Reconciling Manual and Automatic Refactoring

Xi Ge Quinton L. DuBose Emerson Murphy-Hill
Department of Computer Science, North Carolina State University, Raleigh, NC
{xge, qldubose}@ncsu.edu, emerson@csc.ncsu.edu

***Abstract*—Although useful and widely available, refactoring tools are underused. One cause of this underuse is that a developer sometimes fails to recognize that she is going to refactor before she begins manually refactoring. To address this issue, we conducted a formative study of developers' manual refactoring process, suggesting that developers' reliance on "chasing error messages" when manually refactoring is an error-prone manual refactoring strategy. Additionally, our study distilled a set of manual refactoring workflow patterns. Using these patterns, we designed a novel refactoring tool called BeneFactor. BeneFactor detects a developer's manual refactoring, reminds her that automatic refactoring is available, and can complete her refactoring automatically. By alleviating the burden of recognizing manual refactoring, BeneFactor is designed to help solve the refactoring tool underuse problem.**

I. INTRODUCTION

Software is expensive to maintain. As software ages, it must be maintained to reflect evolving user requirements; this maintenance accounts for between 40 and 75% of the total cost of software [7]. One of the primary ways that software developers maintain software is through *refactoring*, the process of changing the structure of code without changing the way that it behaves [8].

Refactoring is both an effective and commonplace way of improving non-functional requirements. Empirical studies of refactoring have found that it can improve maintainability [12] and reusability [14]. Not only does existing work suggest that refactoring is useful, but it also suggests that refactoring is a frequent practice [17]. Cherubini and colleagues' survey indicates that developers rate the importance of refactoring as equal to or greater than that of understanding code and producing documentation [3].

However, refactoring by hand has long been assumed to be error-prone. In order to help developers perform efficient and correct refactoring, various refactoring tools have been developed. These tools promise to help developers refactor faster and with a smaller probability of introducing defects. Refactoring tools have been integrated into most popular development environments, making them available in a variety of programming languages to a large population of developers. Despite the wide availability, our previous work shows that refactoring tools are underused; according to two case studies, about 90% of refactorings are performed by hand [17].

Several solutions have been proposed to solve the underuse problem. For example, tools with improved user interfaces can make refactoring tools more usable [15]. Other novel tools have supported new, specialized types of refactoring [1][4]. Other research has proposed automatic testing to make refactoring tools more reliable [5][6].

Such solutions make one common assumption: That the software developer recognizes that she is going to refactor before she even begins. This assumption is false when a developer has already started a refactoring manually by the time she realizes that she is refactoring. One developer outlined this situation in an interview [17] as:

> *I already know exactly how I want the code to look like. Because of that, my hands start doing copy-paste and the simple editing without my active control. After a few seconds, I realize that this would have been easier to do with a refactoring [tool]. But since I already started performing it manually, I just finish it and continue.*

This situation illustrates how refactoring tools do not support the developer when she does not realize she is refactoring until after she has already begun. Without that realization, a software developer will not use any the refactoring tool, no matter how usable, useful, or reliable that tool is.

In this paper, we investigate how to design a refactoring tool that is aware of a developer's refactoring, rather than relying on the developer's recognition of it. In Section III we describe a formative study about developers' manual refactoring. Building on the study's results, in Section IV we designed a novel refactoring tool. We make the following major contributions in this paper:

- A formative study of developers' manual refactoring. To the best of our knowledge, we are the first to study developers' manual refactoring process. Our study suggests that reliance on compiler errors when manually refactoring is a common and error-prone technique.
- A proof-of-concept refactoring tool called BeneFactor. In addition to relieving the developer from the burden of recognizing that she is going to refactor, BeneFactor also allows implicit refactoring configuration and the interleaving of refactoring and non-refactoring changes.

