DeMark Project

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

May 2, 2018

1 Introduction

Currently on GitHub there are over 573,000 commits with the message "removed unnecessary prints" [1]. Writing print statements is the simplest way for a programmer to see the value of their variables and debug their code. However, these lines of code are not meant for production.

The major problem with printf-debugging is that the source code can become messy if the programmer accidentally commits them to the repository. For example, in one of Apache Qpid-Dispatch commits, they had to remove print statements from their repository because it wasn't necessary and didn't provide any additional value. The programmer had to take the extra time to go back and remove the print statements from the repository.

Having the source code flooded with print statements might also allow users of the software to see parts of the software in the console, which could lead to bigger security or comprehension issues.

Our solution to this problem is to provide programmers with a plugin 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 plugin tool will be used with the IntelliJ IDEA as 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 programmers to mark lines of code in their program through the IDE. Figure 1 illustrates a mock-up of the interface after the user has marked lines of code. When not in use the IDE should be relatively unchanged.

The tool should make the IDE reflect to the user that the lines are marked in a visual manner. For example, our idea is to highlight the marked lines of code in a different color. DeMark is designed to go beyond the simple marking of code. We want to provide functionalities that allows the user to easily manipulate the marked lines of code, such as being able to comment out and uncomment all the lines that are previously marked with a single action, being able to deleting all the marked lines at once, as well as being able to restore all the previously deleted marked lines.

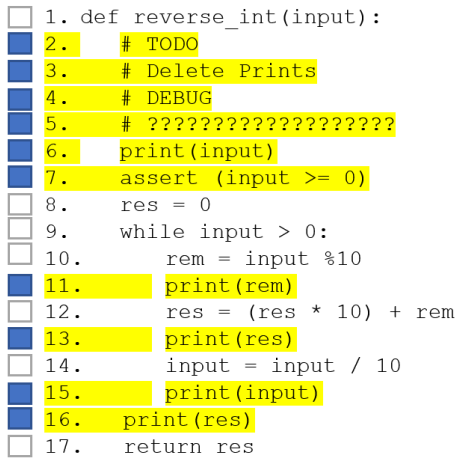

Figure 1: Before clearing marked lines

Figure 2 illustrates a mock-up of what the interface might look like after the user deletes highlighted lines of code. This will be represented as a button that is integrated into the IDE. Marking and deleting lines of code together are the most basic and core features of DeMark.

There are some additional functionality that we want to implement. A feature that we want to implement is allow users to undo the lines of code that they just cleared, essentially readding them to the code after they've been cleared. There should be some semblance of an action history where users can look what they have previously done and undo those actions if they desire. For example, lets say a user delete a marked line of code. The tool should communicate to the user that they took that action and allow them to undo the action.

A more advanced feature we were discussing was user profiles. We want users to be able to switch between different "profiles". Profiles can be think of as different colors, where users can mark and manipulate lines of each color independently. This allows the user to clear out chunks of marked coded instead of all of them at once, while still being able to mark the code. Visually DeMark should reflect the fact that there may be different profiles present. For example we could display a different color for each profile. The user should be able to switch between profiles seamlessly without much in terms of input lag. The tool should update its visual to reflect this.

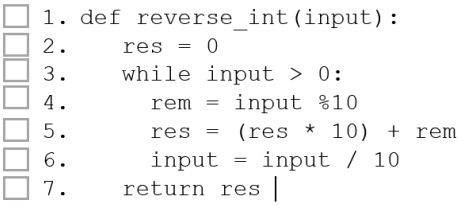

Figure 2: After clearing marked lines

Another feature that we will be providing will be keyboard shortcuts. Software tools usually provide a manner of keyboard shortcuts so that power users can quickly interact with the software without relying on the graphical user interface. We want to provide similar functionality for DeMark. Ideally users should be able to do all of the same functionality through keyboard commands if they choose to.

3 Existing Solutions and Their Limitations

Currently, there are several ways of dealing with finding bugs or understanding our program states. One way is by using logging levels and loggers [5]. An advantage of using a logging levels system is that you can have multiple levels of logging that can also give you information about your program states. It also provides away for you to log all the information that you want about the output. 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 programmer 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 that after the addition of the temporary, non-production lines of code, the developer can search and manually delete those lines. 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 users 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, as user will still need to go through the methods to comment or uncomment the temporary code manually.

