Clipboard: System Editing with Speculative Analysis and Naming Synthesis

ABSTRACT

Program transformation by demonstration can be helpful to complete the code automatically based on the examples given by developers. For instance, automatic transformation can be done with *systematic edits* that are similar but not identical edits to multiple locations for bug fixes, feature editions, and new API adaptation. However, existed transformation tools are not popular because of its tedious configuration process, unintuitive variable names, and lack of impact analysis.

To address the limitations mentioned above, we implement a systematic editing tool called CLIPBOARD to demonstrate examples via drag-and-drop, recommend suitable recipes based on the live edit stream on the fly, synthesize reasonable names based on the naming patterns, and inform developers of the speculative impact of the program transformation.

Our evaluation through blind experiments and controlled user studies shows that our approach greatly improves the usability of existed program transformation tools, the name generated by CLIPBOARD is equally intuitive as the manual created variable names, and the speculative analysis encourage developers to perform safe transformations to multiple places while reminding them of the relevant changes that should be applied before making a systematic edit.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Program editors*

General Terms

Search-based software engineering, Experimentation

Keywords

Program Transformation, Speculative Analysis, Systematic Edit, Naming Pattern

1. INTRODUCTION

Recent studies show that developers repeat their own mistakes or unknowingly repeat the errors from others in similar but not identical contexts. They are leaning to reuse existed code fragments with adaptive changes for similar programming tasks to save development effort [41]. Tools such as linked editing [51], CloneTracker [9] and Cleman [44] are able to modify multiple code fragments as one yet they are confined to identical changes on clone regions. Other API migration tools [42, 3, 40] focus on high-level stylized API transformation with adaptation patterns from codebase.

Sydit [31] and LASE [32] match codebase with the given systematic edit template, abstract context-aware transformation from examples, identify similar edit locations, and apply adaptive changes to similar code segments. Cookbook [20] actively matches the incoming edit stream with multiple edit recipes to recommend suitable transformation rules on the fly. Yet they require developers to manually fill in the variable names introduced by the systematic edits and cannot guarantee a safe transformation without introducing new compilation errors.

Considering that systematic edits have similar syntactic structure yet different types and names, we match pertinent identifiers names and extract mapping rules between matched pairs. Based on these mapping rules, we synthesize reasonable variable names that does not exist before at the target location. For instance, given a new inserted statement `IActionBar actionBar = fContainer.getActionBars()`, our approach automatically recommends the name of the new variable in target as `serviceLocator` in the statement `IServiceLocator serviceLocator = fContainer.getServiceLocator()`.

To help developers make decisions by informing them of the consequence of the transformation, we proactively apply the change to the target location and invoke Eclipse compilation error checker to evaluate the impact of the systematic edits. If the change will introduce any compilation errors, our tool asks for developer's confirm before applying the transformation.

To further simplify the process of generating, selecting and invoking the program transformation rule, CLIPBOARD provides an intuitive drag-and-drop approach to demonstrate the snapshots of the example code region before and after the transformation. Developers can apply the transformation by dragging the item of the edit recipe to the target context, double click on the recommended edit location, or invoking *content assist* engine in Eclipse and completing the change on the fly.

To assess the usability, intuitiveness and efficiency of CLIPBOARD, we conduct a four phases user study with X developers. The results illustrate that CLIPBOARD is able to generate equally intuitive systematic edits and the names of the identifiers initialized by the edits. The drag-and-drop approach to demonstrate the edits is useful, intuitive, and effective for the example-based program transformation.

In summary, our paper makes the following three contributions.

- To simplify the definition, invocation, and configuration process of example-based systematic edits, we allow developers to manipulate the region of systematic edits intuitively via drag-and-drop. Given a list of edit recipes, we match the most suitable systematic edit on the fly and invoke the transformation based on the incoming edit stream.
- We conduct a speculative analysis of the potential compilation errors caused by the transformation and inform the developer of the problems along with the preview of the target code region after the transformation.
- To increase the comprehension of the auto-generated transformation code, we leverage naming patterns between relevant identifiers and synthesize the identifier names initialized by the changes.

The rest of the paper is organized as follows. Section 2 places our tool in the context of related works in code recommendation and systematic edit transformation. Section 3 overviews our approach with an motivation example. Section 4 describes the concrete steps for our tools with matching, filtering and consequence evaluation algorithms. Section 5 presents the design of the user interface and the evaluation results are described in Section 6. The next two sections discusses the threats to the validity and future work for our program transformation recommendation technique. And the paper concludes in Section 9.

2. RELATED WORK

Example-based Editing ClipBoard finds its roots in *programming by demonstration* (PBD), which is also called programming by examples [28]. In PBD, the user demonstrated one or more examples of a program, and the system generalizes the demonstration into a program that can be applied to other examples. Simultaneous editing [33] performs a group of repetitive text editing simultaneously with selection guessing [35] and text constraints [34]. Although the PBD text editors, such as Sublime [18], are able to provide multiple selection and batch editing functions to help developers, they can only make the same textual changes to multiple locations simultaneously. Clipboard exploits program AST structure and performs similar but not identical systematic editing to multiple places.

Semdiff [7] automatically recommends adaptive changes in the face of high-level method additions and deletions, while our approach focuses on more fine-grained systematic transformation and recommend similar edits based on the syntactic structure similarity. Negara et.al [39] mine fine-grained frequent code change patterns with the AST nodes but are unable to transform the similar change to multiple edit locations.Sydit [31] generates edit scripts from a single systematic edit example and requires developers to specify the target locations before applying the transformation. Whereas LASE [32] overcomes the insufficient transformation rule generation from a single example and identifies the target transformation locations automatically. Different from these tools, Clipboard is able to actively match the incoming edit stream with multiple edit templates to recommend the most suitable transformation rule with compilation error checking.

Code clone management Recent research results pointed out that software system inevitably contain a large amount of similar code due to the copy-and-paste programming practice [6, 21]. These similar or identical code fragments, called *code clones*, may impact software quality with inconsistent modifications made to the cloned code snippets [11]. Different from these clone-related bugs detection tools[27, 12], ClipBoard focuses on performing semantic consistency transformation with compilation error checking rather than identifying inconsistency changes after the edits are applied to the codebase.