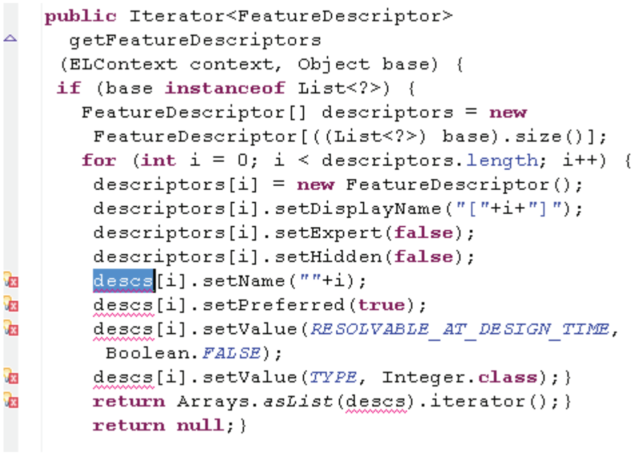
978-1-4673-1067-3/12/$31.00 © 2012 IEEE

ICSE 2012, Zurich, Switzerland

II. Motivation

To use a refactoring tool, a developer must recognize that she is refactoring and select the appropriate refactoring with a menu or a hotkey. If she is unaware that she is refactoring and begins to refactor manually, she may become aware that she is refactoring part way through the manual refactoring. In this case, the developer faces the *late awareness* dilemma: She must either undo all the code changes that she has already made and then invoke the refactoring tool, or keep refactoring manually until the refactoring is complete.

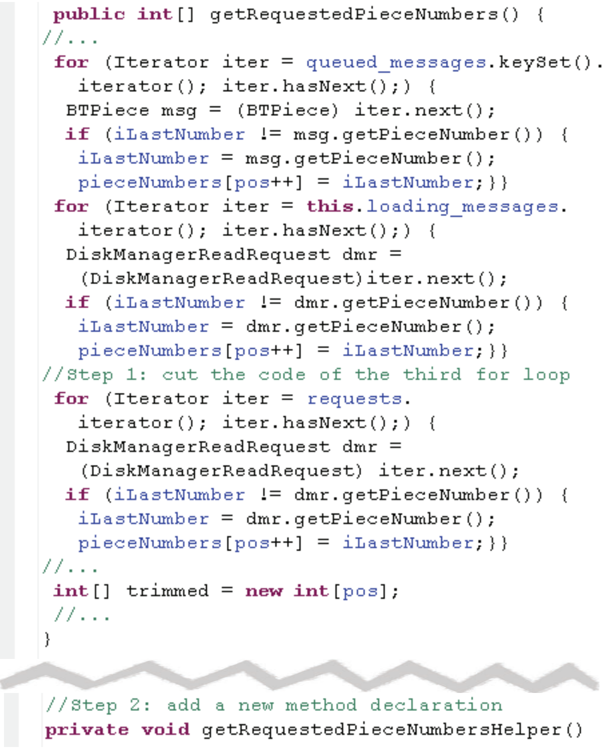
We next illustrate this dilemma using two examples. Suppose Grace is a developer who works on the Apache Tomcat open source project [2]. To improve understandability, she wants to change the name of a local variable *descs* to *descriptors*. She starts doing this task manually from the declaration of *descs* towards the tenth and last reference to *descs*. After changing six names (as shown in Figure 1) she realizes that she is performing a rename refactoring. Grace decides it would be easier to just finish the refactoring manually, even at the risk of introducing a bug.



```
public Iterator<FeatureDescriptor>
  getFeatureDescriptors
 (ELContext context, Object base) {
 if (base instanceof List<?>) {
   FeatureDescriptor[] descriptors = new
    FeatureDescriptor[((List<?>) base).size()];
   for (int i = 0; i < descriptors.length; i++) {
    descriptors[i] = new FeatureDescriptor();
    descriptors[i].setDisplayName("["+i+"]");
    descriptors[i].setExpert(false);
    descriptors[i].setHidden(false);
    descs[i].setName(""+i);
    descs[i].setPreferred(true);
    descs[i].setValue(RESOLVABLE_AT_DESIGN_TIME,
     Boolean.FALSE);
    descs[i].setValue(TYPE, Integer.class);}
    return Arrays.asList(descs).iterator();}
    return null;}
```

Figure 1: Rename local variable refactoring example.

