Motivation

Integrated Development Environments (IDEs) are software applications that provide tools and facilities for developers to write and test software. They can be immensely helpful to developers as they can provide more support to a developer and automate tasks that they would otherwise have to do manually. The cost of an IDE is not insignificant, and unfortunately there is evidence that this investment does not always pay off. They can have dozens or hundreds of great features, but without properly communicating those features to the user, they are essentially worthless. Results of surveys suggest that developers often use only a small number of IDE functionalities out of the total set available [4]. Kline also reported that other issues in percent use of existing tools include low affordance of IDE menus, icon, toolbars, and graphical representation of program structure [4]. 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]. The common issue that all of these studies discuss is the discoverability of IDE tools and plugins that already exist, and that if a developer does not know what their current issue or bug is, then they cannot know what to search, what features to turn on, and/or what plugins to download. Specifically discussing the Eclipse IDE, it has a vast plug-in ecosystem which offers rich rewards, but only for developers who know how to find these "gems".

For the reasons previously described, we propose development of a tool which improves discoverability of existing tools, settings, and plugins for IDEs, specifically Eclipse. IDE Intelligent Tutorials (IDE-IT) would be an Eclipse IDE plugin that provides intelligent tutorials for tools, suggestions for tools, and an interface to easily toggle tools between enabled and disabled. The goals of our proposed project is to teach users about the features of the Eclipse IDE that they may not be aware of. Our plugin will detect when users are not taking advantage of the many features it includes and inform them of how they could, thus providing relevant information at relevant times.

The closest existing plugin to accomplishing our goal is called MouseFeed. MouseFeed works by generating a popup notification any time the user clicks a button in the toolbar or a menu item, reminding them of the hotkey shortcut for that feature. 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. Currently, web searches are the best way to discover useful tools and plugins. Third party tutorials and tool suggestions are published in the form of long videos, cumbersome webpages, forums, and "Top 20 best features for…" articles. However, it's challenging to realize what your current problem is in the first place, let alone what tools or configurations for your IDE would help address these problems. Thus, it's difficult to

know a feature exists in an IDE for something that a developer would find incredibly useful, unless they stumble upon it in various ways (i.e. are told by a colleague, get frustrated with a tedious task and search for an easier solution, etc.). Additionally, most IDEs have some form of tips, usually tips of the day, that try and convey some of their features to the user. However, those tips are commonly irrelevant to the user, 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. 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 key difference between previous approaches to this problem and IDE-IT is that rather than teaching users more about features they already use, 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.

If successful, this plugin improving discoverability of other tools will help users to leverage features to increase the speed at which they write their code, and reduce time debugging and testing. As static analysis tools, debugging tools, etc. would be more readily available and accessible, developers would be more aware of their existence and thus more likely to use them. It would not only improve coding speed and save time, but it would also improve the quality while writing code, as well as improve the percentage use of the IDE, thus making the money spent on it more worthwhile. Those who would benefit most from such a tool are the users who do not take advantage of many of the available features of Eclipse. Developers will benefit as it makes their programming a more enjoyable experience. Companies as a whole will benefit as well as it will produce cleaner code the first time through by reducing the introduction of bugs, thus ensuring more reliable code, and reduce programming time, thus reducing costs. Corporations could then produce deliverables safer, quicker, and cheaper. Additionally, developers of Eclipse and Eclipse tools and plugins will care, as their tools will get more awareness and utilization.

Approach

Our approach is to spit up the functionality into two main components, the front end (FE) and the back end (BE). The BE will be responsible for collecting and monitoring user input through the IDE. The user’s keystrokes, mouse clicks, and the program they are currently editing will be collected. We imagine these will work similarly to listeners - they will actively monitor the work space and update various information as the user changes their code and manipulates their workspace. After this information is collected, it is evaluated to determine if a notification trigger

needs to be sent to the FE. The idea and approach for the evaluator functions is to keep an eye out for a specific sequence of actions (allowing some variability - the actions don't have to be sequential, but they do need to occur in a certain sequence within a reasonable time frame). If that specific sequence of actions is met, then the BE will trigger a feature suggestion to the FE. For example, if a user comments multiple sequential lines, then that will trigger a feature suggestion for commenting a block of code that will be passed to the FE.

