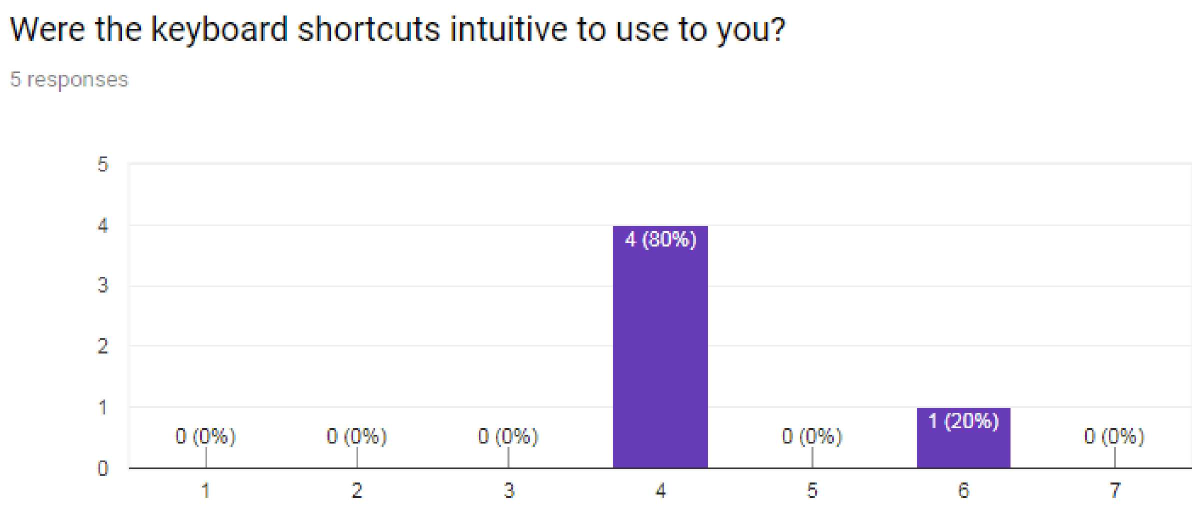


Figure 10: Question 3.d: 4 - "Not intuitive but not unintuitive", 5 - "Slightly intuitive", 6 - "Intuitive", 7 - "Very intuitive"

Currently, all of the default shortcuts starts with the `Alt` key. According to the feedback given after the intuitiveness question (appendix 10.1-Q3.e), starting the keyboard shortcut with the `Alt` can lead to some awkwardness in using the tool. The data shows that while the tool's functionality itself is simplistic and easy to use, activating the separate functions is still not well designed. To enable the tool provides users with the smoothest experience as possible, alternative keyboard shortcuts must be decided upon further in development of the DeMark. This introduces the challenge of picking shortcuts that will not interfere with IntelliJ's built-in shortcuts.

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 focus was determining whether or not a participant was able to install the plugin. This indicated that there is a compatibility issue with different versions of IntelliJ. This is an issue that needs to resolved by the time we start stage 2 of the experimentation.

Additionally, the entirety of section 4 in the survey (appendix 10.1-4) was focused on collecting each 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 10.1-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 10.1-Q4.b).

8.2 Testing: Branch and Path Coverage

As we have been focusing on developing the tool and manually testing it, our current test coverage is only 11% according to JaCoCo. These tests include different use cases for the mark action, including marking and unmarking a line on both one line of code, and multiple lines of code. The tests also covered parts of the highlighter utility and the Demark utility in our project. Note that this data is collected after we have excluded the `main.tests` packages from JaCoCo's reports, as well as all the `main.actions` package as actions are not invoked explicitly by the testing framework so JaCoCo cannot detect the coverage for them.

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". As stated in section 7.2, 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.

9 Schedule

We are breaking 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

- Each developer will familiarize themselves with the documentation layout.

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

- 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 De-Mark 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.

10 Total Hours Spent on this Assignment

We spent around 15 hours in total for this assignment.

References

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

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

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

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

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

Appendix

10.1 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