IDE-IT: Integrated Development Environments - Intelligent Tutorials
Evaluating Users' Knowledge of IDE Commands

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. 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. This document describes 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 only available for the IntelliJ IDEA application, thus Eclipse users cannot benefit from it. 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.

We have included support for the following list of features:
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. Generating getters and setters

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, we have focused entirely on the backend development of IDE-IT. 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.
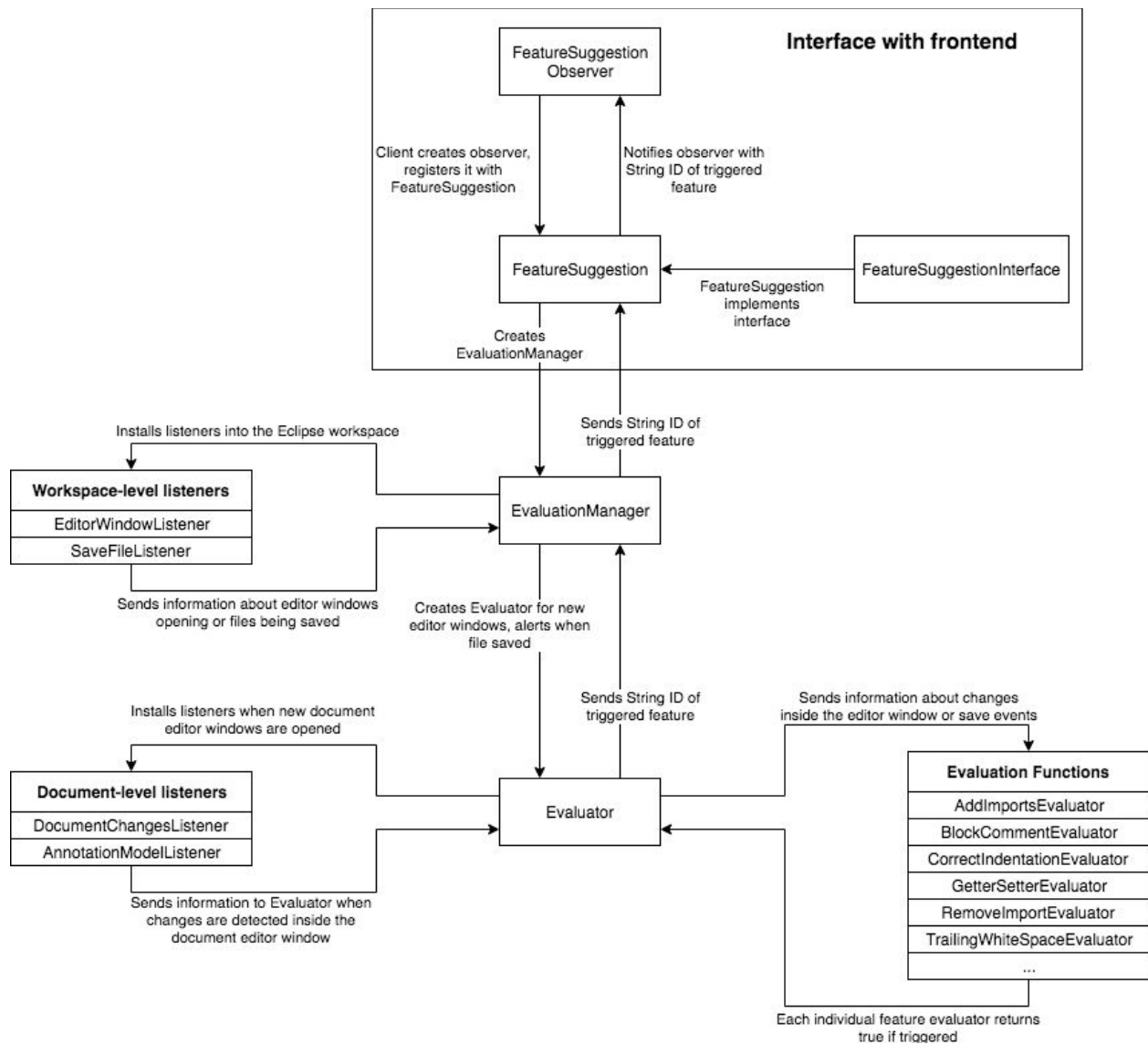
Architecture