The front end will receive the feature suggestions after they are triggered by user action. The user will then get notified of the feature or shortcut they could be using. The FE will have each suggestions for each feature that is supported in the plug in and display the appropriate information to the user. The information may be as simple as letting the user know about a key command or it could inform the user of a feature that the IDE has built in, but needs to be enabled. We hope to provide some more in depth functionality to the user, such as allowing the user could allow the user to toggle the feature on right there, instead of having to navigate through multiple menus and sub menus to find the right configuration to enable the feature.

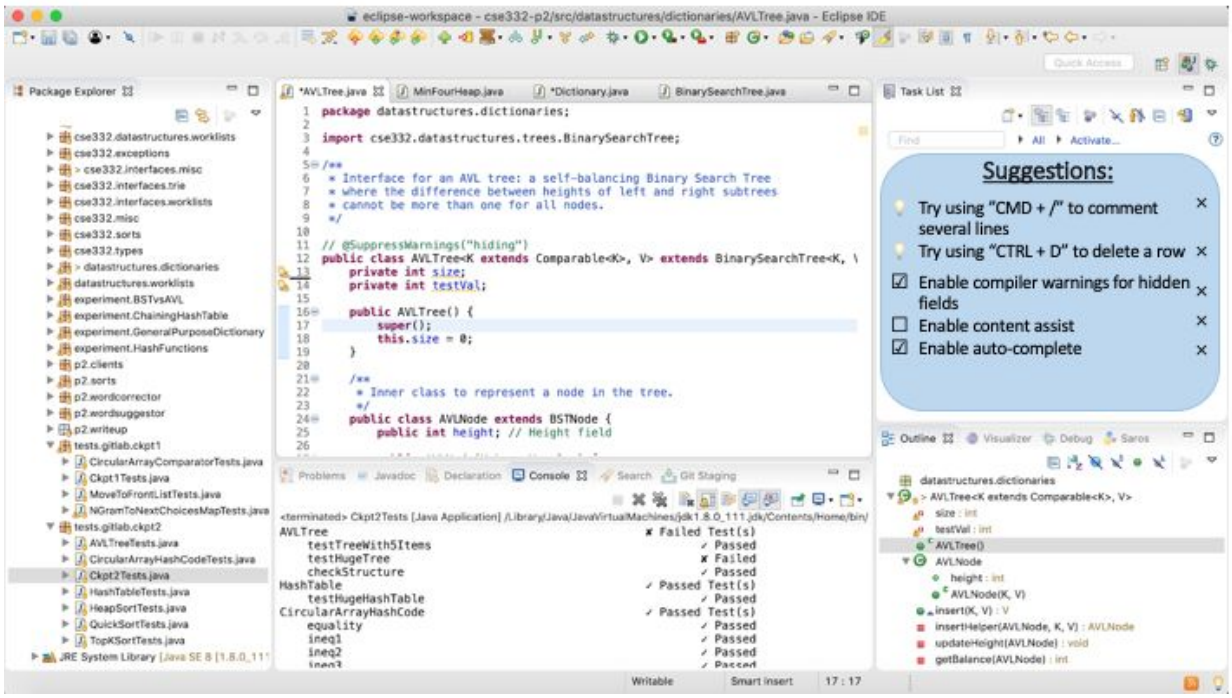

Image 1. Mockup of suggestion box displayed in IDE

To maximize encapsulation and minimize coupling, we will have a set interface that the BE and FE can use to communicate with each other. The only real shared resource required between both FE and BE will be a list of feature identifiers. Currently the plan is to have a set of names (strings) that uniquely identify each feature. From there the BE can push information to the FE when a feature is triggered and both sides will know exactly what that feature is.

