PANDA: Patterns of Arbitrary Nature Difference Applier

Project Engineer: Lily Mast

Program: Bachelor of Science in Computer Science
University of Evansville

Project Sponsor: Dr. Danny Dig
Project Advisor: Dr. Don Roberts

27 April 2016

ABSTRACT

Building on the foundations laid by research into refactorings, PANDA is a new IntelliJ IDEA plugin that will use patterns mined from Java projects on GitHub to automate code modifications. Unlike existing automated refactorings, these common yet diverse change patterns that alter program functionality will be recommended flexibly to the plugin, rather than being hardcoded like most current automated refactorings. PANDA applies these recommended changes to source code represented in an abstract format. The plugin then takes the modified abstract representation and turns it back into human-readable Java.
ACKNOWLEDGMENTS

The project engineer would like to thank her project advisor, Dr. Don Roberts, for his valuable advice and guidance throughout this project. She would also like to thank Dr. Danny Dig for his vision of creating an applier for diverse change patterns. Dr. Jean-Rémy Falleri, primary developer of GumTree, also deserves recognition for his collaboration with the project engineer on issues pertaining to GumTree’s method of tree generation and its nodes.

LIST OF FIGURES

Figure 1: Before-code and after-code
Figure 2: Before-tree
Figure 3: After-tree
Figure 4: Information flow diagram of the plugin
Figure 5: Flow diagram of code formatting
Figure 6: Pretty printed output versus unparsed output
Figure 7: Ambiguous method invocation AST
Figure 8: A user invokes the PANDA plugin
Figure 9: User rejects a code change recommendation
Figure 10: User sets code style preferences
Figure 11: User accepts a correctly formatted PANDA change
Figure 12: Unambiguous explicit method invocation AST

LIST OF TABLES

Table 1: List of conceptual actions
INTRODUCTION

Imagine writing a report for a recent project at work. Co-workers and those in similar jobs all over the world have written similar reports countless times before, but each time the topic and context are slightly different. This time is no exception, and even though each report follows the same general pattern, nearly the exact same work must be done over again. It would certainly be nice if there were a way to set the context of the report and automate the rest, needing afterwards only to smooth things over, polishing what the automatic report-writing engine had created.

This kind of automation for English-language reports, already in use for some of the more impersonal extant journalism articles (e.g. those concerning finance and stocks \(^1\)), makes a good analogy for the current work of the software engineering research group at Oregon State University (OSU) with computer code. Our goal is to speed up coding by mechanizing the implementation of common patterns that appear in Java projects. The research group plans to do this in the guise of an IntelliJ IDEA plugin that will integrate into the environment to which many Java coders are already accustomed.

PROBLEM STATEMENT AND BACKGROUND

In the software engineering industry, 60-90% of development costs arise from software evolution, and researchers have long sought to decrease these numbers; the goal of automating code changes is not new. When writing a program, developers often end up changing pieces of code in similar ways across many different projects. When the code changes do not impact the overall functionality of the code (analogous to rearranging paragraphs in a report to enhance clarity without changing the meaning of the document), these code changes are called refactorings.\(^2\) To refactor is to apply one or more refactorings to a piece of code.

Refactorings, the predecessors to our work, have been researched extensively. There are now many successful implementations of automations for these behavior-preserving patterns that can be automatically applied by the program in which a developer writes their code. This program is called an Integrated Development Environment, or IDE, and is analogous to the word processor used by the report writer. IDEs are able to apply refactorings to computer code by testing how the code runs before and after the refactoring. If it fails any of the pre- or post-condition checks, the refactoring cannot be applied because it would change the behavior of the program. These rigorous checks are what have made automatic refactoring so successful, because developers can count on the refactoring engine of their IDE not to break their code.

