DeMark Project

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

April 12, 2018

1 Motivation

The primary motivation is to provide a tool to facilitate the manipulation of temporary code. The team is planning to develop a plugin for the IntelliJ IDEA that will help streamline the addition, removal, and tracking of temporary code.

The motivation stems from our own experiences as programmers during the processes of debugging and experimenting with our own code. More often than not, when programming programmers want to see what the code is doing. Usually, the simplest way to do this is to write extra code such as print statements or if statements to see the values of variables or the overall flow of the program. These lines of code are usually there for us, the programmers, and are not meant for production. They are a way for us to debug and verify that our program is doing what we intended it to do. When programming, breaking down a problem into chunks is essential and making sure those chunks are correct along the way are important because if we wrote the code all at once and then debug, the program could crash immediately and we wouldn't know where the bug is. That is why when programming, it is usually suggested to review the code written by chunks and verify that it is correct in order to have less bugs to debug towards the end.

One major problem with this approach is that by the end of the coding or debugging session, the source code itself may be filled with lines of code that are there to improve our confidence in our code. For example, we may have programmed 5 lines of actual code but then 20 print statements to verify that the overall flow and variable values are what we expected. One specific example of this would be when trying to debug concurrent code. When the program has been tested enough and debugged to a level where it is able to push to the repository, the source code will be flooded with temporary code and it may be hard to distinguish what was intended for production and what was intended for debugging.

Our solution to this problem is to provide the programmer with a plugin tool to mark these lines of code and later delete them when necessary. The overall goal is to increase programming productivity and eliminate the time to search for unintended code. This plugin tool would be used with the programmers text editors that we support and so the programmer would have the freedom to use it at will. We decided to use a plugin because it would be much easier for the programmer to use if it was integrated with their text editor. This eliminates reading through complicated documentation when using a specific library that tries to solve a similar problem. Our goal of increasing programming productivity stems from the drive to maintain a repository that is both clean and readable while being as efficient as possible.

2 The Real World

Currently, there are several ways of dealing with finding bugs or understanding our program states. One way is by using logging levels and loggers [2]. 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 often requires the use of external libraries, integration into the source code, and they're often language specific.

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 extremely 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.

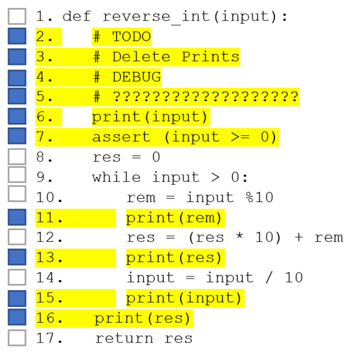
In terms of plugins, there is currently a plugin called LineOps that deal with marking lines and manipulating them. It allows the user to place a bookmark on lines that matches a certain string and perform actions like cut, copy, or delete on those bookmarked lines [4]. However, LineOps only allows users to clear marked lines, or clear the bookmarks completely. DeMark will also allow the user to toggle the displaying of marked lines. The toggle feature will be separate from the clear feature, which will allow the user to actually then clear the marked lines.

DeMark hopes to eliminate some of the disadvantages of these two methods while still providing an intuitive way of keeping track of temporary code.

3 Description

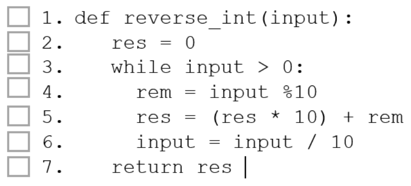
The core functionality is to provide the programmer the ability to mark lines of code in their program through the IDE. Figure 1 illustrates a mock-up of the interface after the user has marked lines of code. When not in use the IDE should be relatively unchanged.

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 functionality for the user to easily manipulate the line of code that they marked in easy and powerful manners. One of the core functionality for manipulating marked lines will be to delete marked lines. The user should be able to delete lines of code that they previously marked with one single action.



```
def reverse_int(input):
    # TODO
    # Delete Prints
    # DEBUG
    # ???????????????????
    print(input)
    assert (input >= 0)
    res = 0
    while input > 0:
        rem = input %10
        print(rem)
        res = (res * 10) + rem
        print(res)
        input = input / 10
        print(input)
    print(res)
    return res
```

Figure 1: Before clearing marked lines



```
def reverse_int(input):
    res = 0
    while input > 0:
      rem = input %10
      res = (res * 10) + rem
      input = input / 10
    return res
```

Figure 2: After 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 most likely 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 thus it is essential that we implement them in some manner.