Some clone management tools, such as linked editing [51], Cleman [44], CloneTracker [9], and CloneBoard [8] detect code clones, allow developers to modify multiple code clones as one, and track changes in separate clone groups. Yet none of these tools are able to apply context-awareness edit transformation.

Program synthesis and code completion It is common that the programmer knows what type of object he needs, but does not know how to write the code to get the object. Program synthesis tools, such as Jungloid Mining [30] and CodeHint [13], search for a chain of method calls to complete the calling path at runtime based on specifications. SemFix [43] and MintHint [22] automatically repairs methods based on symbolic execution, constraint solving and program synthesis. GraPacc [40] extracts context features, rank the best matched pattern from the database and fill in the code automatically, yet it is confined to the pre-defined specifications or API usage patterns.Our prior work, Cookbook [20], automatically matches the incoming edit stream against multiple edit recipes. But it does not recognize naming patterns between related types and variables and fails to evaluation the consequence of the transformation.

The interaction process of Clipboard is similar to the Drag-and-drop Refactoring [26], which improves the invocation and completion approach of refactoring in an intuitive manner but focuses on a set of pre-defined refactoring templates.

Speculative Transformation Codebase Replication [38, 36] and Quick Fix Scout [37] maintain a copy of developer's codebase, makes that copy codebase available for offline analyses to run without disturbed by the developer, and implicates the potential offline code changes continuously. ClipBoard focuses on systematic edit checking before it is actually applied to the source code, rather than general code changes or quick fix recommendations.

Different from change impact analysis tools such as Chianti [47], our approach proactively applies the transformation to the source code and invokes Eclipse compilation error checker to identify compilation errors before undoing the change to the original version of the codebase.

Naming Pattern Caprile et.al. [5] reconstruct identifier names based on both standard lexicon and standard syntax, yet it is confined to pre-defined naming standards. Butler et.al. [4] and Singer et.al. [50] focus on conventional

Java class naming patterns and create a prototype to inform programmers of particular problems or optimization opportunities in their code based on the naming patterns. Høst et.al. [16] exploit whether or not a method name and implementation are likely to be good matches for each other and provide a simple pattern-based naming recommendation approach to evaluate reasonable names for the methods. Kashiwabara et.al [24] recommend candidate verbs for method names based on the association rules extracted from similar code fragments. Aligned with the idea of extracting association rules from similar code fragments, we construct recommended variable names based on the related contexts of the systematic edits.

3. MOTIVATING EXAMPLE

This section describes three scenarios that Clipboard can be used to perform the systematic edits with speculative consequence evaluation and naming synthesis, to apply the adaptive changes to multiple places, and to recommend the edit recipe by monitoring the living edit stream.

org.eclipse.jdt.core.dom.DefaultCommentMapper.v9800 and v9801

```
1. class DefaultCommentMapper {
2. - HashMap<ASTNode, int[]> leadingComments;
3. - HashMap<ASTNode, int[]> trailingComments;
4. + ASTNode[] leadingNodes;
5. + int leadingPtr;
6. + int trailingingPtr;
7. + ASTNode[] trailingNodes;
8.
9. Comment[] getLeadingComments(ASTNode node) {
10. - if (this.leadingComments != null) {
11. -  int[] range = (int[])this.leadingComments.get(node);
12. +     if (this.leadingPtr >= 0) {
13. +   int[] range = null;
14. +   for (int i=0; range==null && i<=this.leadingPtr; i++) {
15. +   if (this.leadingNodes[i] == node)
16. +         range=this.leadingIndexes[i];
17. +  }
18.   if (range != null) {
19.  int length = range[1]-range[0]+1;
20.  Comment[] leadComments = new Comment[length];
21.  return  leadComments;
22.  }
23. }
24. return null;
25.}
26. Comment[] getTrailingComments(ASTNode node) {
27. - if (this.trailingComments != null) {
28. -  int[] range = (int[])this.trailingComments.get(node);
29. +     if (this.trailingPtr >= 0) {
30. +   int[] range = null;
31. +   for (int i=0; range==null && i<=this.trailingPtr; i++) {
32. +    if (this.trailingNodes[i] == node)
33. +          range=this.trailingIndexes[i];
34. +   }
35.   if (range != null) {
36.    int length = range[1]-range[0]+1;
37.    Comment[] trailComments = new Comment[length];
38.    return  trailComments;
39.  }
40. }
41. return null;
42.}
```

Figure 1: Motivation Example to apply the systematic edit with speculative evaluation and naming synthesis

We use a motivating example drawn from `org.jdt.core.dom.DefaultCommentMapper` between `v9800 and v9801`.

Shown in Figure 1, the statements in black remain unchanged while the statements in red with "`-`" are the deleted statements and statements illustrated in blue with "`+`" are inserted statements in the new version. The function m_a (`getLeadingComments()`) and m_b (`getTrailingComments()`) in the class `DefaultCommentMapper` have similar syntactic structure yet different variable types and function names.

Suppose Alice intends to apply similar changes to m_a and m_b by adding an initialization sentence and updating the conditional statement. Such similar yet identical edits cannot be applied via search-and-replace feature and none of the existing refactoring engines in IDE is able to perform these changes. Without assistance, Alice has to manually edit both methods, which is tedious and error-prone.