The automation of such behavior-preserving code changes has proved quite useful and has helped many programmers develop cleaner, more readable, and better designed code. However, developers do not always want a program to work exactly the same way after making changes. When adding a feature to a program, coders are changing its behavior, an action analogous to writing another section in a report to explain another aspect of a project. These non-behavior-preserving changes cannot be applied as easily as refactorings because the action of
checking whether the program functions the same before and after the change is no longer helpful in determining whether the transformation was what the developer intended to do. Such changes also defy easy categorization, whereas refactorings can be catalogued into a list of a couple hundred common changes. Non-behavior-preserving changes have almost infinite variation. As such, studying these kinds of changes is a very new area of research, and our research group is the first to try to create a tool that will apply common, but arbitrary behavior-changing patterns to a block of code.

An example of the current state of the art in applying behavior-changing patterns is given by such transformations as “Surround With…” in IntelliJ, an IDE preferred by many Java developers. This code change can surround code with a loop, a try/catch statement (used for making sure a particular block of code did not cause an error), or any of several other available templates. This functionality of IntelliJ still makes use of specialized pre- and post-tests that check the conditions before and after the change was applied. The developers of IntelliJ implemented each of these checks individually, so any of these “Surround With…” code change automations are only good for applying that particular change. That is, the pattern had to be identified by humans, then a specific applier program had to be written for that pattern only.

Our research goal is to be able to apply arbitrary patterns to code. In other words, the research group hopes to create a first program, the “Learner,” to recognize patterns (over all open Java repositories on GitHub) instead of relying on humans to identify them. Such a program would have the advantage of being able to look through far more code projects than any human would ever have the time to analyze or work on. Using the probability of contexts in which certain patterns tend to appear, a second program would act as a “Recommender” to suggest to a developer a pattern that may help them in their current project.

A third “Applier” program, nicknamed PANDA (Patterns of Arbitrary Nature Difference Applier), is the subject of this report. PANDA takes the suggested pattern (which can be any of the diverse patterns the Learner has identified) and applies it. It is an arbitrary pattern because it can be any that the Learner found, as contrasted with a particular pattern that has been fully analyzed and its use specially coded by humans. It is therefore the job of the Applier to take a difference, described in abstract terms, between the current code and the realized pattern and apply it to the code at hand. This allows for many more behavior-changing code transformations to be automated, since each pattern will not need to have its own applier implemented.

Decreasing common errors during the code-writing process is one of the main problems solved by the Applier tool. For example, a behavior-changing pattern a developer might want to follow in their implementation would be “Convert Element to Collection,” which converts a scalar argument to a method into a collection. To do this manually, the programmer would edit the method so it would do everything it originally did to the one scalar element to every element in the new collection. While converting this element, the developer may forget a step along the way, such as changing the type of the argument, pluralizing the name to reflect the argument is now a collection of elements, or iterating over the collection to execute the method’s
functionality on every element in the collection. When all of these steps are grouped together and described abstractly as one related pattern, applying the pattern is much less prone to error than manually typing in each change necessary to convert a scalar to a list.

An illustration of what the Applier does is shown in the figures below; the “Convert Element to Collection” pattern used in the example was mined in part by Mihai Codoban, one of the team members working on the Learner. The example before-code in Figure 1 is parsed into an abstract syntax tree (AST), a conceptual representation of what each piece of the code means. The Applier needs two distinct ASTs, one to represent the semantics of the before-code and one for the after-code. These are called the before-tree and after-tree, respectively. A parser creates the before-tree, shown in Figure 2. To that tree, PANDA applies the list of actions found in Table 1, which yields the after-tree in Figure 3. The after-tree then is converted back into code to become the after-code of Figure 1b.

**Figure 1:** (a) Before-code of converting an element to a collection. (b) After-code of converting an element to a collection.

**Figure 2:** Before-tree of converting an element to a collection. Dark purple nodes are those that will be moved during the application of the change. Numbers indicate which action from Table 1 (below) will affect a node.
Table 1: A conceptual list of the actions applied to the nodes of the before-tree to turn it into the after-tree.