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 is called when a feature evaluation has been triggered, where the String parameter contains a unique ID of the given feature.
- FeatureSuggestion
    - An implementation of the FeatureSuggestionInterface interface. This is the main object that frontend clients use to manage their interaction with our backend service. Once the client creates a FeatureSuggestion, they can register any number of their own FeatureSuggestionObserver objects with the FeatureSuggestion object to be notified when 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 evaluations notify the same FeatureSuggestion.
- EditorWindowListener
    - Listener that 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 notifies 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 cycles 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), the Evaluator notifies the EvaluatorManager with the unique ID string of the feature that was triggered.
- DocumentChangesListener
    - Created when a new Evaluator is assigned to a document editor window. Responsible for listening for changes made to the document within that document editor window. When changes are detected, the DocumentChangesListener passes the change information back to the Evaluator.
- AnnotationModelListener
    - Created when a new Evaluator is assigned to a document editor window. Responsible for listening for changes to the annotations within the document editor window (e.g. annotations regarding unused imports). When changes are detected, the AnnotationModelListener passes the change information back to the Evaluator.

- VariableDeclarationFinder
  - Parses the AST of a document and stores a list of variable names declared within the document. Designed to provide additional information for feature evaluation.
- Evaluation Functions
  - A set of classes extending an EvaluationFunction abstract class. Each feature that IDE-IT evaluates has its own class extending EvaluationFunction, which contains evaluate functions for document change events that are 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
- getterSetterSuggestion
  - Indicates the user has manually created a get___ or set___ method for a class variable without using Eclipse's "Generate Getters and Setters" command

We list these featureIDs in the file FeatureID.java, accessible by developers who wish to use our backend service, so they can implement support for those feature notifications.

Feature Evaluation

In order to start a feature evaluation, inputs from the user's document changes as well as changes to the annotations in the document window are detected by listeners. These listeners are the means of interfacing directly with Eclipse and act as the first measure to detect user interaction. 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 the user has circumvented an Eclipse feature.

We have found a common pattern in how we detect whether our feature evaluations should be triggered (i.e. detect if the user is circumventing a feature). A pattern we use is a state machine, where states are defined in various ways for each evaluation function, and document and annotation changes are the inputs that transition the machine between different states. 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.

