IDE-IT aka Integrated Development Environments - Intelligent Tutorials
Backend Team

Motivation

Integrated Development Environments (IDEs) are software applications that provide tools and facilities for developers to write and test software. They are designed to be beneficial to developers by providing features like static code analysis as well as automating some common tasks that the developer would otherwise have to do manually. However, those features are essentially worthless if the user does not know they exist. A number of studies have shown that the discoverability of existing IDE tools and plugins is a common issue. Some survey results suggest that developers often use only a small number of IDE functionalities out of the total set available [3]. As an example, a collection of Eclipse usage data from over 120,000 users in May of 2009 found that 88% of users do not use the "Open Resource" command, despite its praise in several online blogs [7]. Another study found that developers may not use tools and features built into their IDEs because they are unaware of the variety of tools offered within the IDEs, find the tools to be too complex, or the tools are not easily accessible [1]. Likewise, developers find that many tools in their IDEs are not trivial to configure, and this prevents them from using the tool at all [2]. In one case study, developers reported that sometimes it is difficult just to get to the menu where the options for configuring a particular feature are, and developers shared stories where they could not figure out how to customize a tool and ended up having to search the web to find out where the tool's preferences were [2]. Popular IDEs have vast plug-in ecosystems which offer rich rewards, but only for developers who know how to find these "gems".

We propose a tool which improves the discoverability of built-in features of an IDE, specifically for the Eclipse IDE. IDE Intelligent Tutorials (IDE-IT) will be an Eclipse IDE plugin that will detect when users are not taking advantage of the many features it includes and provide them with a notification or tutorial to make them aware of those features that are relevant to what they are currently doing. Our project will create a backend service for IDE-IT, which will consist of tracking user inputs and changes made in an Eclipse document editor, and evaluating those inputs and changes to determine if Eclipse's features are being neglected. A simple interface for our plugin will allow a front end plugin developer to easily access the information calculated by our service, and determine how they would like to present the information to the user.

The key difference between previous approaches to this problem and IDE-IT is that rather than teaching users more about features they already use, having the user spend their time navigating tutorials, or relying on random daily "tips" to increase user awareness of features, IDE-IT will teach users about the existence of features that are actually relevant to the way they use Eclipse. IDE-IT will continuously track user action such as document changes, key presses, and mouse clicks, and report in real-time when it determines that features are not being utilized. Through non-invasive notifications when users neglect to use relevant features, we provide a simple reminder that allows them to understand how they can make their work easier without interrupting it.

Related Works

As a whole, there are a few existing solutions that try to address a similar problem that IDE-IT does of informing a user of existing features in different IDEs. IDEA Feature Suggester is a plugin for Intellij that gives pop up suggestions for IDE features in real-time based on user input [4]. Intellij also has a built-in Features Trainer mode that provides interactive tutorials for users [5]. While this can be helpful, the user still needs to seek out this knowledge and sit through a tutorial to get some information. As another example, MouseFeed is a plugin for Eclipse that seeks to teach users about Eclipse's hotkeys for features by reminding them with a popup notification containing the hotkey for a given feature any time they use a feature via the toolbar or menu [6].

As for comparing these related works to the backend aspect of IDE-IT, The IDEA Feature Suggester monitors changes within an open Java document and constantly evaluates them to see if the user could benefit from a feature of the IDE. Though the premise is very similar to that of the IDE-IT backend, IDEA Feature Suggester is rather buggy, and often produces false positive warnings. It managed to break almost immediately when we tested it, and stopped evaluating our input. In addition, IDEA Feature Suggester is a complete plugin for the end-user of IntelliJ with its own frontend, and it does not allow for a customizable frontend interface as the IDE-IT backend does. The Features Trainer mode of IntelliJ does not actively track user input, and instead provides a static library of interactive tutorials in which the user can learn about different features of the IDE. Thus, real-time evaluations of useful features for the user are not calculated as the user types as they are in the IDE-IT backend. Mousefeed only detects user mouse click action within the toolbar or menu of Eclipse to see if the user activated a feature that way. As such, it does not detect features that are being neglected, but instead features that are already being used. This works great if a user already knows that a feature exists, but falls short as a full solution as it requires the user to be aware that a feature exists in the first place.