<table>
<thead>
<tr>
<th>Action Number</th>
<th>Action</th>
<th>Type</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>insert</td>
<td>variable declaration</td>
<td>List&lt;Auto&gt; cars</td>
</tr>
<tr>
<td>2</td>
<td>insert</td>
<td>block</td>
<td>{ ... }</td>
</tr>
<tr>
<td>3</td>
<td>insert</td>
<td>parameterized type</td>
<td>List&lt;Auto&gt;</td>
</tr>
<tr>
<td>4</td>
<td>insert</td>
<td>identifier</td>
<td>cars</td>
</tr>
<tr>
<td>5</td>
<td>insert</td>
<td>enhanced for statement</td>
<td>for( ... ) (...)}</td>
</tr>
<tr>
<td>6</td>
<td>insert</td>
<td>simple type</td>
<td>List</td>
</tr>
<tr>
<td>7</td>
<td>insert</td>
<td>simple type</td>
<td>Auto</td>
</tr>
<tr>
<td>8</td>
<td>move</td>
<td>variable declaration</td>
<td>Auto car</td>
</tr>
<tr>
<td>9</td>
<td>insert</td>
<td>identifier</td>
<td>cars</td>
</tr>
<tr>
<td>10</td>
<td>move</td>
<td>block</td>
<td>{ ... }</td>
</tr>
<tr>
<td>11</td>
<td>insert</td>
<td>identifier</td>
<td>List</td>
</tr>
<tr>
<td>12</td>
<td>insert</td>
<td>identifier</td>
<td>Auto</td>
</tr>
</tbody>
</table>
The benefit of this project for developers is that it will allow them to focus their problem solving on the logic specific to the current project on which they are working (instead of on repetitive patterns that only change a little with respect to the program’s context). Like the automation of refactorings before them, the automation of these patterns should increase productivity and decrease errors in programming common patterns and the time spent debugging such errors.

**REQUIREMENTS AND SPECIFICATIONS**

In order for this project to be useful, there are requirements and specifications that must be met. It is necessary that the project meet the following stipulations so that it matches the information flow diagram in Figure 4 below. This diagram shows information flow between the components that must communicate in order for PANDA to work properly. Code and context must be read into the plugin from the editor; the code is then parsed into a before-tree. This tree and the context are fed to the Recommender, and the actions it recommends are applied to copies of the before-tree to create after-trees. These after-trees are then composed into human-readable code and displayed to programmers using the plugin. Programmers choose the code snippet that best matches the pattern they wish to implement, and this choice is reflected in the editor.
Figure 4: Information flow diagram of the PANDA IntelliJ IDEA plugin. Items in the shaded region indicate items within the project domain boundary.

In order to even start the information flow through the plugin, PANDA is required to work with Java projects. In other words, the input from the editor must accept Java and the final output of the Applier must be in Java. The language choice was motivated by the fact that Java is currently the most widely used general-purpose language. The Learner is working to mine Java patterns from all of the accessible projects on GitHub, and at this stage of the project the research group is only working on mining, recommending, and applying with respect to Java code. Once the research progresses far enough, it could be possible to make PANDA language-agnostic, but this is likely years away.

Another requirement is that the trees be parsed from code to trees and managed by the open-source project GumTree, which is dedicated to modeling differences between code as lists of “actions” on trees. This raises questions about how this dependency might affect the previous requirement. GumTree works with more languages than Java, including Ruby, JavaScript, and PHP. Eventually, therefore, GumTree could be used to help extend PANDA as well. For now, however, the project will continue to deal strictly with Java.

The form of PANDA is required to end up as an IntelliJ plugin. This means that the functionality of the Applier must be bundled into a program that can be installed as an add-on to the IDE, increasing its functionality by adding the abilities of the Applier to those natively
belonging to the IDE. This raises the specification that PANDA must be able to access the editor of IntelliJ and edit the contents thereof.