Using Clipboard, Alice demonstrates the transformation by selecting the region of the function m_a and dragging it to the Clipboard View. Clipboard identifies the method(s) that this code region belongs to and locates all methods that are similar to the code fragment dropped at the Clipboard. After adding a new integer `leadingPtr` and a new instance of `ASTNode` named as `leadingNodes` in the class `DefaultCommentMapper`, Alice edits the function m_a and drags the new version to the Clipboard. Based on two snapshots, Clipboard extracts the matched statements and syntactic programming difference which are a sequence of inserts, deletes, moves and updates edit operations. Our tool recognizes the naming pattern of the relevant function names (`getLeadingComments()` and `getTrailingComments()`) and variable names (`leadingComments` and `trailingComments`) in the old version, extracts the association rule that replacing the string of `lead` to `trail`, and synthesizes the name of the new initialized variable in line 29 as `trailingPtr` according to the variable name `leadingPtr` in the template. Clipboard generates the name of new initialized `ASTNode` in line 32 as `trailingNodes` in the same way. Notice that Alice hasn't defined the new integer `trailingPtr` and new `ASTNode trailingNodes`, when Clipboard conducts speculative analysis of the transformation, it identifies these compilation errors and informs Alice of the problems with the error dialog `trailingPtr cannot be resolved or is not a field at line X, trailingNodes cannot be resolved or is not a field at line X`. Alice selects whether to apply this transformation by ignoring the error or apply it later after resolving the compilation error. If Alice defines the two variables in line 6 and 7 before invoking the transformation, Clipboard will apply the change directly after checking that the transformation is safe and free of compilation errors.

Suppose Alice keeps on programming and finds another location that she prefers to apply the same systematic edit, she invokes the *content assist* engine of Eclipse with the command `Ctrl+space` or types a "`.`" for the function call. When she finds a suitable edit recipe in the pop-up menu, she selects it and checks the preview of the change along with the corresponding compilation errors caused by the change in the area of additional information next to the pop-up menu. Alice decides to make this transformation after reviewing the change preview and the result of the speculative analysis. She makes another click on the menu item and the adaptive change is made to the target. She can also drag the edit recipe from the Clipboard View to the target for the program transformation as well.

4. APPROACH

We discuss our approach to implement CLIPBOARD in four aspects. Section 4.1 describes how we create edit recipe from the given example(s). Section 4.2 shows how we recommend edit locations that may require the same systematic edit. We

discuss the naming synthesis and how to create the substituted code fragment based on the edit recipe in Section 4.3. And finally we share our implementation of speculative analysis in Section 4.4.

4.1 Generate edit recipe

4.1.1 Syntactic program differencing

To generate an edit recipe from the snapshots before and after the edit, we first extract syntactic edits using ChangeDistiller [10]. For each exemplar changed method $m_i \in M = \{m_1, m_2, ..., m_n\}$, our tool compares the old and new versions of m_i and creates a sequence of edits: $E_i = [e_1, e_2, ..., e_k]$ where e_i is an insert, delete, update, or move operation of the AST statement.

- **insert(Node u, Node v, int k)**: insert u and place it as the $(k+1)^{th}$ child of v.
- **delete (Node u)**: delete u.
- **update (Node u, Node v)**: replace u with v in the tree
- **move(Node u, Node v, int k)** move u from its current position to the $(k+1)^{th}$ child of v.

For each insert, update, or move operation which are included in the new version, if there is a variable or a function in this statement that does not exist in the old version, it is added to a set of identifiers that need a name in the target code location. For instance, in Figure **??**, the systematic edit of m_a `getActionBars()` is specified as the input of the edit recipe, CLIPBOARD extracts the sequence of edit operations:

$$E_1 = \begin{cases} e_1 & \text{(Insert: line 3)}, \\ e_2 & \text{(Update: line 2,4)}, \\ e_3 & \text{(Update: line 7,8)} \end{cases}$$

The variable `actionBars` is identified as a new variable which is created by the systematic edit.

4.1.2 Identify common edit operations

$$LCEOS(s(E_i, p), s(E_j, q)) = \begin{cases} 0 & \text{if } p = 0 \text{ or } q = 0 \\ LCEOS(s(E_i, p-1), s(E_j, q-1)) + 1 & \text{if } eq(E_p, E_q) \\ max(LCEOS(s(E_i, p)), s(E_j, q-1)), \\ \quad LCEOS(s(E_i, p-1), s(E_j, q))) & \text{if } !eq(E_p, E_q) \end{cases} \quad (1)$$

$s(E_*, i)$ represents the subsequence $e_1, ..., e_i$ in E_*

Given more than one example, CLIPBOARD identifies common edit operations subsequence in $\{E_1, E_2, ..., E_n\}$ using Longest Common Edit Location Subsequence algorithm [19] described in Equation 1. In Figure **??**, the systematic edit of m_a `getActionBars()` and the change of m_b `getServiceLocator` are demonstrated as the input of the edit recipe. CLIPBOARD extracts two sequences of edit operations E_1 and E_2 as below:

$$E_1 = \begin{cases} e_1 & \text{(Insert: line 3)}, \\ e_2 & \text{(Update: line 2,4)}, \\ e_3 & \text{(Update: line 7,8)} \end{cases}, E_2 = \begin{cases} e_1 & \text{(Insert: line 3)}, \\ e_2 & \text{(Update: line 2,4)}, \\ e_3 & \text{(Insert: line 7)}, \\ e_4 & \text{(Update: line 8,9)} \end{cases}$$

The common edit operations are specified as pink in Figure 2. The longest common subsequence is extracted as e_1, e_2, e_3 in E_1 excluding the edit operation e_3 in the sequence of E_2, which is specified as blue in Figure 2. If the two edit operations e_p, e_q have the same type and their labels' bigram string similarity [1] is above the threshold t_s, we define that these two edit operations are *equivalent* ($eq(e_p, e_q)$). We use the same bigram string similarity threshold $t_s = 0.6$ as LASE [32] to include more matches.

4.1.3 Partially generalize identifiers in edit operations

For each matched edit operation, we extract the matched variable types, variable names, and method names. In Figure **??**, for the matched edit operations $E_1 : e_1$(line 3) and $E_2 : e_1$ (line 3), we receive the pairs as below:

$$\begin{cases} p_1(\texttt{IActionBars} : \texttt{IServiceLocator}), \\ p_2(\texttt{actionBars} : \texttt{serviceLocator}), \\ p_3(\texttt{fContainer.getActionBars():} \\ \texttt{fContainer.getServiceLocator()}) \end{cases}$$

We concrete the Longest Common Subsequence (LCS) [19] in these pairs yet abstract the different subsequences. By removing the longest common subsequences from the pairs, we generalize a common association rule: `ActionBars` $\rightarrow$ `ServiceLocator` from p_1. The same rule is detected at p_2 and p_3 without considering the case difference.

$$\begin{cases} p_1(\texttt{ActionBars} \rightarrow \texttt{IServiceLocator}), \\ p_2(\texttt{actionBars} \rightarrow \texttt{serviceLocator}), \\ p_3(\texttt{ActionBars} \rightarrow \texttt{ServiceLocator}) \end{cases}$$

We define the frequency threshold of the association rules as t_f and use $t_f = 2$ to get rid of trivial mapping rules. In other word, if the association rule is detected for more than once, it is regarded as significant for the common edit operations.