Murphy-Hill et. al describe in their report on recommendation of development environment commands that there are other existing solutions that attempt to address the issue of IDE feature discoverability [7]. IDE documentation can provide a listing of available features, but this also requires the user to proactively search for such information; if the user is not aware they are missing out on any tool functionality, then they may not even know what to search for in the first place. Additionally, most IDEs have some form of tips, usually tips of the day, that try to convey some of their features to the user. However, those tips can be irrelevant to the user's goals, are randomly selected, and/or are more of an annoyance than a help. Other tools in Eclipse slightly similar to this proposal are the "content assist" and "parameter hints". However, these are mostly composed of suggestions in auto completion, variable names, and parameters. *Over the shoulder learning* can also provide users with information about IDE features, where pair programming at a computer allows a peer to notify a user that a feature exists [8]. None of these options address the issue of a developer not knowing the issue at hand, and thus not being aware of what keywords to search, settings to turn on, or tools to enable. The report by Murphy-Hill et. al includes much information about a possible system for recommending IDE features to users, but this system relies on evaluating the user's command usage history to predict undiscovered commands they may be interested in, rather than evaluating the actual text input and document changes themselves as IDE-IT does.

Approach

We are implementing a backend service for IDE-IT. Responsibilities in doing so include monitoring and evaluating the user's input and then notifying any registered observers attached to our service. We plan to track mouse input, keyboard input, content of document changes, and abstract syntax tree (AST) changes. When document changes are detected, information from these inputs will be passed to a series of evaluation functions. Each IDE feature that IDE-IT supports will have its own evaluation function, which will check for a specific sequence of user actions and/or document changes that signal that the user has neglected to use the feature. The IDE features we will focus on are features that we find to facilitate tasks that are most commonly performed when using Eclipse to write Java code.

We will use the terms features, evaluations, and triggers liberally throughout this document. The following are the definitions of these terms in the context of this project.

Feature
: A task, hot key, or menu option that Eclipse provides to the user. Most features are tasks that Eclipse can automate and save the developer time, but only if the developer is aware of it and knows how to use it. IDE-IT will focus on IDE features that we believe are beneficial to users, but can be easily overlooked.

Evaluation
: A function that checks for a specific sequence of user action and/or document changes. This specific sequence of user actions / document changes represents how a user would act when trying complete a task manually instead of using the built in functionality of Eclipse.

Trigger
: The term trigger is used for when a feature's evaluation function requirements are met. When this occurs, the front end will be notified via an interface as described below.

The following list of features are currently prioritized by the IDE-IT team to be implemented:

1. Block commenting
2. Adding import statements using "shift-cmd-o"
3. Removing unnecessary/unused import statements using "shift-cmd-o"

4. Correcting indentation
5. Refactor code base by renaming a variable throughout the entire project

Note that while 2 and 3 above seem quite similar, they are distinct features, and will require different evaluation to determine whether the user would benefit from one or the other. This list of features was chosen and prioritized based on not only the frequent performing of the tasks they automate and their relevance to all Java projects, but also on the estimated feasibility of implementing the evaluations to detect if the user is circumventing these features.

To detect user input, we will have listeners for document changes within each editor window, including listeners for AST changes as the user works in Eclipse. The listeners will provide the change data to the evaluation functions, and the evaluation functions will be responsible for checking a set sequence of user action and/or document changes against the data provided by the listeners. For example, one of of the evaluation functions will check for when multiple sequential lines of code are commented out, by keeping track of what characters were added, and in what order, line, and line offset in the document. If the user entered double forward slashes at the start of each of two adjacent lines, the evaluator would recognize that sequence of actions and trigger a feature suggestion for block commenting. Murphy-Hill et. al describe this approach to solving the problem as “Inefficiency-based Recommendation”, where inefficient user activity is used as the basis for providing recommendations for IDE features [7].