The PANDA project should also be able to interface with the Recommender. Since the Recommender must analyze the context in which it is invoked and then provide a recommendation, the plugin must account for both input and output of the Recommender. Input is the less complex of the two, because once the plugin knows the user seeks a recommendation, the Recommender need only be invoked with the code from the editor and the current position and waited upon to provide an output. Once the Recommender has statistically decided the most likely change to be applied, it must provide the code difference it is suggesting in an abstract form. The Applier must receive this abstract form and create from it a list of actions (see Table 1) that can be applied to the before-tree. A preferable situation for output would be that the Recommender would output a list of actions ready to be applied to the tree, but it is more likely that the Applier will have to massage the format of the data somewhat so that the information is in the form of an action list (defined by the GumTree application programming interface, or API).

PANDA must also meet the specification for ability to apply arbitrary actions to a before-tree to create an after-tree. This means that the Applier be able to modify a before-tree with arbitrary moves, inserts, deletes, and updates in order to turn it into the after-tree. It must be able to do this given individual actions or an arbitrary number thereof. The Recommender will be suggesting the next actions in the pattern. The edits that make up the pattern pertain only to the code modified by the pattern and no other code. PANDA should carry out only the edits specified in the recommendation. Therefore it is required that the Applier not alter any edits the user has made that are unrelated to the pattern.

Some of the more fine-grained specifications of the project’s dealings with the editor are those of formatting. When a change recommendation is accepted by the developer and applied to the existing code, several items should be kept in mind. Whitespace must be preserved. In addition, any code that is not being changed should keep its formatting while new code should match it as closely as possible. Most developers have a preferred way of formatting their code, so any preferences indicated by the user must be respected when a change is applied. If these conditions are not met, the act of applying a change is sure to be quite jarring for the programmer. If code reformats to a standard set by PANDA without regard for preferences, users are likely to resent the plugin and stop using it. Therefore, the project must react flexibly to the original formatting of the text in the editor.

In this same vein, any comments in the code must also be preserved. These notes used by programmers for explanation and documentation purposes within a code file have no bearing on the behavior of the program, yet are often critical to programmers trying to understand the functionality of a code file. Care must be taken here to avoid discarding the comments even though they do not change how the program works, again not only because of how jarring it
would seem to the programmer to have a comment disappear, but also because of how removing these explanations would greatly decrease the readability and maintainability of the code.

Similarly, edits a user has made to the code file separate from the application of the pattern should remain untouched by the plugin. Such a guarantee is yet another insurance that users of PANDA will not feel undermined by the plugin. PANDA, acting only on recommendations, should allow for easy reversal of any applied patterns.

A specified item that is not part of the plugin is a “validation” of GumTree. Validation is an attribute of the apply function. (The apply function itself is represented as a loop in Figure 4.) It is required that this function “validates” GumTree. In this context, validation constitutes creating the same after-tree as parsing the after-file would. Checking whether this validation attribute holds will be carried out with known patterns, and therefore known before- and after-files. Therefore, it is a separate test of the function not seen in the final plugin by the user, but rather in a test suite unto itself.

A helpful illustration of such validation can be found using the “Convert Element to Collection” example explained above. GumTree’s standalone operation would read in the code of Figure 1a to build the before-tree seen in Figure 2. It would then read in the code of Figure 1b and build the after-tree shown in Figure 3. When GumTree takes the difference of these these two ASTs, the result is the list of actions in Table 1. PANDA should, given the action list and the tree in Figure 2, produce the tree in Figure 3. All of this would be done using known code, almost exactly like what appears in Figure 1, as known inputs and their respective expected outputs. This may seem trivial, but it is essential that this validation show the reproducibility of the after-tree by PANDA before the program tries to apply arbitrary patterns to source code. Otherwise, the application of actions suggested by the Recommender will not be reliable applications of the target pattern.

**DESIGN APPROACH**

Such a novel problem offers many opportunities and challenges in designing a solution. Interfacing with the Recommender, actually applying the changes it suggests, presenting options to and receiving choices from the user, and outputting reasonable Java code are the main focus areas of the design problem.

The Recommender currently is run on a trained database completely separately from the rest of the project. The invocation of the Recommender should be encapsulated into a function call, passing the before-tree and necessary context, and receiving as output a list of recommended patterns. The project engineer designed an interface for this exchange, but found that the Recommender is not yet able to adequately provide the necessary information. Therefore, a mocked-up Recommender was created. The ramifications of this are further discussed in the Results section.

