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) is an Eclipse IDE plugin that detects when users are not taking advantage of the many features it includes and provides them with a notification to make them aware of those features that are relevant to what they are currently doing. Our project involves creating a backend service for IDE-IT, which consists 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 [9]. 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 teaches users about the existence of features that are actually relevant to the way they use Eclipse. IDE-IT continuously tracks user action in the form of document changes, 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].

Similarly to the IDE-IT backend, the IDEA Feature Suggester monitors changes within an open document and constantly evaluates to see if users could benefit from a feature of the IDE. Although 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. We have found based on their provided examples, that Murphy-Hill et. al's technique is mainly useful for long pipelines in the development process (e.g. git commands, large-scale refactorings) rather than for smaller single-command sized tasks as are present in much of the coding part of the development process.

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 track changes in the content of documents and in the annotations associated with a document window. When changes are detected, information from these inputs will be passed to a series of evaluation functions. Each IDE feature that IDE-IT supports has its own evaluation function, which checks for a specific sequence of user actions and/or changes that signal that the user has neglected to use the feature. The IDE features we are focusing on are features that we find to facilitate tasks that are most commonly performed when using Eclipse to write Java code.

We 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. Remove trailing whitespace on save
6. Refactor code base by renaming a variable throughout the entire project

Note that while 2 and 3 above map to the same feature (organize imports), they require different evaluation to determine whether the user would benefit from the feature. 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 have listeners for document changes and annotation changes within each editor window. The listeners provide the change data to the evaluation functions, and the evaluation functions are responsible for checking a set sequence of user action and/or document changes against the data provided by the listeners. For example, our block comment evaluation function checks for multiple sequential lines of code being commented out manually, 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 recognizes 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 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

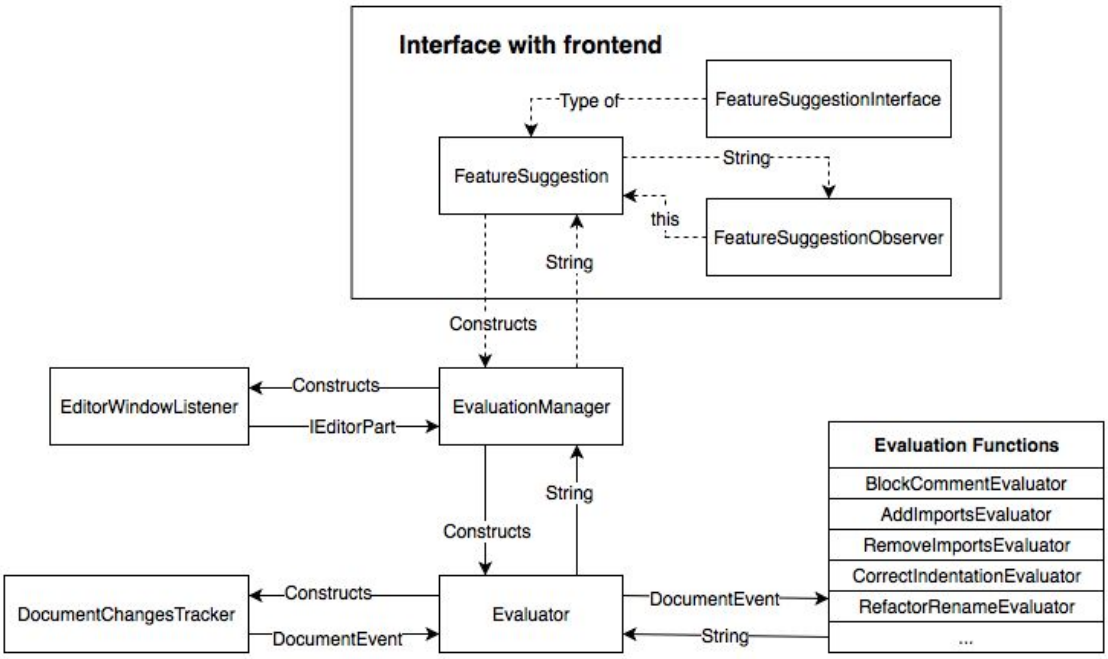


Figure 1: Diagram of internal architecture

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 provides the following interface. This allows 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.

FeatureSuggestionInterface

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
- trailingWhiteSpaceSuggestion
  - Indicates the user has manually removed whitespace from the end of a line, rather than taking advantage of Eclipse's setting to do that automatically on save
- variableRenameRefactorSuggestion
  - Indicates the user has manually renamed a variable in several points throughout the AST, rather than using the Refactor->Rename feature

We list these 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 and changes to the annotations on the document are detected by listeners. These listeners are the means of interfacing directly with eclipse and 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 pass their input to the feature evaluators which determine if a feature is relevant to the user's actions. Finally, the evaluator sends any successful feature evaluation results to the front end interface in the form of the above featureID Strings to signal that a tutorial or notification can potentially be triggered.