This method, however, requires the user to do a small mental reconstruction of what their code actually does to decide whether or not to add specific lines of code. This could become problematic as the time between commits become bigger which makes the recall time longer for each hunk of code. Also, if the difference between a temporary code line and an actual line meant for production is not clear, this will also add some time to the recall. Because DeMark allows the user 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 the user to mark lines as they are typing or immediately after they are typing, and then later the user can delete the marked temporary code. DeMark is a tool that helps programmers 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 users to clear bookemarked lines, or clear the bookmarks completely. DeMarkwill also provide a toggle feature that allows the user 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 user to actually remove the marked lines from a file. DeMarkalso allows the user to undo their clear actions to restore their previously cleared marked lines. This can be useful in the case if the user 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, and as such, each of our components will be extending and using the tools or 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 user these actions are represented as visual elements through a drop down menu or keyboard shortcuts. Then, when the user clicks on "Mark" or enter the correct keyboard shortcut, the API will link the user's input with the appropriate action class,

which in this case will the Mark Action.

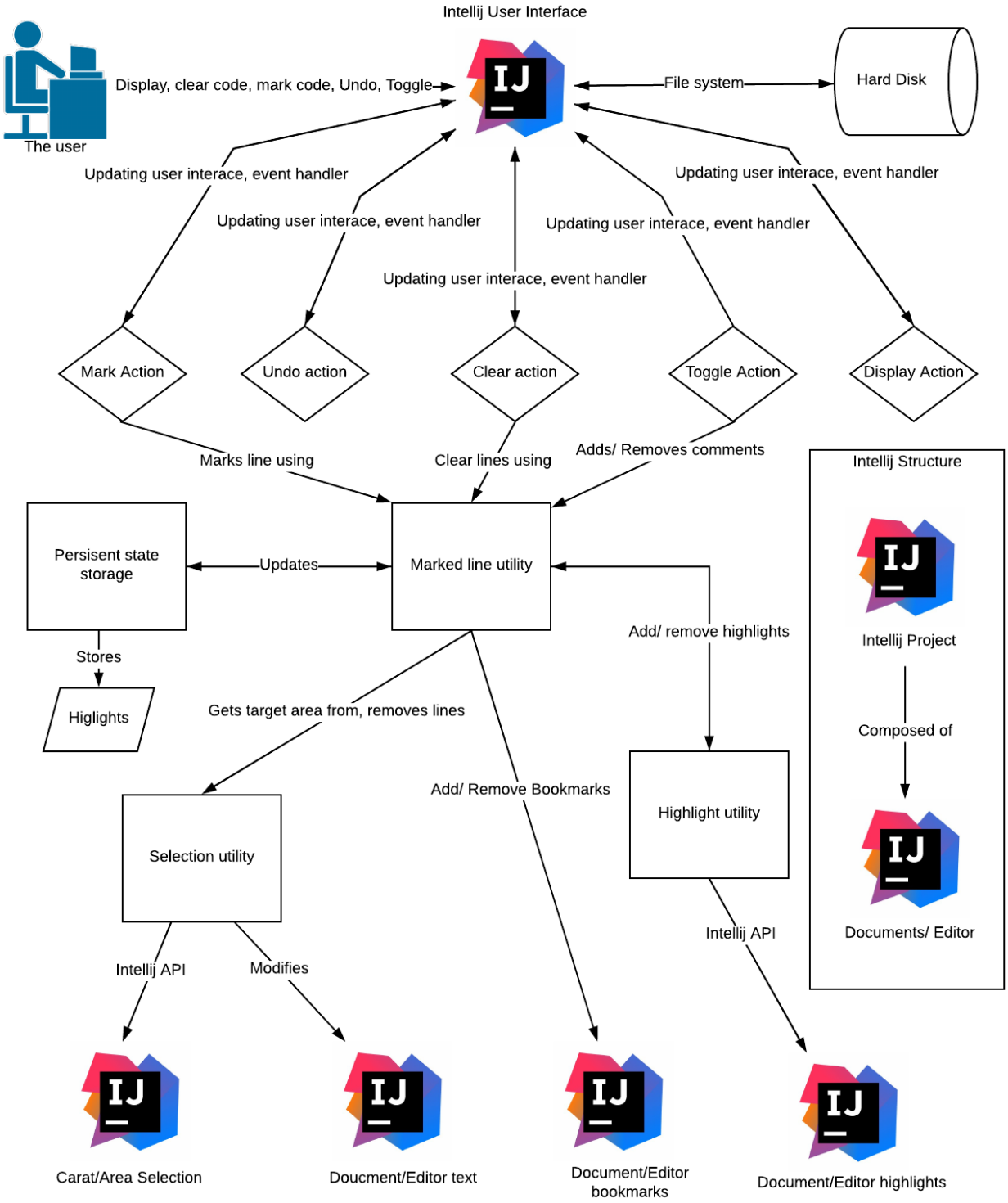

Figure 3: Architecture

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 means of collecting information from the IDEA itself. Mainly, the SDK provides ways to access the IntelliJ text editor and file 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 plugin to gain information on all files in a particular project.

To maintain a cohesive and modularized 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 make 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 user 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 for the user to clear 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.

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 system that we are extending is the IntelliJ platform. Specifically, we are building a plugin for the IntelliJ IDEA. The IntelliJ platform provides the infrastructure to build IDEs that are composable. This meant that the platform is "responsible for the creation of components, and the injection of dependencies into classes" [4]. This allows us to easily develop a plugin for IntelliJ using this platform.