For the scope of this project, our team is focusing entirely on the backend development. In order to ensure good encapsulation and minimal coupling, we will provide an interface for other plugins to use. This was designed specifically with the frontend team of IDE-IT, but is general enough that any plugin could take advantage of the interface. This allows other plugins to use our listeners, evaluations, and notifications as a service. The interface is set up as an Listener and Observer model, where the observers are notified whenever an evaluation function triggers. This interface is described in more detail below.

Architecture

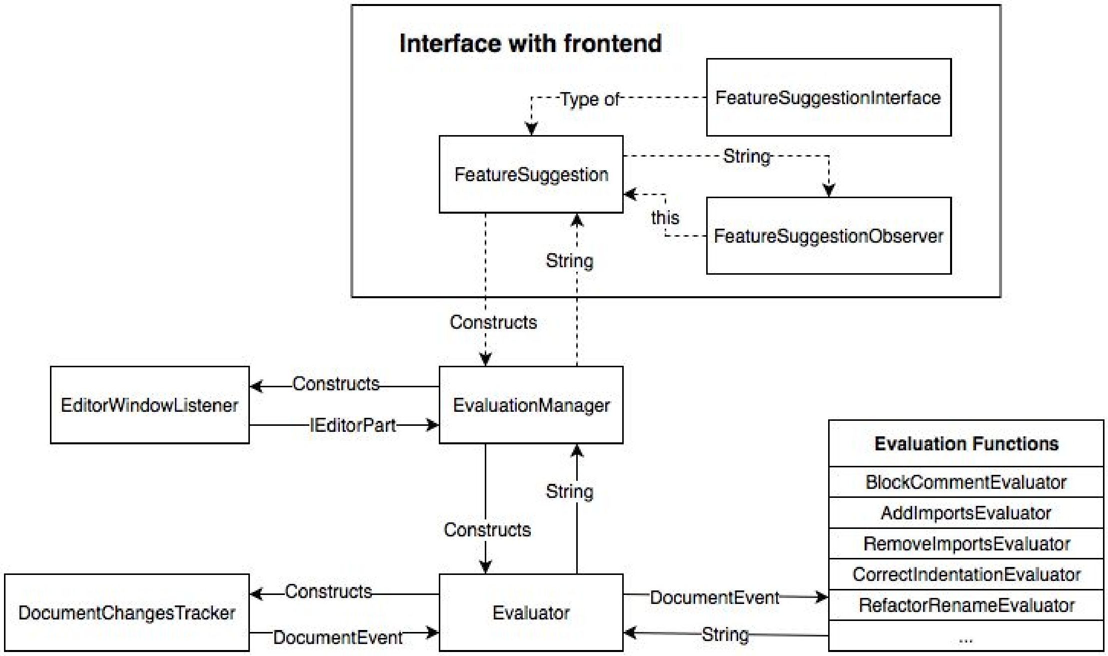


Figure 1: Diagram of internal architecture

The following is a description of the modules in the diagram above:

- FeatureSuggestionObserver
    - An abstract class designed to be extended by a frontend client. Contains a notify(String) method, which will be called when a feature evaluation has been triggered, and the String parameter will contain a unique ID of the given feature.
- FeatureSuggestion
    - An implementation of the FeatureSuggestionInterface interface. This is the main object that frontend clients will use to manage their interaction with our backend service. Once the FeatureSuggestion is created, clients can register any number of their own FeatureSuggestionObserver objects with the FeatureSuggestion object to be notified when a user has neglected a feature evaluation has been triggered.
- EvaluatorManager
    - Created when a frontend client first calls the start() method of the FeatureSuggestion object. The EvaluatorManager assigns Evaluators to document editor windows, keeps track of all active Evaluators that have been assigned to document editor windows, and handles reporting triggered features from each Evaluator to the FeatureSuggestion. This ensures that all triggered feature reports notify the same FeatureSuggestion.
- EditorWindowListener
    - An extension of IPartListener2 from the Eclipse Plugin API, which fires off events based on users' navigation through the Eclipse workspace. Created and added to

the list of Eclipse workspace listeners when the EvaluatorManager is constructed. Listens for activation of document editor windows (i.e. when a document editor window or opened, or its tab is switched to). When that occurs, the EditorWindowListener will notify the EvaluatorManager to assign an Evaluator to the given document editor window.