We have found a common pattern in how we detect whether some of our feature evaluators should be triggered. A pattern we often use is a state machine, where the state is defined by the last matching action that was performed or whether certain annotations exist in the editor. In some evaluators, the state also includes the timestamp of that action. For example, our block comment evaluator keeps track of the number of the last line a user commented out and the timestamp of when they did; when another line is commented out, its line number and timestamp is compared to the previous commented-out line number and timestamp to see if they are adjacent and performed over a certain threshold of time. This time threshold accounts for the case when the user actually does use the feature. In the case of the block comment feature, Eclipse actually implements this by performing the comment action line-by-line programmatically (i.e. faster than a human manually can). Without this time difference threshold, our evaluator would trigger several times during a block comment feature usage. To write this pattern,

State machine feature evaluators:

- Block comment evaluator
    - States are defined by the last line that was commented out and the timestamp of when it was commented out
    - When changes are made to the document content, determines whether the change resulted in a previously non-commented line being fully commented out using "//"
    - If a line is commented out:
        - Checks that the line is adjacent to the last line commented out, and that the timestamps of the two actions are > 100ms apart. If both are true, the feature is triggered
        - The state is updated to store the new commented line and timestamp as the last commented line and timestamp
- Add import statements evaluator
    - States are defined by whether an "unresolved type" annotation exists in the document

    - When changes to the annotation model of the document are made, the state is updated with a boolean flag indicating whether there are any existing unresolved type annotations
    - When a change is made to the document content, the line the change is made on is compared to the line before the change. If the change added an import statement to the line, and the current state is that unresolved type annotations exist, then the feature is triggered.
- Correct indentation evaluator
    - States are defined by the last line that indentation was adjusted on and the timestamp of when it was adjusted
    - When changes are made to the document content, determines if the change only adjusted the amount of white space at the front of the line (either with spaces or tabs), and if so, sets the state of the machine to reflect the new last line indented and its timestamp
    - If the change only adjusted the amount of white space at the front of the line, and the line is adjacent to the last line that had white space adjusted, and the changes were made at least 100ms apart, then the feature evaluation is triggered

Other feature evaluators:

- 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
- Trailing white spaces evaluator
    - When changes are made to the document content, stores the contents of the line the change was made on before the change is applied
    - After the change is applied, the before-change line is compared to the after-change line, and if the only change was removing all whitespace from the end of the line, the feature is triggered.

We use various Eclipse APIs to help with detecting user input and parsing the AST. The main API that we use is the Eclipse PDE (Plug-in Development Environment) which is used for developing, testing, and debugging eclipse plugins. We also take 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

To evaluate the success of our plugin, we are evaluating our success on each individual feature that we evaluate for. To do this, for each feature evaluation we have implemented we have instructed a few developers (students) to find every way they can to circumvent using that feature. As an example, for the block comment feature, we asked them to comment out multiple lines of code, but to pretend that they don't know the block commenting feature exists. By observing all the ways that our subjects circumvented each feature we obtained a set of cases (ways of circumventing the feature) for each feature that we want to cover in the feature's evaluation algorithm. Concrete measures of success for each feature are 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 eventually cover each feature 100% (all cases considered).

The main focus of this plugin is to increase feature discoverability. To that end, we 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.