Using the association rules, our tool partially generalizes the common edit operations. For example, the common edit operation $E_1 : e_1$ ($E2 : e_1$) is generalized as `Ir$0` r`$0 = fContainer.getr$0();`

4.1.4 Extract common edit context

CLIPBOARD searches for the common edit contexts using a statement-level parser adapted from Fluri et.al. [10]. This parser first traverses the AST in pre-order and collapses all the nodes within one statement into one node that specifies the statement, as shown in Figure 2. Each statement is collapsed into one node while keeping the tree structure. Using Maximum Common Embedded Subtree Extraction (MCESE) algorithm [29], we compare the AST structure of the method m_c with the AST structure of the edit recipe context before the transformation. Shown in Equation 2, this algorithm goes through the tree structure, compare the statement type of each node, and evaluate the textual similarity via an off-the-shelf clone detection algorithm [23]. This clone detection method concatenates the statement into tokens and substitutes the identifiers, types and constants into abstract tokens based on the Java grammar rules, so that code portions with different variable names can be identified as equivalent. If the types of the node p, q are equivalent ($eq(s[0], t[0])$), we measure the similarity of the nodes with the number of the matched tokens using the

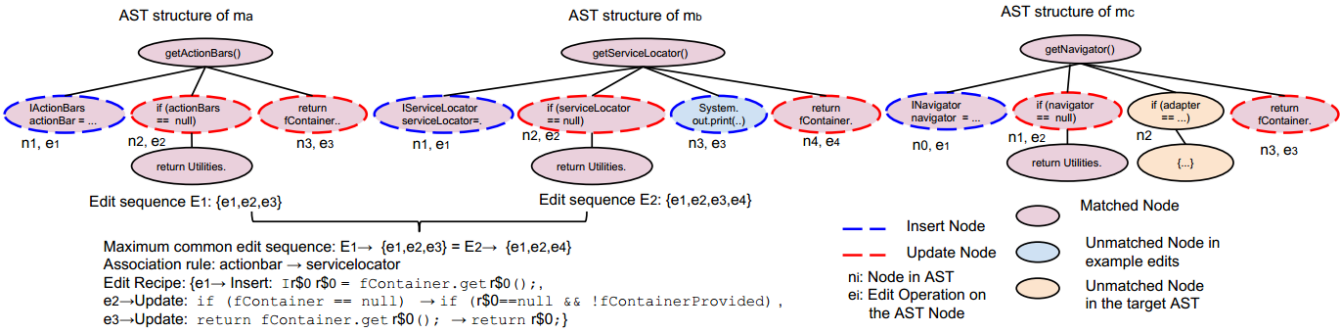

Figure 2: AST structure of m_a and m_b

clone detection algorithm ($m((tail(s), tail(t)))$).

$$MCESE(s,t) = \begin{cases} 0 \text{ if s or t is empty} \\ max \begin{cases} MCESE(head(s), head(t)) \text{ if } eq(s[0], t[0]) \\ +MCESE(tail(s), tail(t)) + m((tail(s), tail(t))), \\ MCESE(head(s)tail(s), t), \\ MCESE(s, head(t)tail(t)) \quad otherwise \end{cases} \end{cases}$$

s, t represent the compared AST trees. (2)

Here we define the statement match threshold as $t_s = 2$, that is, if the two statements are equivalent in type and have more than 2 matched nodes, they are regarded as matched in common embedded subtree. The maximum common embedded subtree serves as the context of the edit recipe. If only one example is specified to generate edit recipe, we include all nodes in this example's AST.

4.2 Recommend edit locations

When we try to recommend the similar edit locations that may need the same systematic edit, instead of setting the statement match threshold described in Section 4.1.4, we define the similarity of two AST trees s, t as the sum of matched tokens of each statement in the Maximum Common Embedded Subtree of s, t. Notice that the similarity varies with the number of nodes in the AST trees and we set a basic threshold t_s as 20 to filter out trivial similarity statements such as method declarations and return statements. CLIPBOARD sorts the similarity of each function from high to low based on the weight of AST similarity against the example. In Figure 2, the pink nodes illustrate the Maximum Common Embedded Subtree of m_a and m_b. The pink nodes in m_c's AST are relevant to the corresponding nodes in the common subtree.

4.3 Generate the substituted code with naming synthesis

Lastly, the edit operations with statement nodes in the edit recipe are consistently applied to the relevant nodes in the target AST. In Figure 2, when developer prefers to apply the edit recipe on the method m_c, the edit operations $\{e_1, e_2, e_3\}$ in the edit recipe are applied to the target tree correspondingly.

$$\begin{cases} e_1 \text{ Insert: } \texttt{IR\$0}r\texttt{\$0 = fContainer.get}r\texttt{\$0();}, \\ e_2 \text{ Update: } \texttt{if (fContainer == null)} \rightarrow \\ \texttt{if (}r\texttt{\$0 == null \&\& ! fContainerProvided)}, \\ e_3 \text{ Update: } \texttt{return fContainer.get}r\texttt{\$}_0\texttt{;} \rightarrow \texttt{return } r\texttt{\$}_0\texttt{;} \end{cases}$$

CLIPBOARD traverses the target AST twice in pre-order to finish the transformation.