Late awareness of refactoring occurs for other refactoring types as well. Suppose that Glen is a developer who works on the Vuze project [25]. He notices that the *getRequestedPieceNumbers()* method shown in Figure 2 contains three *for* loops that are nearly identical. To enhance maintainability, he intends to extract the loop as a new helper method. Glen starts by cutting the code in the third loop, and then declares the helper method *getRequestedPieceNumbersHelper()*. Glen then realizes that he is performing the extract method refactoring, and that he would like to have a refactoring tool figure out what variables to pass in to the method. To do so, he undoes his changes and invokes the extract method tool, even though he has to configure the refactoring tool with the name of the helper method, which he already specified once while manually refactoring.



```
 public int[] getRequestedPieceNumbers() {
//...
 for (Iterator iter = queued_messages.keySet().
   iterator(); iter.hasNext();) {
  BTPiece msg = (BTPiece) iter.next();
  if (iLastNumber != msg.getPieceNumber()) {
   iLastNumber = msg.getPieceNumber();
   pieceNumbers[pos++] = iLastNumber;}}
 for (Iterator iter = this.loading_messages.
   iterator(); iter.hasNext();) {
  DiskManagerReadRequest dmr =
   (DiskManagerReadRequest)iter.next();
  if (iLastNumber != dmr.getPieceNumber()) {
   iLastNumber = dmr.getPieceNumber();
   pieceNumbers[pos++] = iLastNumber;}}
//Step 1: cut the code of the third for loop
 for (Iterator iter = requests.
   iterator(); iter.hasNext();) {
  DiskManagerReadRequest dmr =
   (DiskManagerReadRequest) iter.next();
  if (iLastNumber != dmr.getPieceNumber()) {
   iLastNumber = dmr.getPieceNumber();
   pieceNumbers[pos++] = iLastNumber;}}
//...
 int[] trimmed = new int[pos];
 //...
}
```

```
//Step 2: add a new method declaration
private void getRequestedPieceNumbersHelper()
```

Figure 2: Extract method refactoring example.

III. Formative Study

In order to resolve the late awareness dilemma, we intend to build a novel refactoring tool that can complete manual refactorings. However, three important research questions must be answered first:

- RQ1. *How correctly do developers refactor manually?* If developers mistakenly modify program behavior when refactoring manually, then refactoring tools can potentially improve the refactoring process.
- RQ2. *How significant is the late awareness problem?* If many developers do not recognize that they are refactoring before they begin, then the late awareness is a contributor to refactoring tool underuse.
- RQ3. *What are developers' manual refactoring workflows?* To create a refactoring tool that can complete refactorings automatically, the tool must be able to recognize when a developer is refactoring. Models of manual refactoring workflows will help our tool recognize when a developer is refactoring.

We conducted a formative study to answer these questions.

A. Participants and Refactorings

We recruited 12 developers to participate in our study. Six were graduate students and five were commercial software developers. We also recruited one refactoring researcher (not one of the authors) who has current development experience of more than ten years. Although we did not collect all of the participants' demographic information, at least 10 of the 12 participants had professional programming experience of more than four years.

We asked participants to perform refactorings in the source code of the Vuze [25] project. We chose this project is for its large size and maturity.

We manually inspected the source code of Vuze, then located fourteen locations where refactoring could reasonably be performed. At each of the fourteen locations, we asked participants to perform one refactoring without the assistance of any refactoring tools. Among the fourteen refactorings, we chose three that were especially complex: A rename local variable refactoring, an extract method refactoring and a change method signature refactoring. When performing these complex refactorings, participants needed to carefully consider how to avoid changing program behavior. We describe the complex refactorings in detail in Section III-C1.