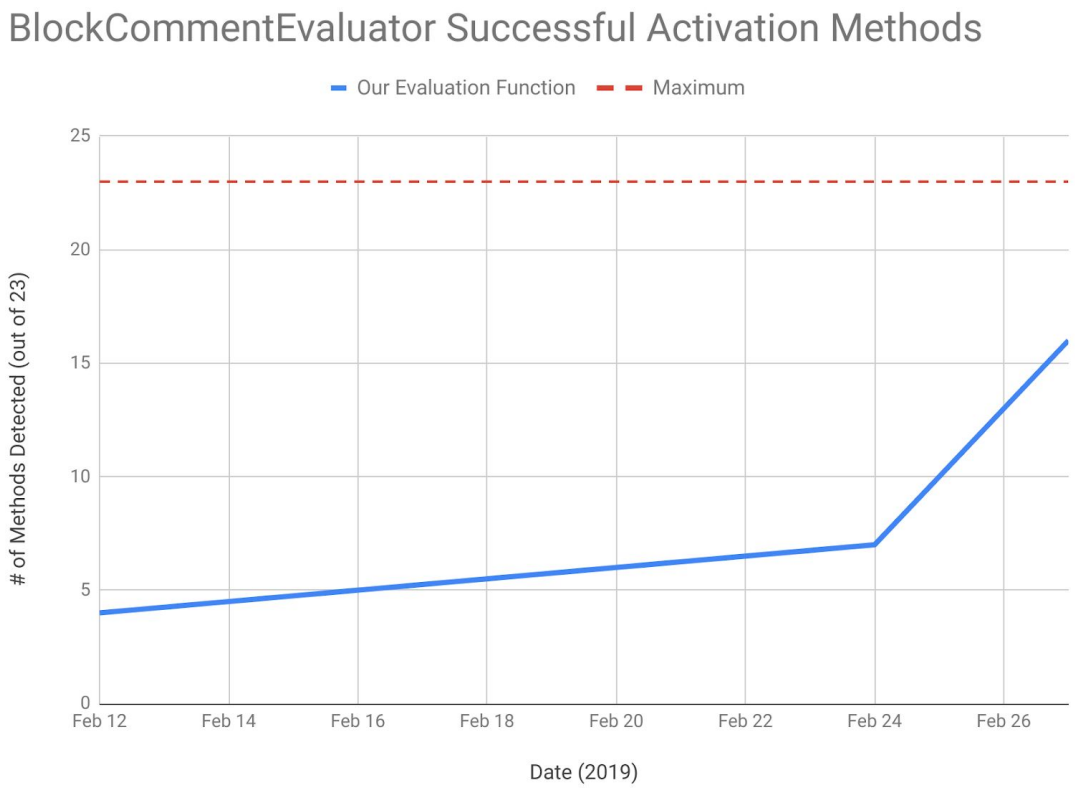
Initial Results

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. The results can be found in a report 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.

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. Of the 7 that do not trigger, 6 are due to the fact that we have not implemented checking for the user entering multiline "/* */" comments. Our focus thus far has been covering as many single line "//" comment cases as possible, since those are the type of comments Eclipse uses in its block commenting feature. The final non-triggering test case represents a user commenting out several lines of code in such a way that no two adjacent lines are commented out one after the

other (for example commenting out lines 3, 1, 4, 2 in that order). Though we would like to adjust our algorithm to cover this case, we recognize that it is a particularly unusual/unlikely way of commenting out several lines and thus not a primary concern for us.



We are in the process of creating similar tests for all of our evaluation functions. We also plan on creating a set of false positive user actions that we can test our evaluation functions against. A false positive would be a situation where our evaluation function triggers, but it shouldn't. These would be tests designed to do things similar to the actions we are trying to detect, but be slightly off. For example, if a user makes multiple end of line comments on consecutive lines, we do not want our evaluation function to trigger. The results of those tests would provide quantitative data showing how well ouf evaluation functions perform in regards to false positives.

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

There have been a handful of challenges when developing this plugin. The most challenging aspect of this project has simply been acclimating to the Eclipse plugin API. With no previous

experience between our team in creating Eclipse plugins, there was a ramp-up time in learning how Eclipse plugins in general are structured and how to access data we need for our listeners and evaluators. Learning this API is an ongoing process, and we are still finding new ways to take advantage of it to improve our project.

Another challenge we face is to correctly evaluate when the user could use information about a certain Eclipse feature without producing false positive suggestions. In general, to determine whether a feature provided by Eclipse has been neglected by the user, we take into account both the changes made inside the document editor as well as the state of the document annotations. Combining information from these sources to accurately make this determination without triggering false positives has been a tricky learning process. 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. Aside from writing the evaluation functions themselves, it has also been difficult to pinpoint a concrete definition 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 also requires unique evaluation code for each feature [7]. Though some amount of boilerplate code structure can be used for these in the form of the abstract methods of the FeatureEvaluator abstract class, the actual implementation of those abstract methods needs to be defined separately for each evaluator. This means that as the number of features we evaluate for grows, the amount of time we spend writing evaluation code grows as well. Certain features are also be more difficult to write evaluation functions for than others. To mitigate these challenges, it has helped to think carefully about how to divide the work of writing evaluation code among the team, as well as organizing our code/modules in such a way as to simplify the process of adding a new feature evaluation.

Lessons Learned

In creating the IDE-IT backend plugin, there are certain aspects of our process that we believe worked well, as well as some that we felt we could have improved upon in hindsight:

- Working as a team
    - Rather than pigeonholing ourselves into rigid roles/responsibilities in the process of creating our plugin, we instead ended up sharing the work of most responsibilities. Some amount of specialization and division of work occurred naturally, and was necessary to reach milestones on time. However, having multiple team members work on the same functionality allowed for different perspectives on the given problem, which allowed us to refine our solutions more effectively and learn new ways of doing things.
    - When pushing changes to our codebase, we required each other to submit pull requests so that we could do code reviews on submitted branches before

merging. This allowed us to catch many potential logic and style issues before contaminating our master branch with them. This also allowed all team members to see how one person used the Eclipse API, which could then be used elsewhere in the project.