The general specification of being able to apply arbitrary actions to a before-tree to create an after-tree is one that offers several alternatives for implementation. Some are too complex or
too integrated with the IDE. Since it is preferable to keep PANDA as language- and IDE-agnostic as possible for potential future extension to other languages and IDEs, these methods were discarded and the method of applying actions to a GumTree AST, alluded to above, was adopted. Further discussion of previous and current methods of application of arbitrary actions is given below.

The original approach for applying arbitrary actions was to read them from the list, recognize them, and apply them directly within the IntelliJ editor. This approach required familiarity with the IDE’s AST wrapper format known as the Program Structure Interface (PSI). GumTree actions were analyzed for their node type, which was then translated to PSI. For example, what GumTree calls a simple name is known as an identifier in a PSI tree. The problem with this method was that the different trees used for code abstraction did not have a one-to-one correspondence. An illustration of this can be found in method declarations. In PSI format, one of the child nodes of a method declaration node is its parameter list, which in turn has all of the parameters as children. However, in GumTree’s AST representation, a method declaration has all of its parameters as children (without the intermediary parameter list node). This caused a disconnect in the IntelliJ editor; when trying to apply an add action using PSI, it needs to know the correct parent underneath which to add the new element. Since GumTree has no vocabulary for a parameter list, adding a parameter to a method invocation involved a convoluted process to find the proper child (parameter list) of the grandparent (the method declaration) of the parameter to be added. The process also had many corner and edge cases that needed to be addressed, causing PANDA’s implementation code to become less understandable. Therefore, other methods were explored to solve the problem of applying the actions.

The new approach for applying actions is to use GumTree’s format for every one of PANDA’s internal processes. This means that instead of applying actions directly into the PSI of the IntelliJ editor, the actions are applied onto the before-tree (previously only used for input to the Recommender). This will produce an after-tree. This after-tree will then be pretty-printed into the editor. Pretty-printing is like typing; it takes the meaning contained in an AST and “writes out” what the tree means in code. The pretty-printed string will then be pasted into the editor, where IntelliJ will automatically parse it into PSI as for all of the other code the user typed there.

These strings that are pasted into the editor have the IntelliJ formatter run on them. Most developers have a specific way they like to format their code, which is likely to be different from the formatting of the pretty-printed string. The solution for this is that IntelliJ has settings where developers can set their formatting preferences. Once these preferences are set, any IntelliJ plugin may invoke the formatter, and the code in the editor will be properly formatted according to those settings. It is in this way that PANDA outputs acceptable-looking Java code. This part of the process occurs during steps 9 through 12 of the information flow outlined in Figure 4, and is illustrated in greater detail in Figure 5 below.
A final problem mentioned in the specification section, that of preserving user edits unrelated to the applied pattern, did not require extra implementation work to solve. Although it is one that refactoring researchers have studied and struggled with in the past, PANDA has a unique advantage in that the patterns it applies are arbitrary. Since refactoring research deals with hardcoded changes that must meet pre- and postconditions, ways of maintaining unrelated user edits have been developed. A notable example can be found in refactoring tool BeneFactor. This tool rolls back changes the user has already made before applying the refactoring. After the behavior-preserving change has been completed by the engine, BeneFactor reappplies any changes that were unrelated to the refactoring.\(^7\) Another approach by the refactoring tool WitchDoctor was to detect refactoring opportunities, even in unparsable program states, before the user started to work on them by hand.\(^8\) These two tools rely on the hard-coded refactorings mentioned earlier, which run strict pre- and post-tests to make sure behavior is preserved. Since

---

**Figure 5:** Flow diagram of how pretty-printed strings are formatted. This figure follows the same numbering scheme as Figure 4; formatting occurs in steps 9 through 12 of that process. Step 0, the user setting the code style preferences, occurs before the PANDA plugin is invoked.
the patterns PANDA deals with are allowed to change the behavior of the code, and PANDA applies by inserting individual nodes into ASTs that may or may not be parsable, modifying unrelated user changes is not an issue this plugin has to take into account. However, the project engineer explored precedent techniques to deal with the problem if it would have presented an impediment to the plugin.