- Evaluator
  - Responsible for evaluating document changes detected within a single document editor window. When document changes are detected, the Evaluator will cycle through each feature evaluation function, passing the document change event information. If a feature evaluation function returns true (indicating that the user has neglected to use the respective feature), it will notify the EvaluatorManager with the unique ID string of the feature that was triggered.
- DocumentChangesTracker
  - An extension of IDocumentListener from the Eclipse Plugin API. Created when a new Evaluator is assigned to a document editor window. Responsible for listening for changes made within that document editor window. When changes are detected, the DocumentChangesTracker will pass the change information back to the Evaluator.
- Evaluation Functions
  - A set of classes extending an EvaluationFunction interface. Each feature that IDE-IT evaluates will have its own class extending EvaluationFunction, which contains an evaluate() function that is called when an evaluator receives the signal that a document change has occurred.

In addition, the backend service for IDE-IT will provide the following interface. This will allow other plugins (the frontend for IDE-IT, for instance) to use our plugin as a dependency by creating their own class that extends our FeatureSuggestionObserver class and register itself with our FeatureSuggestionInterface. The main form of communication sent through the FeatureSuggestionObserver is a unique featureID String representing the feature that was neglected by the user (the list of available featureIDs will be discoverable through the FeatureSuggestionInterface->getAllFeatureIDs() method as noted below)

FeatureSuggestionObserver

void notify(String featureID)

All observers will be notified with the featureID of an Eclipse feature when an evaluation function triggers.

*FeatureSuggestionInterfac***e**

boolean registerObserver(FeatureSuggestionObserver obs)

Register an observer with our service to be updated when any evaluation functions trigger.

boolean removeObserver(FeatureSuggestionObserver obs)

Removes a registered observer with our service.

void start()

Starts the backend service. Observers will receive notifications when they are triggered

void stop()

Stops the backend service. All listeners created by the backend service will be removed, and observers will no longer receive notifications.

boolean isRunning()

Provides a check to see the current state of the backend service

List<String> getAllFeatureIDs()

Provides a list of all featureIDs. This is designed to be used so frontend services can check against this list for accuracy and verification.

In communicating with the team developing the frontend plugin for IDE-IT, we have defined the following featureIDs thus far:

- blockCommentSuggestion
    - Indicates the user has manually commented out multiple adjacent lines
- addImportStatementsSuggestion
    - Indicates the user has manually added import statements for unresolved classes within the document
- removeUnusedImportsSuggestion
    - Indicates the user has manually removed unused import statements within the document
- correctIndentationsSuggestion
    - Indicates the user has manually corrected line indentations, rather than using the automatic indentation correction feature
- variableRenameRefactorSuggestion
    - Indicates the user has manually renamed a variable in several points throughout the AST, rather than using the Refactor->Rename feature

We plan on providing the list of featureIDs in a file accessible by developers who wish to use our backend service, so they can implement support for those feature notifications.

Inputs from the user's document changes (including AST changes) will be detected by listeners. These listeners will be the means of interfacing directly with eclipse and will act as the first measure to detect user interaction. Note that both primary and secondary changes are tracked here because simple changes can be easier to detect through primary inputs, while more complicated ones may not be possible at all to detect through listeners (e.g. the user using some combination of other plugins and/or shortcuts to make the document changes.) These listeners will then pass their input to the feature evaluators which, given the document changes, will determine if an intelligent tutorial or notification is relevant to the user's actions. Finally, the evaluator will send any successful evaluations to the front end interface to signal that a tutorial or notification can potentially be triggered.