- Specifications and planning
    - One issue we ran into during our work was that when we initially implemented our add/remove imports evaluation functions, we had various ideas about when these particular evaluation functions should actually be triggered, and implemented them before coming to a consensus as a group. It took a few days of communicating back and forth with each other and with the instructors before we finally agreed on how/when these should be triggered. Luckily, the evaluator functions we had written worked for our consensus, but the added confusion served as a lesson for agreeing on all parts of a specifications ahead of time.

- Working with Eclipse Plugin API
    - The Eclipse plugin API is massive. When we began our project, we thought that the time it would take to research this API would end up being one of the main challenges of our project. What we ended up discovering was that the functionality we needed for our plugin was a very small subset of this API. Though some amount of research was unavoidable, we found that by simply "jumping in" and trying things out with some help from online searches, we were able to locate this functionality rather quickly, rather than spending weeks just doing research. This aspect of our process allowed us to cover a lot of ground quickly in the beginning phases of the project.

- Having a "framework first" development mentality
    - Our original backend framework (i.e. input data that listeners catch and send to evaluation functions) was sufficient for our first couple evaluation functions, but as we added more evaluation functions we found that we needed to utilize new types of input data. When we improved the framework to track additional data, we often realized the evaluation function algorithms we had already written would have been simpler/more effective if we had included the additional data to begin with. Reworking these earlier evaluation functions to take advantage of the new data sources took extra time, which we might have been able to avoid if we had spent more time to develop a more thorough framework first.
    - We also learned that if you have a solid framework, adding new functionality becomes much easier. Regarding the issue of modularity, a key benefit of the way that we implemented our framework is that by separating our evaluation functions into their own classes that extend an abstract FeatureEvaluator class (which defines several methods useful to various evaluation functions), adding new evaluation functions became a much easier process, and the resulting code was simpler and more efficient.

    - When we implemented our build system, we had to refactor how our entire project was set up in order to have it meet the build system requirements. If we had thought about how our project would work in regards to a build system initially, we would have saved the time from having to refactor the project structure.

- Dealing with trade-offs
    - A tricky aspect of our project involved choosing how conservative we wanted to be when deciding whether or not to trigger an evaluation function. Too conservative and we risk not notifying the frontend when a user could truly benefit from a feature; not conservative enough and we risk inundating the frontend with excessive feature evaluation triggers. The lesson learned here was that issues like this need to be discussed and their options evaluated, as the choice between tradeoffs can have a major impact on the quality of the product

Discussion

We feel the project as a whole went well. There was a bit of stumbling early on before we were able to concretely decide the project focus and what each sub team would be responsible for. Once we overcame that hurdle and all members had a shared goal that we were working toward, it was a much smoother path. It also allowed interesting discussion and sharing of ideas toward what we all wanted this project to be. We all had the same idea of creating something that suggested different IDE features to user, but everyone had a slightly different vision of exactly how it was going to suggest those features. Working in a larger group and working out all of those details was a great experience in communication, idea sharing, and making decisions based on consensus or deciding that sub teams would be responsible for some decisions.

We were able to hit most of our original goals. The one disappointment was that we were unable to implement the evaluator for the Refactor-Rename feature. This feature evaluator would have involved a difficult process of tracking AST changes to determine when to trigger, and toward the end of our project we felt that our time would be better spent by implementing a number of simpler evaluator functions in order to increase the versatility of our project. In addition, we wanted to make time for refining and writing tests for what we had completed

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.

9. Barcellos, J., Thien, D., Wahlquist, E. “IDE-IT Backend Plugin”. https://github.com/DavidThien/IDE-IT

Appendix

Schedule

- ✔ = Completed

Week	Tasks
Week 4	• Write specification ✔ (2/1/19) • Determine list of Eclipse functionality to evaluate for ✔ (2/1/19) • Determine an interface to connect with the front end ✔ (2/1/19) • Compile list of documentation and resources ✔ (1/27/19) • Identify the extension points we need to integrate the plugin ✔ (2/3/19) • Create a basic framework for a working Eclipse plug-in ✔(2/3/19)
Week 5	• Write listeners for users making changes to the document in the document editor ✔ (2/10/19) • Create presentation slides ✔ (2/14/19) • Create user manual ✔ (2/12/19)
Week 6	• Write framework for evaluation of user input ✔ (2/18/19) ○ Listens for document changes and sends relevant information to all feature evaluation functions • Update manual based on current status ✔(2/18/19)
Week 7	• Interface with front end to produce a working prototype of plugin ✔ (2/20/19) • Implement feature evaluations for adding and removing imports ✔ (2/25/19) • Update manual based on current status ✔ (2/25/19)
Week 8	• Address feedback from previous week ✔ (3/4/19) • Implement feature evaluation for correcting indents ✔ (3/4/19) • Update manual based on current status ✔ (3/4/19) • Go through thorough review process with the team to discuss current usability, determine what is feasible to fix in time remaining
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

Feedback: We have addressed all feedback.