The following are visual descriptions of these state machines. For each diagram, double-circled nodes indicate when the feature is triggered:
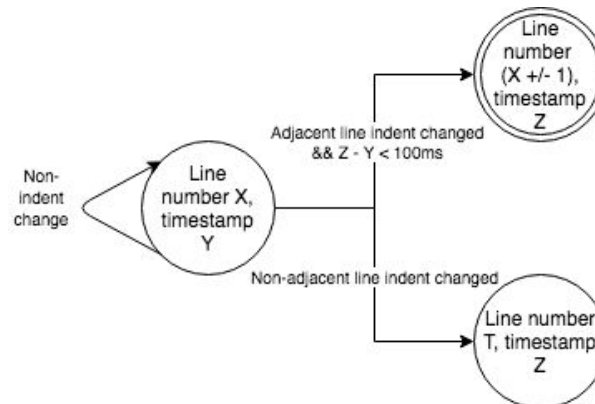
- Block comment evaluator
  - States defined by the last line number commented out and the timestamp of that comment
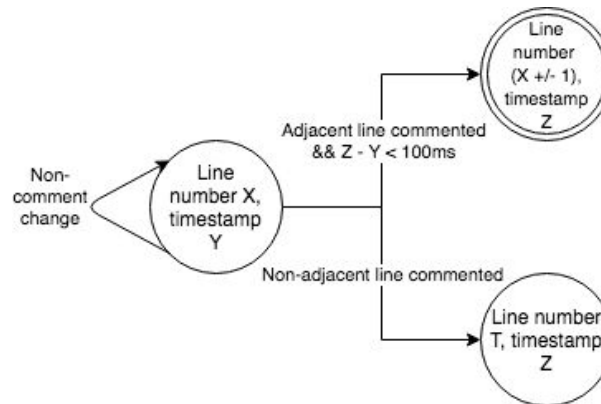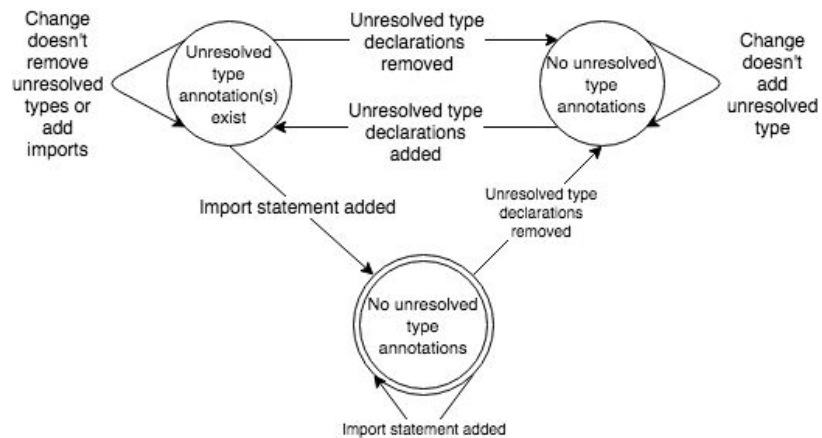

- Correct indentation evaluator
  - States defined by the last line number that the indent was changed on and the timestamp of that change
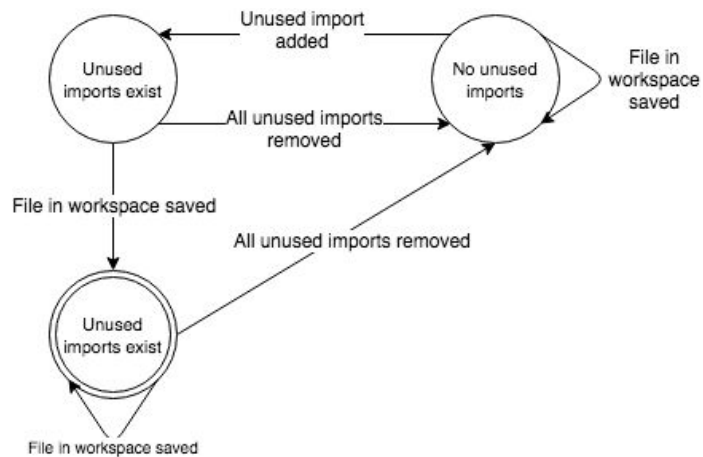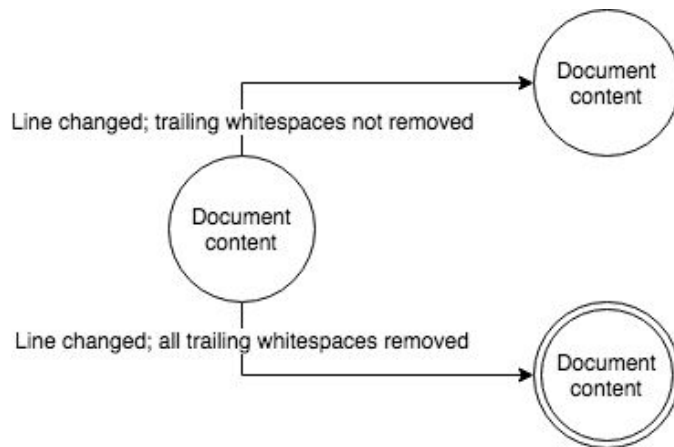