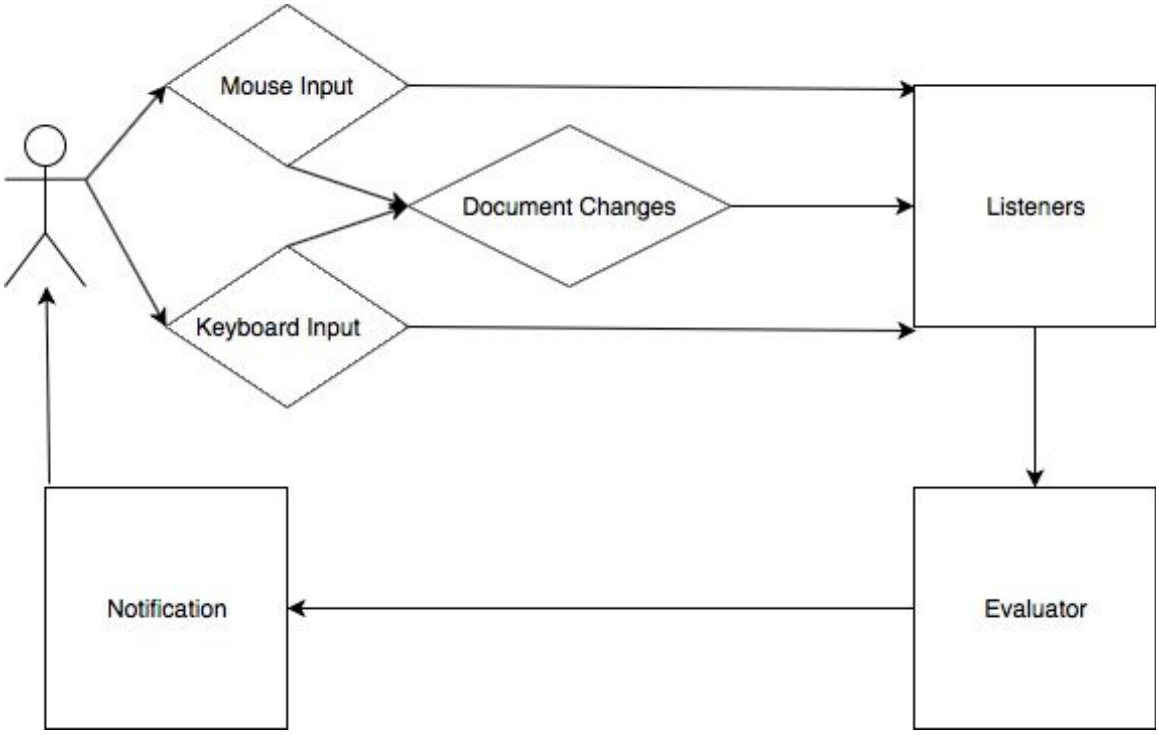

Image 2. Proposed methodology of how user inputs translate to feature suggestions

Outside of the standard unit and suite testing, we will do testing through experimentation. This will involving having ourselves (and hopefully others) use the plug in during development of their own projects. We let the users act naturally to see if it triggers any suggested features but mainly we will give specific tasks to the user. For example, we will tell a user to pretend that they don't know how to refactor their code through a universal variable rename feature, and to instead do that manually. If they act as if that feature doesn't exist, and try and complete this task manually, then the plug in should naturally suggest the correct feature. If it does not suggest the correct feature, then we can identify what actions the user took and decide if those sequences of actions should also be a trigger for the suggested feature.

This testing and experiment methodology can be effective, but one challenge is that it may prove to be more 'finicky' to actually do properly. There may be issues where a user completes a sequence of actions that's just slightly deviated from the trigger. Ideally our triggers will be general enough to catch this sequence of actions, but it also may not. There are infinitely many sequences of user action that can occur, but our triggers will only recognized a finite sequence of actions. One of the most challenging aspects is going to be making the plug in feel like it works well and natural with a variety of different sequences of user input.

The main way we will measure the success of our product is if our experimenters reliably trigger the suggested features when developing on their IDE while pretending they don't know about that feature. Without giving the experimenters the specific sequence of actions they need to take to generate the suggested feature, a success can be measured if the feature is suggested

by doing what they naturally would try anyway. A failure is if the suggested feature does not appear for the user, when they are actively trying to manually do something a feature could do better and quicker. These successes and failures could be tracked for each feature and quantified.

Challenges and Risks

In developing IDE-IT, certain functionality may prove to be difficult to implement. One of the primary functions of IDE-IT will be to constantly track user input including document changes, key presses, and mouse action. Once this is done, that input will need to be parsed and evaluated to determine whether or not a notification about a given feature needs to be displayed. Making this determination may be difficult, as correctly detecting when an action has been performed that a feature could have taken care of requires careful combining of information from all these input sources. In addition, various features will require unique evaluation code. 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. 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.