To address the issue of user satisfaction with applied recommendations, the project engineer encapsulated the pattern application in a Write Command Action, an IntelliJ construct that allows for easy undo/redo functionality of a plugin’s actions. She designed a popup graphical user interface (GUI) element to ask the user whether they were satisfied with the change that had been applied. This ensures that the user knows they can undo the applied pattern if they are not satisfied with it.

The choice to use the GumTree format for the entire application process takes care of the issue of comments, as GumTree has AST nodes representing both line comments and block comments. (However, these nodes have yet to be utilized by GumTree, which will be addressed later.) Even so, this choice to pretty-print a GumTree AST into the editor raises some interesting issues. The first method of dealing directly with the PSI during application preserved the programmer’s settings in terms of formatting as well as their notes in the form of comments. However, this method had to be discarded since GumTree AST and PSI are incompatible. Pretty-printing assumes a common format for Java coders, but the formatter fixes any assumptions that are found to be in error once the code formatting preferences are set.

The original design assumed a pretty-printing functionality within GumTree itself. However, the project engineer discovered that this functionality does not yet exist in that project. This discovery led to the slight modification of steps 9.1 and 9.2 in Figure 5 to use unparsed strings instead of pretty-printed strings. The difference between between an unparsed string and a pretty-printed string is that the latter is formatted based on a standard. (See Figure 6 below for an illustration of the differences between the two.) However, both have the same semantic meaning when parsed. This means that when these strings are turned into ASTs, the trees will be identical. The project engineer decided to implement an Unparser component rather than a pretty printer since the original design and the final project make use of the IntelliJ formatter to retain user code style preferences anyway. A pretty printer would make assumptions about formatting that would have been overridden by the formatter in any case, so it makes more sense to defer all formatting responsibility to the code formatter.
Before implementing the Unparser to add AST-to-Java functionality to GumTree, an additional design step was required. The best way of unparsing an AST from a separate module (in this case, GumTree) would be to use the visitor pattern. However, the visitor pattern requires the objects it is visiting to be of different types so it can call the `visit()` method polymorphically. However, all nodes in a GumTree AST are of type `Tree`. Instead of by type, they are differentiated by an integer field that is assigned a type number. Therefore, the design for the Unparser functionality employs a switch statement on the type of each node. The Unparser has a method for each node type to specify how it should be unparsed (the same design as if the project engineer had been able to use the visitor pattern).

RESULTS

PANDA is now a functional plugin. It adequately fulfills the requirements and specifications. The project engineer made a few minor alterations to the design plan, which have been discussed. The majority of the original design was viable and followed.

The plugin has been implemented for IntelliJ IDEA, fulfilling the requirement that the pattern applying functionality be available in this popular IDE. The plugin deals with the Java language, as required. It is able to read in the Java code that the user types into the editor and outputs Java code. Once it reads in the Java source, the string is passed to GumTree and successfully turned into a tree in GumTree’s AST structure.

The plugin has an interface with a mocked-up Recommender. This defines the API for the information that will eventually be passed to the real Recommender. This step was necessary because the existing Recommender, still in development by our research partners at Iowa State University (ISU), exhibits too much coupling between the database training step and the step where a recommendation is provided. The project engineer has fulfilled the requirement of creating an interface with the Recommender by passing the information the real Recommender eventually will need to the mocked-up Recommender, which returns a mocked-up recommendation in the format (i.e. list of actions) that will be provided by the real Recommender. Once the design of the Recommender is improved to decouple the time-consuming database training step...
(which takes many hours) from the recommendation-providing step, it should be easy to insert it into the project since the interface is already defined. Currently, the mocked-up Recommender only supplies a couple hard-coded recommendations for specific file contexts. This stems from the fact that the research team working on the Learner is still working to find more patterns. Even if many patterns had already been identified, the PANDA project engineer has no way to determine which patterns are more likely in a given context without the statistical analysis and recommendation phase of the Recommender being decoupled from the database training phase.