- Add import statements evaluator
  - States defined by the current status of unresolved type annotations

Change doesn't remove unresolved types or add imports | Unresolved type annotation(s) exist | Unresolved type declarations removed | No unresolved type annotations | Change doesn't add unresolved type

Unresolved type declarations added

Import statement added

Unresolved type declarations removed

No unresolved type annotations

Import statement added

- Remove import statements evaluator
  - States defined by the current status of unused imports across the entire Eclipse workspace

Unused import added | Unused imports exist | No unused imports | File in workspace saved

All unused imports removed

File in workspace saved

All unused imports removed

Unused imports exist

File in workspace saved

- Removing trailing white spaces on save evaluator
  - States defined by the current text content of the document

Line changed; trailing whitespaces not removed

Document content

Document content

Line changed; all trailing whitespaces removed

Document content

- Generating getters and setters evaluator
  - States defined by the current text content of the document. The AST is parsed when changes add a new public or protected statement, and if a document change adds a public or protected get*, set*, get_*, or set_* method declaration, where * is an existing field name, the feature is triggered.

Document change does not add getter or setter method declaration for field of class

Document content

Document change adds getter or setter method declaration for field of class

Document content
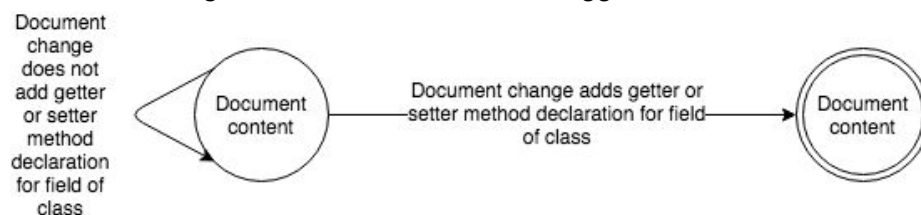
We use various Eclipse APIs to help with detecting user input and parsing the AST. The main API that we use is the Eclipse PDE (Plug-in Development Environment) which is used for developing, testing, and debugging eclipse plugins. We also take advantage of the Eclipse 4.x SDK and its underlying API. In the future, we also plan to package the plugin in Eclipse's standard plugin packaging system and potentially make it available in tandem with the front end on the Eclipse marketplace, an online package repository for Eclipse.

Evaluation

Methodology

To gauge the success of our plugin, we analyze the performance of each different type of feature suggestion evaluation against a pre-created list of possible user actions for the feature. These actions represent the user trying to circumvent using the given feature by performing the task manually. For example, with the block comment feature, we have a list of over two dozen ways a user could comment out 2-4 lines of code using "//" or using "/* */" comment styles. This includes manually typing the comment characters, pasting the comment characters, or using a hotkey to comment a single line at a time. These lists were created through the efforts of all the developers on this project and from over-the-shoulder observation of student volunteers who

were instructed to find every way they can to circumvent a given feature. We then translated those user action lists into test suites with tests for each action sequence. Each test mocks the user inputs that are equivalent to the given sequence of user actions. We then run our current implementation of the different feature suggestion evaluations against these tests and determine how well IDE-IT performs based on how many of these tests are passed successfully.