For the first traversal, our tool starts from the root node of the target AST that matches with the root node in the edit recipe context, extracts the association rule `ActionBars` → `Navigator`, checks the edit operations in the edit recipe, and finds that a new node should be inserted as the first child of the root node based on edit operation e_1. CLIPBOARD generates a new node n_0 and inserts it into the corresponding location in the target AST, with the value of `IR$0` r`$0 = fContainer.get`r`$0();`. Our tool moves to the next node in the target AST and matches n_1 in m_c's AST with n_2 in the common subtree (or m_a's AST). The edit operation e_2 is related to the node n_2 in the example, so the edit e_2 is applied to the node n_1 in the target and update its value from `if (fContainer == null)` to `if` r`$0 == null && !fContainerProvided)`. The child of node n_1 in m_c's AST is mapped with the child of n_2 in m_a's AST, the association rule `actionBars`→`navigator` is extracted, and is confirmed based on the frequency threshold of the association rules.

Our tool then checks the node n_2 in m_c's AST which is not matched with the edit recipe context, neither do its children nodes. CLIPBOARD identifies next matched node as n_3 in m_c's AST with n_3 in the context of edit recipe, and applies the edit operation e_3 related to node n_3 in m_c's AST. Therefore, the value of n_3 in m_c's AST is updated from `return fContainer.getNavigator()` to `return` r`$`$_0$`;`. The association rule `actionBars` →`navigator` is extracted along with this update operation.

Notice that our tool does not require exact match in the context thus it is able to apply the systematic edit to the context that is quite different from the edit recipe, such as the example shown in Figure **??**. Our approach starts from the target AST, maximizes the opportunity to match the nodes with edit operations, and applies the change to all matched nodes.

During the second traversal of the target AST, based on the association rule `actionBars`→`navigator` and the value of the node n_1 after transformation, we replace all values of $r\$_0$ as `navigator` and generate the substituted code for the target edit location.

4.4 Speculative Analysis

After generating the code based on the edit recipe and the target, we display it as the preview of the transformation. If developer decides to make the transformation, we proactively replace the target with the target region and invoke the compilation error checker in Eclipse platform to evaluate the consequence of this transformation. If there is no error, the transformation is applied successful, yet if any problems are returned, CLIPBOARD calls the *undo manager* of the Eclipse platform to rollback the change, informs developers of the compilation errors. and let developer choose whether to apply the transformation by ignoring the errors or not.

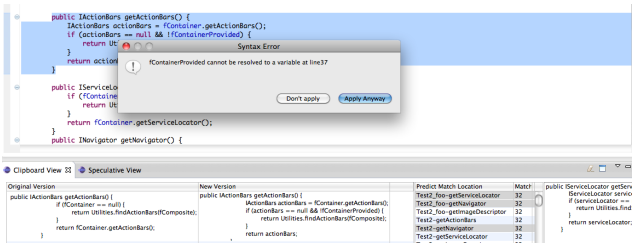

Figure 5: Screenshot of Error Dialog that proactively notifies developers with the compilation errors after the systematic edit

5. USER INTERFACE

5.1 Create and apply edit recipes via Clipboard View

We implement our approach via building an Eclipse Plugin call Clipboard. Figure 3 depicts the view of Clipboard. By selecting the code region and dragging it to the Clipboard, developers specifies the code fragment before making the systematic edit. If the selected region is part of a method, the method is identified while only the selected nodes are specified as the edit. On the other hand, if more than one functions are selected, Clipboard will generate a separate AST for each method and return the methods that map with any functions in the selected region.

Based on the old version of the example(s), Clipboard evaluates the similarity weight of each function in the same package and displays the recommended edit locations in the recommendation list.

Once developer finishes the systematic edit, he or she can take the snapshot again by selecting the program region and dragging it to Clipboard View again.

More than one example can be demonstrated as the input of the edit recipe to filter out trivial edits. Developers can select multiple examples from the Clipboard and generate the common systematic edit by selecting the item of *generate edit recipe* at the pop-up context menu triggered by the right click.

After the new version of the edit recipe is specified, Clipboard automatically generates the code after transformation based on the edit recipe and the top-ranked recommended edit location. The new version will be displayed as the preview of the transformation. Developers can select any items in the recommendation list and preview the code after transformation. If they want to preview the results of applying the edit recipe to multiple places, they can select multiple items from the recommendation list and select *Inspect all previews* at the context menu popped-up after right click. The Speculative view will be opened to display all previews with corresponding compilation errors invoked by the transformation. This speculative analysis is done by applying the edit recipe to one edit location at a time, checking the compilation error using the error checker in Eclipse, and calling the *undo manager* before trying the next edit recipe. The result is shown in Figure 4.

Finally, developers decide whether they prefer to perform the transformation by reviewing the potential compilation errors and the substituted code snippet after the edit. If there are errors caused by the systematic edit, Clipboard will pop up a warning dialog about the problems and let developers decide whether to carry on this change that will introduce error or not, as shown in Figure 5. If the change will not cause any compilation error, or the problems have already been approved by the developer, Clipboard replaces the original code snippet with the transformation code generated by the Clipboard to apply the systematic edit.

5.2 Apply the systematic edit on the fly

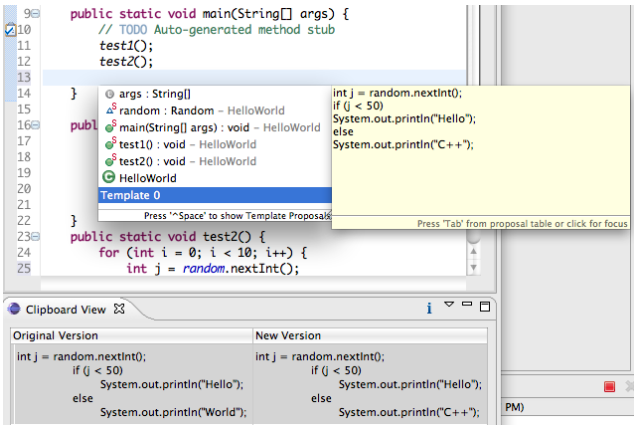

Figure 6: Screenshot of applying the systematic edit on the fly

Clipboard also monitors user's editing stream and matches the context with all edit recipes on the fly. It invokes the content assist engine (also known as code completion engine) in Eclipse to display the ranked candidate recipe list in a pop-up menu regarding the similarity against the current context. Shown in Figure 6, one of the edit recipes is selected from the candidate list. Our tool takes the chosen edit recipe and the current context as the input of systematic editing, displays the speculative analysis result and the preview of the code snippet after applying the edit recipe, and substitute the current context with the transformation code once developer decides to apply the systematic edit. The same as invoking the systematic edit with Clipboard View, if any compilation errors are detected by proactively applying the edit, our tool will notify the developer with the warning dialog.