5.2 Graphical User Interface Mockup

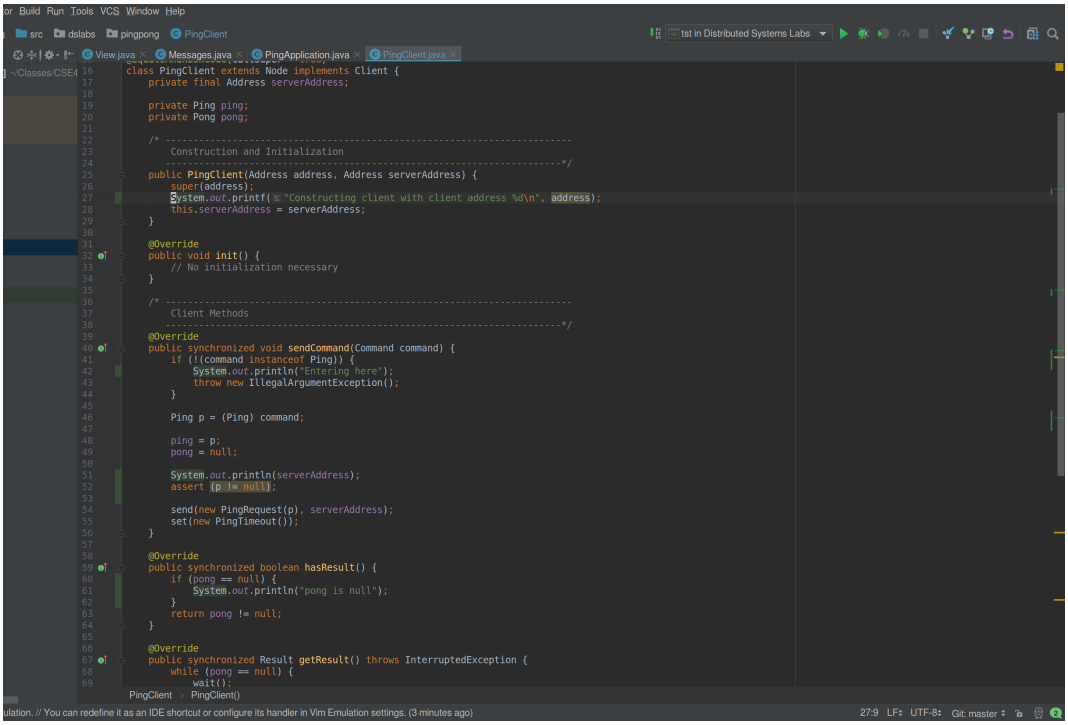

Figure 4: Original Interface

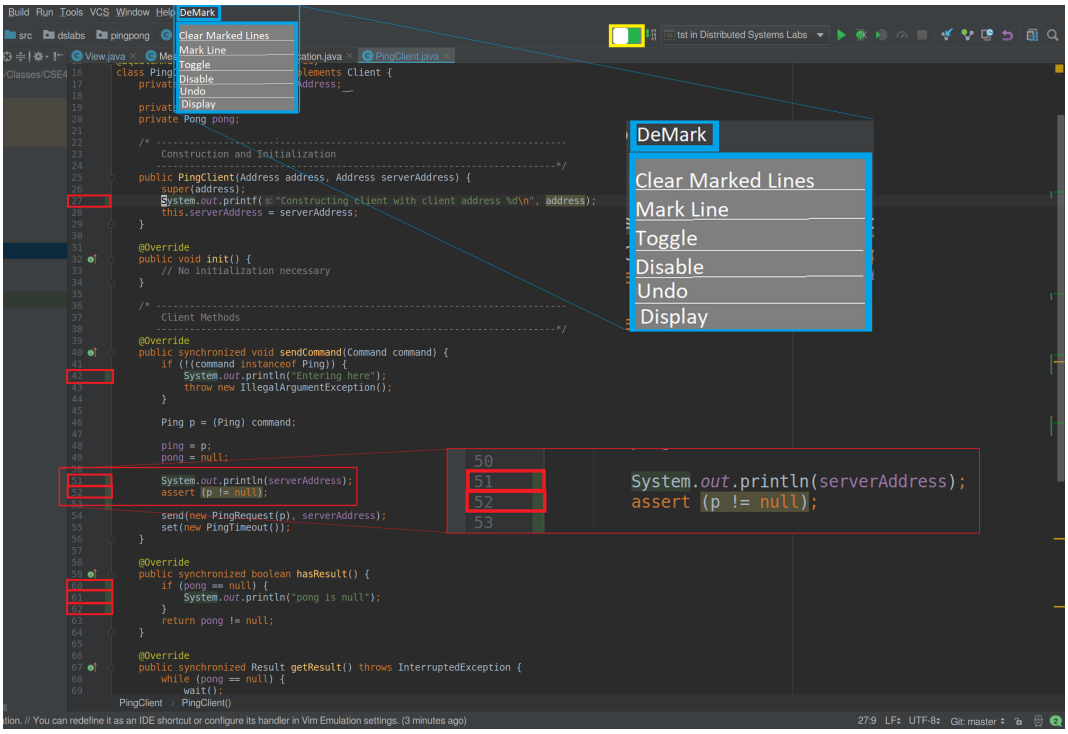

Figure 5: Plugin Installed

Figure 4 portraits the original IntelliJ display without our plugin installed. With the plugin installed, the user will be able to see all available options to them illustrated in figure 2 in the drop down menu, which is outlined in blue. Currently, the mockup shows only core functionality of DeMark and will have optional functionality once more features are available.

The section outlined in yellow in figure 5 is the toggle button. It allows the user the turn on and off the marked lines. If the toggle button is on, DeMark is enabled, and if it is off, the DeMark plugin is disabled, commenting all lines that were previously marked. While the toggle button is off, the user is allowed edit the commented marked lines, this includes insertion and deletion. The user will also be allowed to mark and unmark new lines. However, the user will not be able to use the clear and unclear functions.