For the features we have implemented evaluation for thus far, our algorithms to detect whether a feature has been triggered are as follows:

- Block comment evaluator
    - Listens for changes made to the document
    - Determines whether a document change has commented out a line that was not already commented out by checking that the line before the change minus whitespace did not start with "//", but the line after the change minus whitespace does
    - Keeps track of the last line that was commented out from a document change and when that change occurred
    - If a document change comments a line out, and the last line that was commented out is adjacent to the line, and the two changes occurred farther than 100ms apart, this feature evaluator is triggered (returns true)
        - This 100ms threshold accounts for the case where the user actually does use the block comment feature. When using this feature, Eclipse comments the lines out one-by-one under the hood (almost instantly). If we did not include this threshold, our feature evaluator would falsely trigger many times if a user used the block comment feature.
- Add import statements evaluator
    - Listens for changes to the annotations in the document (for example, in Eclipse, when a user references an unresolved/unimported type, an annotation is added to the document window warning the user about the unresolved type)
    - When changes to the annotation model are made, a boolean that keeps track of whether there are any existing unresolved type annotations is updated
    - Tracks the last two additions made to the document. If they are "t ", checks to see if the preceding word is "import". If it is, and there are currently existing unresolved type annotations, this feature evaluator is triggered (returns true)
- Remove import statements evaluator
    - Listens for changes to the annotations in the document
    - When changes to the annotation model are made, if there are any existing "the import is never used" annotations, this feature evaluation is triggered (returns true)

We will use various Eclipse APIs to help with detecting user input, parsing AST, managing plugin lifetime, and displaying the tutorials for the features themselves. The main API that we will be using is the Eclipse PDE (Plug-in Development Environment) which is used for developing, testing, and debugging eclipse plugins. We will also be taking advantage of the Eclipse 4.x SDK and its underlying API. In addition to the APIs we are using, the release will eventually involve packaging the plugin in Eclipse's standard plugin packaging system and potentially making it available in tandem with the front end on the Eclipse marketplace, an online package repository for Eclipse.

Project Evaluation

Our goal with this plugin is to accurately interpret user input to identify when a user could benefit from a built in IDE feature. To evaluate the success of our plugin, we will evaluate our success on each individual feature that we evaluate for. To do this, for each feature we will instruct a few developers (probably other students) to find every way they can to circumvent using that feature. For example, we would ask them to comment out multiple lines of code, but pretend that they don't know the block commenting feature exists. By observing all the ways that our subjects circumvent each feature we can obtain a set of cases (ways of circumventing the feature) for each feature that we plan to cover in the feature's evaluation algorithm. Concrete measures of success for each feature can then be calculated by comparing the number of cases of the feature that we successfully cover with the total number of cases for that feature. Ideally, we will cover each feature 100% (all cases considered).