6. EVALUATION

To assess the accuracy, efficiency, and usability of Clipboard, we use two oracle test suites drawn from Java open-

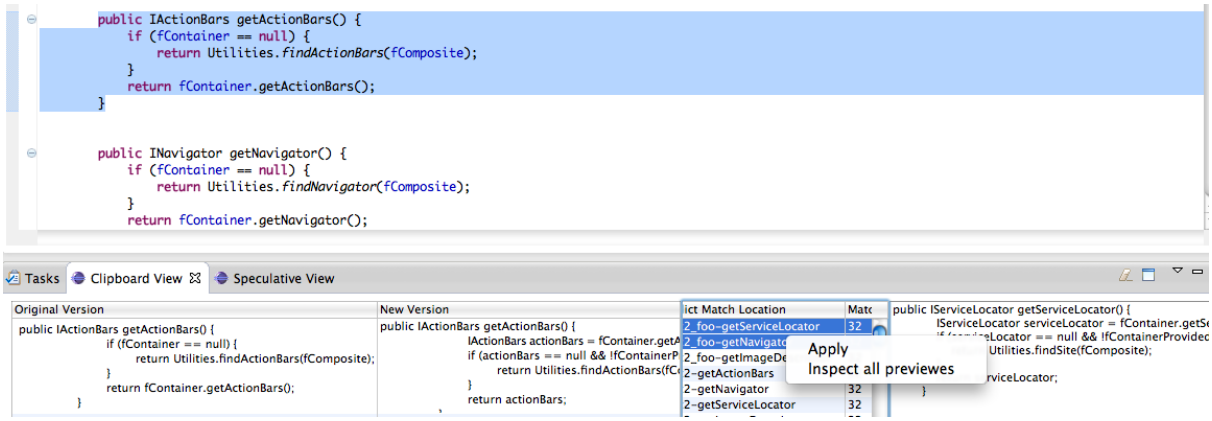

Figure 3: Screenshot of Clipboard View to generate edit recipe and apply the edit recipe

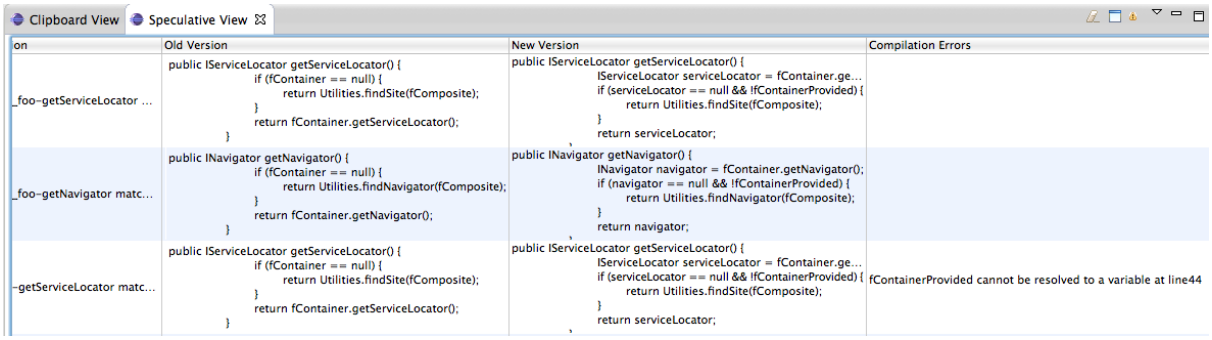

Figure 4: Screenshot of Speculative View to inspect all previews of the selected edit locations with corresponding compilation errors.

source programs. One test suite consists of 56 systematic edits from five Java programs (jEdit, Eclipse jdt.core, Eclipse compare, Eclipse core.runtime, and Eclipse debug). The other test suite has 22 systematic edits group with multiple systematic edit examples that fix the same bug in multiple commits from Eclipse JDT and Eclipse SWT, based on the work of supplementary bug fixes [45]. The supplementary bug fixes are defined as bug fixes that complete the fixing of one bug with multiple commits. The work illustrates that developers miss locations that needs to be changed when fixing a bug.

We evaluate whether Clipboard is able to recommend the edit locations that should be changed with the systematic edits, whether it is able to apply the same or similar transformation correctly compared to the manual changes in the code repository, whether our tool is able to give a reasonable name for the new initialized variables. We ask and answer the research questions below:

- What is the accuracy of our approach?
- How pertinent is it between the recommended edit locations and the edit recipe?
- How meaningful our recommended variable names are for users?
- How useable is drag-and-drop systematic edits with speculative analysis?
- How efficient is it to create, invoke, and perform the systematic edits?

We measure the accuracy and pertinence of recommended edit locations with a case study described in Section 6.1. To evaluate the usability, intuitiveness, and efficiency of our tool, we conduct a 45 minutes user study on X participants with 4 subtasks. We describe the user study in Section 6.2.

6.1 Initial case study

6.1.1 What is the accuracy of our approach?

We define accuracy as the percentage of correctly applied transformation tasks out of all tasks we created based on the test suites. For each example in the first test suite which only have 2 similar systematic edits, we arbitrarily select one of them as the inferred edit and perform the change to the other. We use the second test suite to evaluate the systematic edits whose edit recipes are generated via more than one example. We manually select 2 systematic edits and apply the generated edit recipe to other edit locations in the same systematic edit group. We then measure the accuracy of the transformation results against the manual change made by developers in the code repository.

Particularly, out of 56 systematic edits in the first test suite, two of them have non-continuous contexts and require the same edit to be applied to more than one locations in a single method. Figure **??** (from `org.eclipse.compare`: v20060714 vs. v20060917) shows one of these cases. Nei-

ther Sydit nor LASE is able to handle these cases. Since Clipboard traverses the target AST and maximizes the opportunity of matching potential edit locations, it is able to handle the case that the same change should be replicated to multiple locations in one AST.