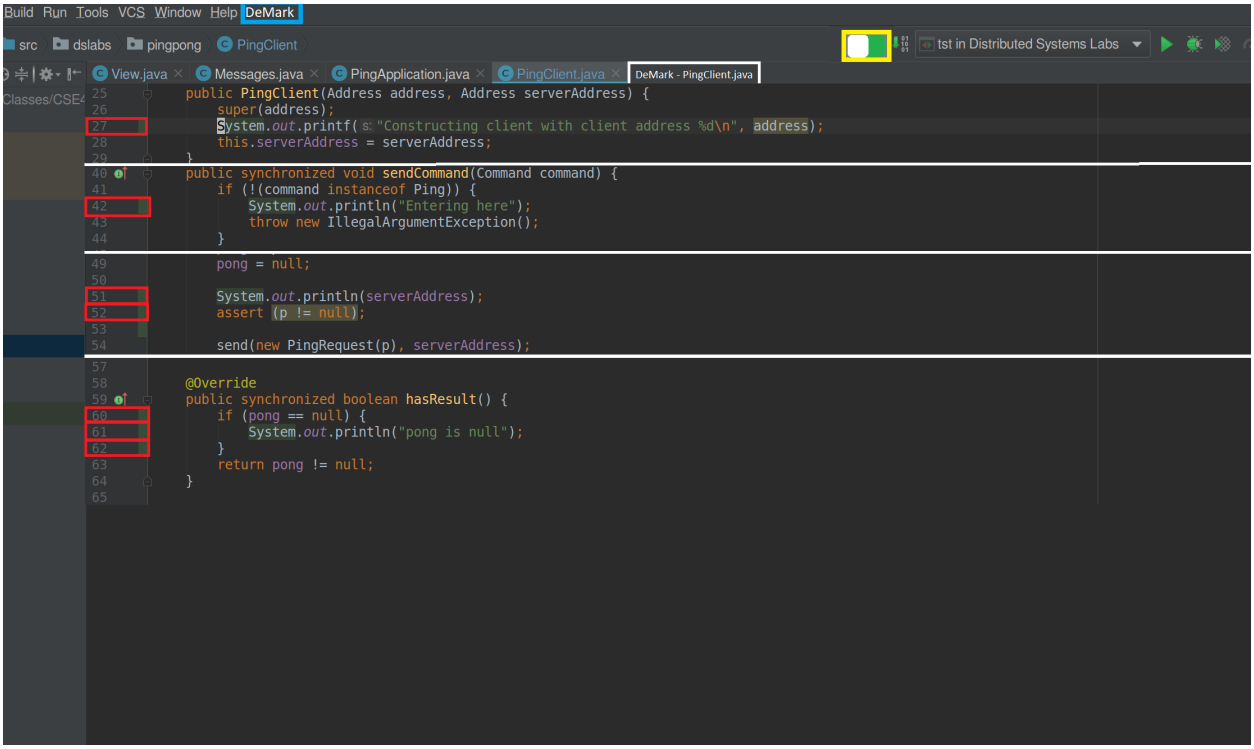

Figure 6: Display Function

The sections outlined in red represents the way DeMark is going to mark lines. The user will be 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 plugin 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 would like. Display mode allows the user to see their marked lines in their immediate context. The goal of this mode is to allow the user 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. This mode will also allow the user 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 user will not be able to make any changes to any lines while in display mode.

6 Technologies

The plugin we are developing is an user interface add on, which adds new interactive components and new functionalities to the IDE. Since we develop for the IntellijJ IDE, we will be using IntelliJ Platform Plugin SDK, which provides the most important APIs we need to build our plugin. We use gradle-intellij-plugin as the workflow to build the plugin. It takes care of the dependencies of our plugin, including the base IDE as well as other plugins that our plugin may depend on. [2] We will be using Java 8 for our development language since that is the language of the IntelliJ Platform.

7 Experimentation and Testing

One of the main goal of the DeMark plugin is to increase the 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 will conduct 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 plan receive from the experimentation will display our tool's contribution to the developing world.

7.1 User Trial Experimentation

We plan to have at least two stages of experimentation for DeMark. For each stage of experimentation, we will conduct a case study, where we will 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 experimenting will be distributing the alpha versions of DeMark 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 plan to provide preliminary instructions on how to use the tool. Data will be 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 will be provided to the users who are testing the alpha version of the tool as a way to measure our initial results. The data will aid the team in determine the stages of development for DeMark, specifically whether to move on to adding features and functionality. The biggest challenge in this stage would be the involvement of test users in reporting their experience as well as any bugs. As such, designing and implementing an accessible reporting system as well as error tracking system for DeMark will be essential to the success of this initial stage. This is the link to the survey for the first stage of experimenting: http://bit.ly/DeMarkPrelimSurvey.

The second stage of experimenting will be 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 prove to 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 for both of these experiments to provide impactful data, each stage of the experiment 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 plugin 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 Schedule

During our project, we will break down the development process by weeks. Within each week, there will be a main goal and with each main goal are sub goals to to keep us moving forward and 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. That said, we are currently on schedule in terms of development.

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

✓ **Week 3**: Begin plugin 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 plugin 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

- Experimentation framework set up for bug report and issue logging.
- User feedback framework set up.
- **Project Due**: Initial project result.

Week 8: User Trial Experimentation Stage 1

- Testing of the basic functionality among friends and peers.
- Collected and consolidated data and feedback from experiments.
- **Project Due**: Project presentation.

Week 9: User Trial Experimentation Stage 2

- 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.

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/.