PANDA displays its ability to apply arbitrary actions through a rigorous suite of tests that examine application of moves, updates, deletes, and inserts. These tests also validate GumTree by showing that the AST diff it provides gives the correct information to create the expected tree. These tests also ensure that unrelated user edits are preserved by verifying that no independent nodes are altered.

The Unparser is functional for many GumTree nodes. However, while implementing the Unparser, the project engineer discovered that in its creation of its AST, GumTree loses semantics. In particular, the ASTs for implicit and explicit method invocations are indistinguishable. For example, GumTree produces the same tree for the strings `foo.bar(baz)` and `foo(bar, baz)`, namely a method invocation node with three identifier/simple name nodes as children (shown in Figure 7). This ambiguity led to the project engineer contacting Dr. Jean-Rémy Falleri, head developer of the GumTree project. The project engineer is working with Dr. Falleri to resolve this ambiguity. This communication also revealed that the Unparser would be a welcome addition to the GumTree open source project.

![Figure 7](image)

*Figure 7:* Ambiguous AST created by both the explicit method invocation `foo.bar(baz)` and the implicit method invocation `foo(bar, baz)`.

The original design also assumed that GumTree used the comment nodes it defines. During implementation, the project engineer discovered that out of the three comment node types (JavaDoc, line comment, and block comment), only JavaDoc nodes are added to the GumTree AST, and even these lose the original text. The project engineer will work with Dr. Falleri to resolve these comment retention issues before pushing her Unparser to the GumTree repository.

PANDA runs IntelliJ’s built-in code formatter on the output of the Unparser. The lack of code formatting on the right of Figure 6 is clearly unacceptable, but so too would be the output of a pretty printer that made assumptions about code style. For example, suppose a user prefers spaces inside their sets of parentheses. If they invoke a PANDA recommendation, as in Figure 8,
and the style of their code changes when the recommendation is applied as shown in Figure 9, they are unlikely to accept the change because it does not respect their whitespace preferences. PANDA deals with this flexibly, deferring the formatting options to the user. Figure 10 shows how a user can choose how their code will be formatted in the IntelliJ preferences. PANDA respects these preferences, and the recommendation applied in Figure 11 is accepted by the user because it both respects their personal style preferences and helps their coding endeavor.

Figure 8: A user (who prefers whitespace inside their parentheses) prepares to request a PANDA recommendation.

Figure 9: The user rejects the change, not because it is unhelpful, but because it disregards their code style preferences. This illustrates the negative repercussions of using a pretty printer on its own.
Figure 10: A user sets their code style preferences to retain whitespace within certain parentheses.

Figure 11: The user accepts the change because it is useful and respects their formatting preferences.

FUTURE WORK

Since the existing plugin uses a mocked-up Recommender, the most salient opportunity for future work on the PANDA plugin itself is encapsulating a call to the real Recommender. This will require cooperation with the Recommender team at ISU to decouple their database-training functionality from their recommendation-providing functionality.

The current mocked-up Recommender’s output (a list of actions) is simplified; it assumes the Recommender will only give one recommendation when in fact it will offer between three and five. Each pattern recommendation consists of a list of actions required to perform that pattern. In order to determine which pattern to apply, the programmer must choose which is the most useful. To this end, PANDA should be expanded so that it applies all potential patterns, producing multiple after-trees. These will be pretty-printed into temporary views, which the developer can review and choose between via a GUI. These views will be presented to the user
as a pop-up window or large context menu, and will be implemented using the GUI elements
available to IntelliJ plugin developers. The user will click on the widget in the window or the
large menu item containing the code they wish to choose. This GUI that presents the multiple
recommendations has to come after the Recommender is created. It is not feasible to mock this
up since it would require the project engineer to identify the top three or five patterns for a code
file context, and these patterns have not even been identified yet by the Learner. Once the
Learner identifies enough patterns, the Recommender will be trained on a database containing
them. After the Recommender has the correctly trained pattern “vocabulary” and an improved,
decoupled design, this phase of PANDA’s development can begin.