One issue with this methodology and the results it produces is that the success of each feature evaluator's performance depends partly on our own definition of how each feature should be triggered or not triggered, whereas end users may have different ideas about what constitutes circumventing a feature. This risk is mitigated somewhat by the fact that we have gathered our lists of user actions to test from a variety of people, but it is not likely that our definition will align with that of every user. Using the block comment evaluator as an example, we have defined its triggering to require a minimum of two lines commented out, whereas some users may prefer it to require a minimum of three lines commented out. However, because the main focus of this plugin is to increase feature discoverability, we err on the side of maximizing our true positive triggers even if it means also having more false positive triggers. A true positive trigger means that our plugin correctly suggests a feature to a user when it should. A false positive trigger means that our plugin suggests a feature to a user, but the feature suggested does not align with the actions that led to the suggestion. We can test for both true positives and false positives through user testing and quantify both. Ideally we want minimal amounts of false positives, but we focus on maximizing true positives as our main priority.

Results

Tests of true positive cases (cases where the feature evaluator should trigger)			
FeatureEvaluator	**Total Positive Cases**	**Cases Covered**	**Percentage Covered**
AddImport Evaluator	3	3	100%
BlockComment Evaluator	25	18	72%
CorrectIndentation Evaluator	39	37	94%
GetterSetter Evaluator	18	12	66%
RemoveImport Evaluator	1	1	100%
TrailingWhiteSpace Evaluator	6	6	100%

By looking through the use cases that we successfully catch and the ones we miss (when we should catch them), we can determine what IDE-IT does well and what limitations it has.

- Add import statements evaluator
    - Successfully detects when the user manually types or pastes an import statement when there is an unresolved variable declaration in the document.

- Block comment evaluator
    - Successfully detects when users comment out lines of code with "//" on multiple sequential lines. The "//" can be manually typed, pasted in, or the user can even use the hotkey to comment out a single line.
    - Fails to detect when the user comments out multiple lines of code in non-sequential order. For example, if the user types "//" at the start of line 3, then line 1, then line 2, and finally line 4.
    - Fails to detect when the user uses multiline commenting with "/* */". This is something we would like to successfully detect, but it is a non-trivial change from the current implementation.

- Correct indentation evaluator
    - Successfully detects when the user adds and/or removes one or more spaces or tabs at the beginning of multiple sequential lines of code.
    - Fails to detect when the user adds and/or removes one or more spaces or tabs to the start of multiple lines of code in non-sequential order. For example, adding or removing white space at the start of line 3, then line 1, then line 2, then line 4.

- Generating getters and setters evaluator
    - Successfully detects when the user manually types the public or protected method declaration with the name get[varName], get_[varName], set[varName], or set_ [varName]. Where varName is the variable name of a previously declared variable in the document.
    - Successfully detects when the user pastes in a public or protected method declaration with the name get[otherName], get_[otherName], set[otherName], or set_ [otherName], then backspaces to remove the [otherName] and manually types or pastes in a previously declared variable name. [otherName] is an arbitrary string that does not include a previously declared variable name.
    - Fails to detect when the user pastes in a public or protected method declaration with the name get[otherName], get_[otherName], set[otherName], or set_ [otherName], then selects [otherName] and manually types or pastes in a previously declared variable name, replacing the [otherName]

- Remove import statements evaluator

○ Successfully detects when there is an unused import annotation within the Eclipse workspace (any opened projects) upon saving the workspace.

● Removing trailing white spaces on save evaluator
○ Successfully detects when the user removes a space, tab, or a combination of the two from the end of the line. The whitespace can be removed by backspacing or deleting one character at a time or by selecting all of it and removing through delete or backspace.

We also have compiled a separate test suite that tests our evaluation functions against user test cases where we believe the evaluation function should *not* trigger. This list of tests is not comprehensive or exhaustive by any means. These tests were compiled as an extension of our unit tests and testing the evaluation functions with users and seeing when evaluations triggered incorrectly. For example, with the block comment evaluator, some of the negative case tests verify that the evaluator does not trigger when a user places "//" in the middle or the end of a line of code.

These results are included for completeness, but are not relied on with the same significance as the positive case tests. Any failing negative case tests are considered a bug to be fixed more than an indication of how well IDE-IT performs.