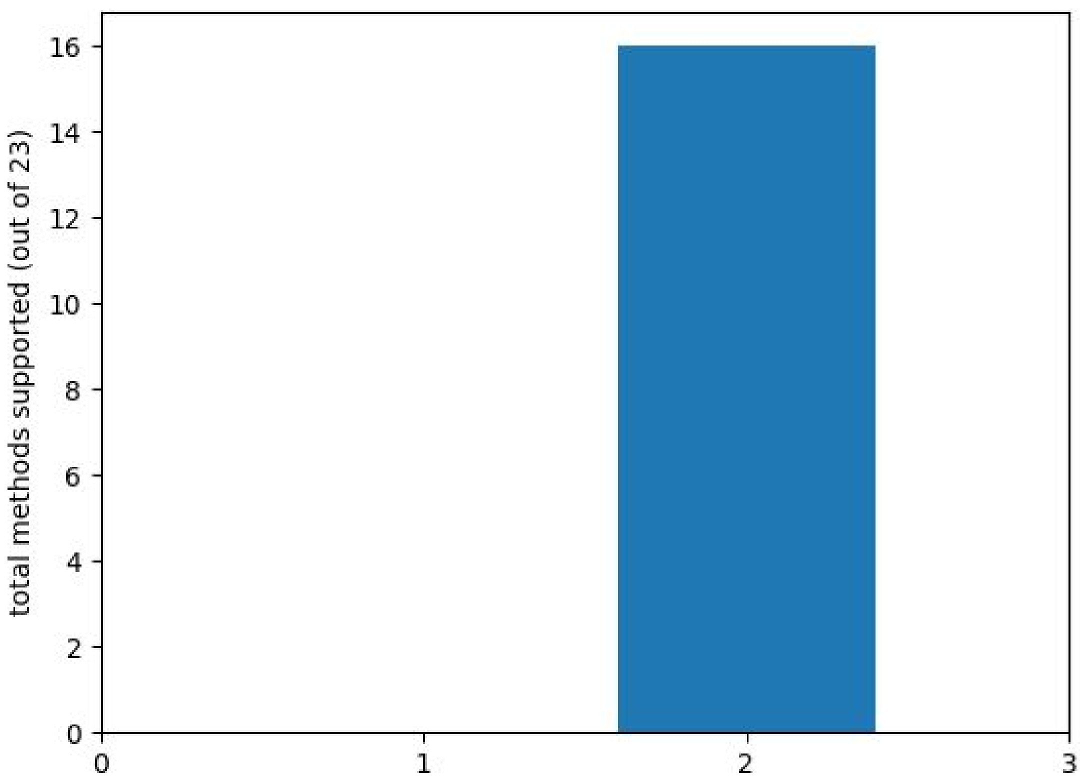
The main focus of this plugin is to increase feature discoverability. To that end, we will err on the side of having minimal false negative suggestions if it means having more false positive suggestions. False negative suggestions in this context is if a user should have a feature suggested to them, but our plugin does not suggest it. A false positive is if the user is suggested a feature that is irrelevant to their current actions. We can test for both false negatives and false positives through user testing and quantify both. Ideally we want minimal amounts of both, but we will focus on minimizing false negatives as our priority.

Initial Results

After composing the list of user actions that can circumvent features (as described above), we translated those user actions into tests. These tests mock user input to perform the sequence of actions previously determined as a way to circumvent a feature. Some of these sequence of actions seem more reasonable than others, but we have tried to take a wide variety of actions into account. We can test our current implementation of the evaluation functions against these tests by using the bash script "runTestCases.sh" located in the backend_plugin folder in the repo. A report can be found with the these mock user input tests located in the target/site subfolders of the repo.

We will also run a nightly check against our current implementation through Travis in our github repo. This will provide a quick and easy reference to see if we are constantly improving as we continue to update and develop this project.

Currently, we only have a single data point of initial results. This graph will ultimately be a line graph showing how our plugin correctly catches more and more test cases as the project gets further along in development. We currently have 23 use cases we test against for our block comment evaluation function Our current implementation of this evaluation function correctly triggers on 16 of those 23 test cases.



Instructions on how to recreate these results can be found in our usual manual in the IDE-IT backend plugin repo located at https://github.com/DavidThien/IDE-IT.

Challenges/Risks

There are a handful of expected challenges and risks with this proposed Eclipse plugin. The most challenging aspects of this project is to create a plugin that's helpful to developers, correctly identifies and evaluations when the user could use information about a certain Eclipse feature, and does not produce false positive suggestions. An example of a false positive would be if a user was typing Javadoc comments above a method and our tool evaluated the action and triggered a block commenting suggestion.

To determine whether a feature provided by Eclipse has been neglected by the user, we need to take into account the changes made inside the document editor as well as the user mouse and key press actions that contributed to the change. Combining information from these input sources to accurately make this determination without triggering false positives will be difficult. As an example, if a user types two “//” slashes, are they commenting out an entire line or just part of the line? If they are only commenting part of the line, it would not be correct to tell them about the “cmd-/” shortcut to comment a line out, as this would comment the entire line. Also, are they commenting out an existing line, or starting a new line as a comment? Though using “cmd-/” will comment the line out in either case, it is normal for users to manually type “//” to comment out a fresh line, and constantly triggering this warning each time they do might be overkill. Thus the difficulty is not only figuring out how to evaluate if a given feature has been ignored, but also defining for each feature exactly what it means for the user to ignore it.