There are some additional functionalities that we want to implement. An advanced feature that we want to implement is allow users to undo actions that they perform. 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.

Additionally we want the user 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. 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.

Another feature that would be nice is to provide the user with 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.

4 Architecture Diagram

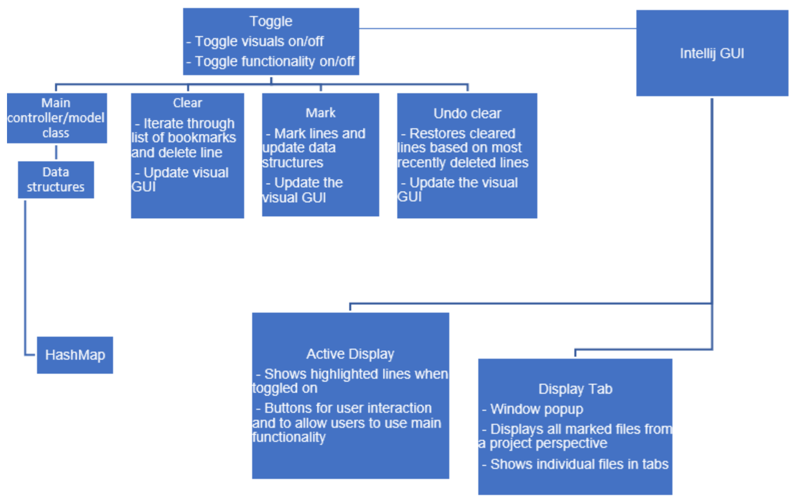

Toggle
- Toggle visuals on/off
- Toggle functionality on/off
Intellij GUI
Main controller/model class
Data structures
Clear
- Iterate through list of bookmarks and delete line
- Update visual GUI
Mark
- Mark lines and update data structures
- Update the visual GUI
Undo clear
- Restores cleared lines based on most recently deleted lines
- Update the visual GUI
HashMap
Active Display
- Shows highlighted lines when toggled on
- Buttons for user interaction and to allow users to use main functionality
Display Tab
- Window popup
- Displays all marked files from a project perspective
- Shows individual files in tabs

Figure 3: Architecture

Our plugin will consist of two major components. One major component will be a class that acts as the model and controller for our program. At the highest level is a toggle function that will essentially allow the user to turn on and off the functionality and visuals of our plugin. When the user toggles off our plugin, it stores the lines that are marked as well as hiding them. The plugin also disallows any further marking/clearing when the toggle is off. To do all of our bookkeeping we are going to use a main class that represents our model that will store all the data structures (in our case a HashMap) and that it’s structures will be used in our core functionality. Next to the main class are the core functionality that we want to implement in our program. We want the user to be able to mark lines, when this happens the program should update the data structures and visuals. We want the user to able to clear lines, when this happens the program should update the data structures by deleting relevant lines and updating visually to reflect any changes. We also want the user to be able to undo clears, which is essentially a reversal of the last actions that the user committed. The model class will interact with the actions by passing its data structures to the actions for the actions to use. All the actions and model depend on the toggle.

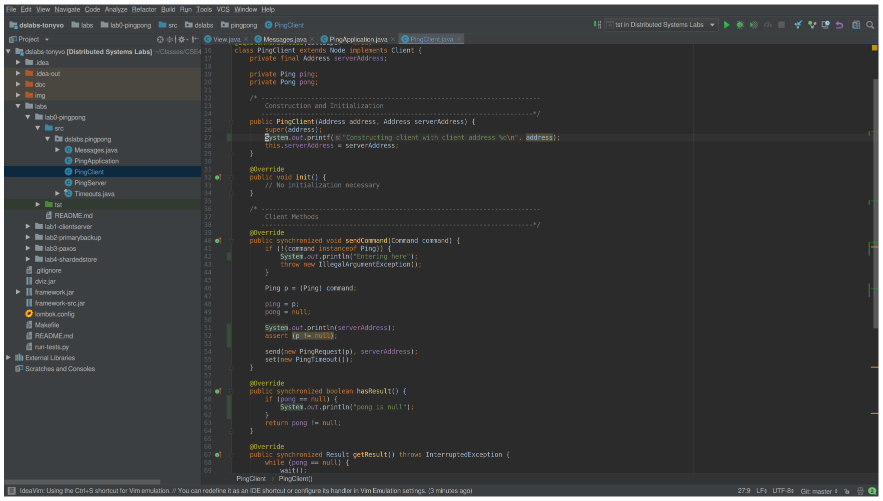
Another major aspect of our plugin is extending the IntelliJ GUI. For our plugin there will be essentially two types of visual interfaces. The first type includes visual aspects that are always going to be present in the GUI, called active displays in our diagram. This is represented by the highlighted lines and the buttons that we add to allow the user to interface with our plugin. We are designing our buttons to be a drop down menu where the user can click buttons to perform the actions that we want to implement. The details of how this will look like will be further discussed in Section 5.2. The other side of the extension of the Intellij GUI is a display tab. This will allow the user to view all marked lines in files from a more project overview level. We want this to be a window consisting of tabs where each tab is a marked file. Each tab should display the file and what lines are marked along with a range of lines above and below to provide context.