The project sponsor hopes to see a Replayer component in the future. Although this will
not be part of the PANDA plugin, it will be useful for the three-part project as a whole (the
Learner, Recommender, and PANDA the Applier). This component will act as a verification of
the usefulness of the Recommender. Such a Replayer will be a robot of sorts, which will simulate
manual changes that a user might make by “typing” code up to a halfway point (n/2) in the action
list of length n. PANDA will then be invoked in hopes that it causes the Recommender to
suggest the second half of the pattern (e.g. some subset of actions from Table 1). Once its
suggestions have been applied, a comparison will be made between the expected output and the
output created by the Replayer. Expected output would be obtained by applying all n actions to
the before-tree normally. This expected after-tree is deterministic because of the validation of
PANDA. The Replayer, in contrast, will stop “typing” at the n/2 action and use PANDA to apply
the Recommender’s suggestions from that point forward, making a created after-tree. If this
results in the created after-tree being equal to the expected after-tree, the Recommender was
useful in that situation. If, however, the Replayer has not created an equivalent after-tree, that run
will be recorded as the Recommender not being useful. The Replayer will run many times, each
time testing a different list of actions representing a pattern to be applied. Each run will be
recorded as either a success or a failure, which will be useful for generating a report analyzing
the usefulness of the Recommender. However, since this Replayer program is not part of
PANDA but rather a verification of the Recommender useful only to the research as a whole and
not to the Applier itself, the Replayer has been left as a future project. Because the
Recommender is not ready, implementation of the Replayer must wait until the Recommender is
viable to be included in the plugin.

Future work that does not hinge on the completion of the Recommender centers around
the Unparser. Utilizing the existing GumTree comment nodes to retain programmers’ notes
would certainly enhance the Unparser’s usefulness. Also, the project engineer and Dr. Falleri
have agreed that an additional “receiver” node needs to be added to GumTree. The AST in
Figure 7 should only be created when the implicit method invocation foo(bar, baz) is
parsed. When the explicit method invocation foo.bar(baz) is parsed, the tree in Figure 12
below should result. The GumTree team should be addressing this problem presently. Once this
ambiguity is fixed, it will be easy to complete implementation of the remaining unparsing cases and release the Unparser to be used by the public as part of the GumTree package.

![Unambiguous AST representing the explicit method invocation foo.bar(baz).](image)

**CONCLUSION**

The primary significance of this project is the development of the Unparser. Such functionality will convert GumTree ASTs to Java code. Due to the fact that the recommendation-providing step of the Recommender is not yet sufficiently uncoupled from the time-consuming database training step, the recommendation of diverse learned patterns hinges on the progress of the team at ISU. However, the Unparser has already shown itself to be a coveted addition to GumTree. Its implementation has prompted inquiries from international academic groups (who use GumTree in their research) as to when it will be available and if they can assist in its development. Therefore, the project engineer intends to continue working with Dr. Falleri to resolve parse ambiguities and retain comments. Collaboration with others to refine and test the Unparser is also likely. These next steps will lead to the Unparser being added to the GumTree open-source project on GitHub and used by researchers around the world.

**REFERENCES**

BIOGRAPHY

Lilia “Lily” Mast is an up-and-coming computer scientist with a passion for software engineering at a high level of abstraction. She grew up in Colorado and chose to attend the University of Evansville (in Evansville, Indiana) for its rare mix of an ABET-accredited CS program in the context of a small, liberal-arts university. In addition to her computer science major, she is completing minors in mathematics and music studies. After graduation, she plans to join a community theatre company in her spare time to re-engage her passion for the stage. In July of this year, she will start her career as a software engineer at Data Ductus, a Swedish-based company that specializes in network function virtualization and software defined networks.