Tests of true negative cases (cases where the feature evaluator should not trigger)			
FeatureEvaluator	Total Negative Cases	Cases Covered	Percent Covered
AddImport Evaluator	5	5	100%
BlockComment Evaluator	4	4	100%
CorrectIndentation Evaluator	8	8	100%
GetterSetter Evaluator	8	8	100%
RemoveImport Evaluator	2	2	100%
TrailingWhiteSpace Evaluator	6	6	100%

A brief analysis of what the negative case results for each evaluation function are as follows:

- Add import statements evaluator
  - Successfully does not trigger when the user adds an import without an unresolved variable. (In this case, there is no import statement that Eclipse could automatically add for the user).
  - Successfully does not trigger when the user types anything beginning with "import", such as declaring a variable with the name "importantVariable" when there is an unresolved variable.

- Block comment evaluator
  - Successfully does not trigger when the user places the characters "//" anywhere besides the start of consecutive lines of code
  - Successfully does not trigger when the user places a single backslash in front of consecutive lines of code.

- Correct indentation evaluator
  - Successfully does not trigger when the user adds and/or removes spaces and/or tabs anywhere besides the start of consecutive lines of code

- Generating getters and setters evaluator
  - Successfully does not trigger when the user creates a get or set method declaration that does not include the name of a previously declared variable.
  - Successfully does not trigger when the user creates a get or set method declaration that includes a partial match to a previously declared variable.

- Remove import statements evaluator
  - Successfully does not trigger when the user saves the document with no existing unused import annotations.
  - Successfully does not trigger when the user saves the document after removing an unused import.

- Removing trailing white spaces on save evaluator
  - Successfully does not trigger when the user removes any character besides a space or tab at the end of a line.
  - Successfully does not trigger when the user removes a space or tab anywhere besides at the end of the line.

We can test our current implementation of the evaluation functions against our positive and negative cases by using the bash script "runTestCases.sh" located in the backend_plugin folder in the repository. The results can be found in a report located in the /target/site folder of your local repository after "runTestCases.sh" has been ran.

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

Because the IntelliJ IDEA Feature Suggester is similar to our plugin, we would like to be able to directly compare the results of the two plugins. Unfortunately, according to the IDEA Feature Suggester documentation, the set of features it evaluates has only two entries in common with IDE-IT - line/block commenting and auto-correcting indents. In our testing of the Feature Suggester, we were unable to produce any suggestions for auto-correcting indents. As for the line commenting evaluation, the IDEA Feature Suggester defines this trigger differently; the user is notified any time they comment even a single line out. Though this policy does cover all cases in our evaluator tests, it floods the frontend with excessive triggers; we feel that commenting out a single line manually is not the main issue, but rather when time is wasted manually commenting out several lines.

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 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. This 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 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 few 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

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.

Discussion

The IDE-IT backend plugin shows that inefficiency-based recommendation to address the issue of IDE feature discoverability is possible through real-time evaluation of user input. Though the number of features supported is minimal at this time, that number can easily increase as the main framework has been implemented and adding additional feature evaluators has become a simpler task through abstraction.

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.

If work is to continue on this plugin, the next step would be to rework our FeatureEvaluator interface to incorporate AST change information more efficiently. By doing so, we open the door to several more possibilities for features to support. We also would like to further increase the similarities between different evaluation functions to try to pin down a more common structure for all of them. If we can do this, adding additional features will become even easier, and the code will be much more consistent and easy to understand.

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 ✓ (3/8/19)
Week 9	- Update manual based on current status ✓ (3/11/19) - Polish: include updates from previous week's discussion ✓ (3/12/19) - Complete rough drafts of final documentation ✓ (3/12/19) - Further usability discussion and testing ✓ (3/12/19)
Week 10	- Refine specification ✓ (3/18/19) - Update manual based on current status ✓ (3/18/19) - Prepare presentation on project ✓ (3/18/19)

Feedback: We have addressed all feedback.