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" [3]. This allows us to easily develop a plugin for IntelliJ using this platform.

5.2 Graphical User Interface Mockup



```java
class PingClient extends Node implements Client {
    private final Address serverAddress;

    private Ping ping;
    private Pong pong;

    /* -------------------------------------------------------------------------
        Construction and Initialization
       -----------------------------------------------------------------------*/
    public PingClient(Address address, Address serverAddress) {
        super(address);
        System.out.printf( s: "Constructing client with client address %d\n", address);
        this.serverAddress = serverAddress;
    }

    @Override
    public void init() {
        // No initialization necessary
    }

    /* -------------------------------------------------------------------------
        Client Methods
       -----------------------------------------------------------------------*/
    @Override
    public synchronized void sendCommand(Command command) {
        if (!(command instanceof Ping)) {
            System.out.println("Entering here");
            throw new IllegalArgumentException();
        }

        Ping p = (Ping) command;

        ping = p;
        pong = null;

        System.out.println(serverAddress);
        assert (p != null);

        send(new PingRequest(p), serverAddress);
        set(new PingTimeout());
    }

    @Override
    public synchronized boolean hasResult() {
        if (pong == null) {
            System.out.println("pong is null");
        }
        return pong != null;
    }

    @Override
    public synchronized Result getResult() throws InterruptedException {
        while (pong == null) {
            wait();
```

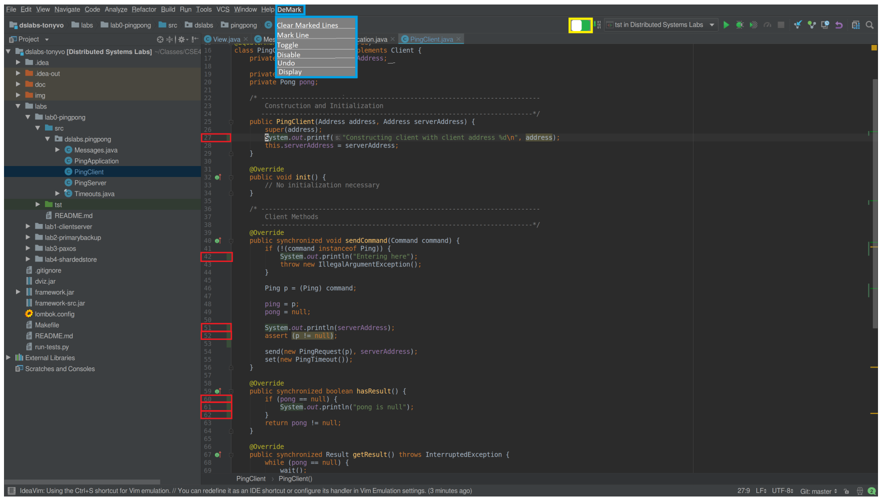
```
IdeaVim: Using the Ctrl+S shortcut for Vim emulation. // You can redefine it as an IDE shortcut or configure its handler in Vim Emulation settings. (3 minutes ago)
```

Figure 4: Original Interface

Figure 4 is what the original IntelliJ interface looks like without our plugin installed. After the user has installed our plugin, the interface will look like figure 5.

With the plugin installed, the user will be able to see all available options 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, removing all lines that were previously marked.