6.1.2 How pertinent is it between the recommended edit locations and the edit recipe?

We measure the number of recommended edit locations (N_e) from the top ranked edit locations to the actual edit location in code repository. Notice that N_e will be influenced by the number of methods (n_m) Clipboard evaluated against the edit recipe, the number of AST nodes (n_t) in each evaluated method, and the number of AST nodes in the edit recipe (n_e).

6.2 User study

We ask X developers in the industry with Y years programming experience in Java or relevant programming languages to participant our blind experiment and user study. For each of them, we conduct 4 studies in a row:

1. A five minutes blind experiment with X naming pairs to evaluate the auto-generated names against manual-created names;
2. A ten minutes blind experiment with Y systematic edit examples to compare the auto transformation against manually systematic edits;
3. A thirty minutes user study with W systematic edits tasks to evaluate the efficiency, usability and intuitiveness of our tool
4. A 5 minutes questionnaire with Z questions to collect the feedback from end users.

Participants will not be informed of the purpose of the study until the user study. Between the blind experiments and the user study, we spend another five minutes explaining the functions of the Clipboard so that the participants will be able to use our tools and finish the tasks in the user study.

6.2.1 Blind experiment on naming synthesis

To answer the research question **How meaningful our recommended variable names are for users?**, we identify Z examples from the oracle test suites whose auto-generated identifier names are different from the manual-created ones while the transformation and context are almost the same. We use these Z examples for this blind experiment and shuffle the order of the real edits and the auto-generated transformation result in each example. Without telling the participants the purpose of our experiment, we ask them to select the version they prefer in each example pair. X% of developers choose the auto-generated names.

From the developers' comments, ...

6.2.2 Blind experiment on systematic edits

To understand the difference between automatically generated systematic edits and the manually changes, we conduct a blind experiment with X examples from the oracle test suites whose auto-generated edits are different from the manual edits. We shuffle the order of the manual edit and the automatic transformation result in each example and ask participants to choose the one they prefer in each example.

	1	2	3	4
A	(cb, E_1)	(sr, E_2)	(sr, E_3)	(cb, E_4)
B	(sr, E_1)	(cb, E_2)	(cb, E_3)	(sr, E_4)
C	(cb, E_3)	(sr, E_4)	(sr, E_1)	(cb, E_2)
D	(sr, E_3)	(cb, E_4)	(cb, E_1)	(sr, E_2)

Table 1: Task assignment table

Y% of participants prefer to use the transformation for the reason that ...

6.2.3 User study on the Clipboard

To measure the usability and intuitiveness of Clipboard, we ask participants to finish Z systematic tasks via Clipboard. Before conducting the study, we give a five minutes explanation to the participants and help them understand how to use Clipboard. The participants spend X minutes on average to finish all transformation tasks, and ... We compare Clipboard with common *search and replace* text editor functionality and demonstrate that our approach is more effective and robust.

We record the average time of predicting the edit locations, generate transformation code based on the edit recipe and target code, apply the change to the target without invoking compilation error, and the time spent on manual editing by *search-and-replace* practice. We create 24 tasks and separate them into 4 groups (E_1, E_2, E_3, E_4) with an almost equal length of context, equal number of edit operations, and equal number of relevant functions and variables. We also separate our participants into four groups (A, B, C, D) who have almost equal experience in programming and do not know the content of the tasks before the case study. As shown in Table 1, we assign the task group E_i to the group X at the time slot n ($n = 1, 2, 3, 4$) with the approach of either using Clipboard (cb) or using *search-and-replace* (sr) practice.

We record the time the participants spent on the tasks and compare the transformation results using Clipboard and using traditional *search-and-replace* practice to manually apply the changes.

6.2.4 Questionnaire to collect feedback

To assess the utility of our tool and collect feedback from participants, we design a questionnaire with X questions. We ask them to ..., X% of participants believe that Clipboard is useful and more reliable compared to the *search-and-replace* practice and any other source code editors such as sublime [18].

In summary, the evaluation shows that Clipboard is more efficient than traditional *search-and-replace* practice for the systematic edits. It can effective eliminate the omission errors and inconsistent systematic edits. And it is equally intuitive compared to the manual editing. The automatically generated names of the variables that do not exist in the old version is considered to be more meaningful compared to the manual generated variable names as developers might tend to use abbreviations rather than the full names while full names improve the comprehension of the program. With the transformation preview, speculative analysis, and meaningful naming synthesis, developers are willing to use our tool to conduct systematic edits automatically.

7. THREATS TO VALIDITY

External Validity Our tool is confined to the Eclipse that supports Java only. Yet we believe that the transformation algorithm and the approach to synthesize the identifier names can be generalized to other object-oriented languages in a different Integrated Development Environment.

8. DISCUSSION AND FUTURE WORK

By applying the systematic edits to multiple places might lead to some redundant code that can be extracted as a common method and reuse it in the future. We believe more work can be done in extracting the common context to a shared method for opportunistic refactoring. How to automatically generate a meaningful method name for the extracted method remains challenging. We synthesize the identifier name based on the association rules of relevant variables. It is also possible to recognize the naming pattern from the type of the variable, such as `ServiceLocator` and `serviceLocator`. More work can be done to further synthesize the method name based on the context and refine the approach for the naming synthesis with constraint solver.

9. CONCLUSION

In this paper, we present a tool for systematic edits using edit recipes learned from examples. We describe our approach to construct an edit recipe, identify similar edit locations, and apply the transformation on the target with speculative analysis. Our evaluation shows that Clipboard effectively eliminates the tedious invoking and configuration process for the program transformation with an intuitive drag-and-drop approach to demonstrate the examples for the systematic edits. Our case study on two oracle test suites illustrates that our approach is able to reduce the omission errors and inconsistent changes. It is more efficient and robust compared to the traditional *search-and-replace* systematic edit practice. Our user study shows that Clipboard is able to provide meaningful names for the new variables which do not exist before the transformation. The changes the Clipboard made is as intuitive as the manual edits while introducing less omission errors. Developers are more confident to perform the systematic edits automatically with the transformation preview and speculative analysis results provided by Clipboard.