As a side-effect of an inefficiency-based recommendation system, evaluating whether a given feature has been ignored will also require unique evaluation code for each feature [7]. This means that boilerplate evaluation code will not be possible, and that as the number of features we evaluate for grows, the amount of time we will spend writing evaluation code will grow as well. Certain features will likely also be more difficult to evaluate for than others. For example, determining whether the user has ignored the “cmd-/” shortcut to comment a line out will likely be easier than determining whether the user has neglected the “Refactor->Rename” feature of Eclipse (which could involve complex evaluation of the document’s AST). To mitigate these challenges, it will help to think carefully about how to divide the work of writing evaluation code among the team. It will also help to organize our code/modules in such a way as to simplify the process of adding a new feature evaluation.

Responsibilities

Infrastructure and Testing: David
Public facing interface and evaluation functionality: John
Implementing listeners and Evaluator/EvaluatorManager classes: Eric

Schedule

- ✓ = Completed
- ☐ = Not completed

Week	Tasks
Week 4	• Write specification ✓ • Determine list of Eclipse functionality to evaluate for ✓ • Determine an interface to connect with the front end ✓ • Compile list of documentation and resources ✓ • Identify the extension points we need to integrate the plugin ✓ • Create a basic framework for a working Eclipse plug-in ✓
Week 5	• Write listeners for users making changes to the document in the document editor ✓ • Create presentation slides ✓ • Create user manual ✓
Week 6	• Write framework for evaluation of user input ✓ ○ Listens for document changes and sends relevant information to all feature evaluation functions
Week 7	• Interface with front end to produce a working prototype of plugin ✓ • Implement feature evaluations for adding and removing imports ✓ • Update manual based on current status ✓
Week 8	• Fix any issues reported from previous week ☐ • Implement feature evaluations for correcting indents and renaming variables ☐ • Go through thorough review process with the team to discuss current usability ☐

	• Determine what is feasible to fix and what is not in time remaining ☐ • Update manual based on current status ☐
Week 9	• Polish: include updates from previous week's discussion ☐ • Complete rough drafts of final documentation ☐ • Further usability discussion and testing ☐ • Update manual based on current status ☐
Week 10	• Refine specification ☐ • Prepare presentation materials ☐ • Update manual based on current status ☐

Works Cited

1. Albusays, Khaled and Ludi, Stephanie. (2016). "Eliciting Programming Challenges Faced by Developers with Visual Impairments: Exploratory Study." *IEEE/ACM Cooperative and Human Aspects of Software Engineering,* 82-85.

2. Johnson, Yoonki Song, Murphy-Hill, & Bowdidge. (2013). "Why don't software developers use static analysis tools to find bugs?" *Software Engineering (ICSE), 2013 35th International Conference on Software Engineering*, 672-681.

3. Kline, R., Seffah, A., Javahery, H., Donayee, M., & Rilling, J. (2002). Quantifying developer experiences via heuristic and psychometric evaluation. *Proceedings IEEE 2002 Syposia on Human Centric Computing Languages and Environments 2002,* 34-36.

4. Podkhalyuzin, Alexander. "IntelliJ IDEA Feature Suggester". https://plugins.jetbrains.com/plugin/7242-idea-feature-suggester

5. Karashevich, Dombrovsky. "IDE Features Trainer". https://dl.acm.org/citation.cfm?id=2393645

6. Palamarchuk, Andriy. "MouseFeed". https://marketplace.eclipse.org/content/mousefeed

7. Murphy-Hill, E., Jiresal, R., & Murphy, G.C. (2012). Improving software developers' fluency by recommending development environment commands. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 42:1-42:11

8. M. B. Twidale. Over the shoulder learning: Supporting brief informal learning. *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, 14(6):505–547, 2005.

Feedback

We have addressed all feedback.