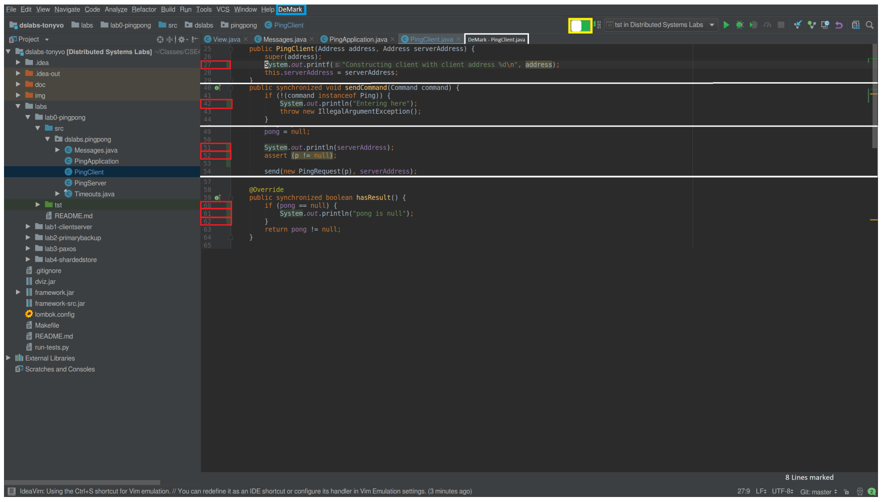
```java
class PingC                  plements Client {
    private                  Address;

    private
    private Pong pong;

    /* -------------------------------------------------------------------
        Construction and Initialization
       -----------------------------------------------------------------*/
    public PingClient(Address address, Address serverAddress) {
        super(address);
        System.out.printf( s: "Constructing client with client address %d\n", address);
        this.serverAddress = serverAddress;
    }

    @Override
    public void init() {
        // No initialization necessary
    }

    /* -------------------------------------------------------------------
        Client Methods
       -----------------------------------------------------------------*/
    @Override
    public synchronized void sendCommand(Command command) {
        if (!(command instanceof Ping)) {
            System.out.println("Entering here");
            throw new IllegalArgumentException();
        }

        Ping p = (Ping) command;

        ping = p;
        pong = null;

        System.out.println(serverAddress);
        assert (p != null);

        send(new PingRequest(p), serverAddress);
        set(new PingTimeout());
    }

    @Override
    public synchronized boolean hasResult() {
        if (pong == null) {
            System.out.println("pong is null");
        }
        return pong != null;
    }

    @Override
    public synchronized Result getResult() throws InterruptedException {
        while (pong == null) {
            wait();
```

IdeaVim: Using the Ctrl+S shortcut for Vim emulation. // You can redefine it as an IDE shortcut or configure its handler in Vim Emulation settings. (3 minutes ago)

Figure 5: Plugin Installed

The sections outlined in red represents the way DeMark is going to mark lines. Ideally, 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 "display" function creates tabs that are in display mode. Figure 6 shows what it would look like:



```java
    public PingClient(Address address, Address serverAddress) {
        super(address);
        System.out.printf( s: "Constructing client with client address %d\n", address);
        this.serverAddress = serverAddress;
    }
    public synchronized void sendCommand(Command command) {
        if (!(command instanceof Ping)) {
            System.out.println("Entering here");
            throw new IllegalArgumentException();
        }
        pong = null;

        System.out.println(serverAddress);
        assert (p != null);

        send(new PingRequest(p), serverAddress);

    @Override
    public synchronized boolean hasResult() {
        if (pong == null) {
            System.out.println("pong is null");
        }
        return pong != null;
    }
```

8 Lines marked

Figure 6: Display Function

Display mode allows the user to see their marked lines in their immediate context. The goal is that this mode will allow the user to be able to focus on codes that are closer to their suspected bugs, allowing them to better reason through their code. 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.

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. [1] We will be using Java 8 for our development langauge since that is the language of the Intellij Platform.

7 Experimentation and Testing

One of the main goal of the DeMark plugin is to significantly 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, and possibly user profiles.

7.1 User Trial Experimentation

We plan to have at least two stages of experimentation for DeMark. 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 only core functionality 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. 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.

The second stage of experimenting will be distributing an improved beta version of DeMark to a larger set of developers. This version will also have the core functionality along with several added features. 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.

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.

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

[2] *Simple logging facade for java (slf4j).* Available at `https://www.slf4j.org/`.

[3] *Intellij platform sdk guide*, Mar 2018. Available at `http://www.jetbrains.org/intellij/sdk/docs/intro/intellij\_platform.html`.

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