In summary, our approach can effective inform developers of the potential edit operations that may require the same systematic edits, synthesize the identifier names which do not exist before transformation, and provide speculative analysis and transformation preview to assist developer for the program transformation.

10. REFERENCES

[1] G. W. Adamson and J. Boreham. The use of an association measure based on character structure to identify semantically related pairs of words and document titles. *Information Storage and Retrieval*, 10(7-8):253–260, 1974.

[2] J. Andersen and J. L.Lawall. Generic patch inference. *ASE 2008*, pages 337–346, 2008.

[3] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo. Semantic patch inference. *ASE 2012*, pages 382–385, 2012.

[4] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Mining java class naming conventions. In *ICSM*, pages 93–102, 2011.

[5] B. Caprile and P. Tonella. Restructuring program identifier names. In *ICSM*, pages 97–107, 2000.

[6] J. R. Cordy. Exploring large-scale system similarity using incremental clone detection and live scatterplots. *ICPC 2011*, pages 151–160, 2011.

[7] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE*, pages 481–490, 2008.

[8] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *ICSM*, pages 169–178, 2009.

[9] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *ICSE*, pages 158–167, 2007.

[10] B. Fluri, M. Würsch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007.

[11] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *OOPSLA*, pages 175–190, 2010.

[12] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *OOPSLA*, pages 175–190, 2010.

[13] J. Galenson, P. Reames, R. Bodík, B. Hartmann, and K. Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *ICSE*, 2014.

[14] X. Ge, Q. L.DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. *ICSE 2012*, pages 211–221, 2012.

[15] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[16] E. W. Høst and B. M. Østvold. Debugging method names. In *ECOOP*, pages 294–317, 2009.

[17] http://www.eclipse.org/recommenders/manual/.

[18] http://www.sublimetext.com/.

[19] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[20] J. Jacobellis, N. Meng, and M. Kim. Cookbook: In situ code completion using edit recipes learned from examples. 2014.

[21] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *ESEC/SIGSOFT FSE*, pages 55–64, 2007.

[22] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. Minthint: Automated synthesis of repair hints. *CoRR*, abs/1306.1286, 2013.

[23] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng.*, 28(7):654–670, 2002.

[24] Y. Kashiwabara, Y. Onizuka, T. Ishio, Y. Hayase, T. Yamamoto, and K. Inoue. Recommending verbs for rename method using association rule mining. In

CSMR-WCRE, pages 323–327, 2014.

[25] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *ICPC*, pages 3–12, 2006.

[26] Y. Y. Lee, N. Chen, and R. E.Johnson. Drag-and-drop refactoring: Intuitive and efficient program transformation. *ICSE 2013*, pages 23–32, 2013.

[27] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Software Eng.*, 32(3):176–192, 2006.

[28] H. Liberman. Your wish is my command: Programming by example. In *Morgan Kaufmann Publisher*, 2001.

[29] A. Lozano and G. Valiente. On the maximum common embedded subtree problem for ordered trees. In *In C. Iliopoulos and T Lecroq, editors, String Algorithmics, chapter 7. KingÕs College London Publications*, 2004.

[30] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, pages 48–61, 2005.

[31] N. Meng, M. Kim, and K. S.McKinley. Systematic editing: Generating program transformations from an example. *PLDI 2011*, pages 329–342, 2011.

[32] N. Meng, M. Kim, and K. S.McKinley. Lase: Locating and applying systematic edits by learning from examples. *ICSE 2013*, pages 502–511, 2013.

[33] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX Annual Technical Conference, General Track*, pages 161–174, 2001.

[34] R. C. Miller and B. A. Myers. Lapis: smart editing with text structure. In *CHI Extended Abstracts*, pages 496–497, 2002.

[35] R. C. Miller and B. A. Myers. Multiple selections in smart text editing. In *IUI*, pages 103–110, 2002.

[36] K. Muslu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Improving ide recommendations by considering global implications of existing recommendations. *ICSE 2012*, pages 1349–1352, 2012.

[37] K. Muslu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis of integrated development environment recommendations. *OOPSLA 2012*, pages 669–682, 2012.

[38] K. Muslu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Making offline analyses continuous. *FSE 2013*, pages 323–333, 2013.

[39] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *ICSE*, 2014.

[40] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N.Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. *ICSE 2012*, pages 69–79, 2012.

[41] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. *ASE 2013*, 2013.

[42] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to api usage adaptation. *OOPSLA 2010*, pages 302–321, 2010.

[43] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In *ICSE*, pages 772–781, 2013.

[44] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Cleman: Comprehensive clone group evolution management. In *ASE*, pages 451–454, 2008.

[45] J. Park, M. Kim, B. Ray, and D.-H. Bae. An empirical study of supplementary bug fixes. In *MSR*, pages 40–49, 2012.

[46] B. Ray, M. Kim, S. Person, and N. Rungta. Detecting and characterizing semantic inconsistencies in ported code. In *ASE*, pages 367–377, 2013.

[47] X. Ren, B. G. Ryder, M. Störzer, and F. Tip. Chianti: a change impact analysis tool for java programs. In *ICSE*, pages 664–665, 2005.

[48] S. R.Foster, W. G.Griswold, and S. Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. *ICSE 2012*, pages 222–232, 2012.

[49] N. Schwarz. Hot clones: Combining search-driven development, clone management, and code provenance. In *ICSE*, pages 1628–1629, 2012.

[50] J. Singer and C. C. Kirkham. Exploiting the correspondence between micro patterns and class names. In *SCAM*, pages 67–76, 2008.

[51] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *VL/HCC*, pages 173–180, 2004.

[52] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei. Can i clone this piece of code here? In *ASE*, pages 170–179, 2012.

[53] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical api usage. In *ICSE*, pages 826–836, 2012.

[54] G. Zhang, X. Peng, Z. Xing, S. Jiang, H. Wang, and W. Zhao. Towards contextual and on-demand code clone management by continuous monitoring. In *ASE*, pages 497–507, 2013.