Another potential challenge of this project may be performance-related. Currently, Eclipse handles static evaluation of code in real-time as the user makes changes to the document. This evaluation is done efficiently, and does not seem to produce a noticeable performance drop. However, with our evaluation code stacked on top of the existing static evaluation, we may find that Eclipse performance is affected, particularly as the number of feature evaluations we perform increases. Unfortunately, whether or not this is the case will be difficult to determine until we have reached the point where we can begin to test our evaluations. One way to mitigate this risk will be to manually test our plugin as soon as we begin to add evaluations (it will be a good idea to manually test each time we add a new evaluation for a feature). If we notice that performance begins to fall during these tests, we may need to rethink how to perform certain evaluations.

In general throughout this project, we will be limited by the abilities of the Eclipse API. Initial research indicates that key functionality we need (such as the ability to listen for document changes, user key presses, mouse clicks etc.) exists within the Eclipse API. However, it is possible that intricacies of the API may present us with roadblocks that require us to rethink how to organize a particular module of the plugin. In some cases, this may not be apparent until we actually begin to write the code for a given module, meaning that some work may need to be re-done at times. We can minimize this risk by researching relevant functionality of the Eclipse API before designing or beginning work on a given area of our plugin.

Though there are no monetary costs in developing this plug-in, it will take time to build and test the tool to completion. We have created a plan for the next two months with the tasks needed to structure the back-end and front-end development. We anticipate to have a functioning interface in around six weeks, and we will use the following week to fix any issues that arise. In the case that we finish the basic implementation earlier than planned, we will add more functionality to the plug-in and refine our existing tool so that it is easy to use, intuitive, and helpful for the user during their development process.

Proposed Timeline

Week	Subteam	Tasks
Week 3	Frontend	
	Backend	
	All Team	- Project Proposal completed - Set up Git repo, mailing list, and communication methods - Determine each member's sub-team
Week 4	Frontend	- Compile list of documentation and resources to use - Decide on basic design of user interface - Choose interface elements to include (input controls, navigational components, informational components, etc.)
	Backend	- Basic Eclipse Plugin Framework Created - Write specification - Compile list of documentation and resources - Identify the extension points we need to integrate the plugin
	All Team	- Determine list of functionality we want to include - Determine what features to highlight and what their triggers will be
Week 5	Frontend	- Implement a working prototype for what the interface will look like - Will not use backend functions yet - Test the prototype functionality
	Backend	- Working functions that can read user mouse input, determine what highlighted text from a user is, determine hotkeys, determine when search function is used, as well as other inputs - Test for the functionality - Save and load config files for the plugin
	All Team	
Week 6	Frontend	- Continue implementation stated in week 5
	Backend	- Determine/implement algorithm to interpret user inputs, analyze desired goal, and map to useful tools/settings
	All Team	
Week 7	Frontend	- Interface with backend to allow a functional prototype
	Backend	- Interface with front end design to allow a functional prototype
	All Team	- Prototype of a working project with frontend and backend - Determine any issues, both aesthetically and functionally - Have 5-7 working tutorial functions
Week 8	Frontend	- Fix any issues reported from previous week

		- Include additional tutorial functions
	Backend	- Fix any issues reported from previous week - Include additional tutorial functions
	All Team	- 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
Week 9	Frontend	
	Backend	
	All Team	- Polish: include updates from previous week's discussion - Continue to add additional tutorial functionality as needed - Complete rough drafts of final documentation - Should have an almost fully functional plugin - Further usability discussion and testing
Week 10	Frontend	
	Backend	
	All Team	- Finalize everything - Refine specification - Prepare presentation materials

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. Khoo, Y., Foster, J., Hicks, M., & Sazawal, V. (2008). “Path projection for user-centered static analysis tools.” *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering,* 57-63.

4. 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.

5. Oberhuber, Martin. “Recommended Compiler Warnings.” *The Eclipse Foundation.*

6. Styger, Erich. (2013). “Top 10 Customization of Eclipse Settings.” *DZone.*