The fourteen refactorings spanned 8 refactoring types: Rename (2), extract constant (1), extract local variable (2), inline local variable (2), change method signature (2), extract method (2), introduce parameter (2), and pull up method (1). We chose these types from the types listed in Fowler's catalog [8] according to three criteria. The first is their frequency in real world. According to our previous research, using automatic tools to perform rename, extract local variable, inline, extract method, and change method signature accounts for over 85% of all refactoring tool usage [17]. This high tool usage suggests that these refactoring types occur frequently. Simplicity is the second criterion because simple refactorings would be easier for participants to complete in a short study session. The third criterion is wide coverage of a variety of software entities, such as constants, temporary variables, local variables, fields, and methods. We also chose pull up method because it involved class inheritance.

B. Data Collected

We collected data using a pre-study questionnaire, videos of participants' manual refactorings, and a post-study questionnaire. To view the participants' refactoring videos and their answers to the questionnaires, the reader can refer to our study material website.[1]

1) Pre-study Questionnaires: We recruited developers through email. If they were willing, we sent them a consent form and pre-study questionnaire. One subject consented, but did not fill out the remainder of the questionnaire. We asked the following questions in the questionnaire:

- Coding experience: About what percentage of your job involves writing code?
- Java proficiency: How proficient are you with the Java programming language, rating from 1 to 5 (1 for not at all and 5 for expert)? Participants produced a slightly skewed distribution of Java experience (median=3.5), with all participants reporting at least some experience and two participants considering themselves experts.
- Refactoring familiarity: How familiar are you with the practice of refactoring, rating from 1 to 5 (1 for "not at all" and 5 for "I refactor every time I program")? Participants produced a fairly normal distribution of refactoring experience (median=3).

To perform the study, participants connected to a remote computer running the Eclipse IDE and containing the code to be refactored. Using the online screen sharing service join.me [10], participants could easily view and operate Eclipse. At the same time, we established a Skype [21] conversation with the participant to give them directions.

2) Observing Manual Refactoring: After setting up screen sharing, we opened the first code location where we wanted the participant to perform a refactoring. We told the participant which refactoring we wanted her to perform and that no automatic refactoring tools were allowed. If the participant was unfamiliar with the refactoring, we gave her an explanation of what the resulting code should look like. We were careful to avoid telling the participant how to perform each refactoring. Once we had answered all of the participants' questions about the task, we asked her to start.

We repeated this process with all 14 refactorings. While the participant was performing these refactorings, we used screen capture software to record the entire process, which we analyzed after the study session ended.

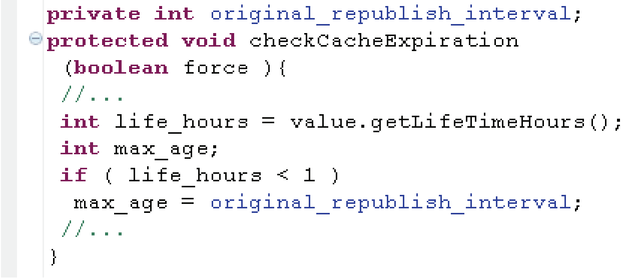
To analyze each refactoring, we tagged it as either "correct," "incorrect," or "unknown." "Correct" meant that the participant's refactoring resulted in the code structure that met with Fowler's definition [8] and did not modify the software's behavior. "Incorrect" indicated that the software's behavior was modified, but the participant's refactoring resulted in a code structure that met with Fowler's definition. Refactorings tagged as "unknown" included those that the participant skipped; those that were finished only with our detailed guidance (we guided some participants to spare them the embarrassment of not being able to complete the task); those that were finished by invoking refactoring tools; and those that we were not able to tell whether the new code structure met with Fowler's definition. Only refactorings tagged as "correct" and "incorrect" were used in our analysis of refactoring workflows.

3) Post-study Questionnaire: After the study, we presented each participant with a questionnaire that asked the following questions:

- Q1. How often does this situation occur: I get part way through a code change when I realize there is a refactoring tool that can help me do the job. Choose from one of the following words: Never, rarely, sometimes, often, and always.

---

[1] https://sites.google.com/site/refactoringstudy/



```
private int original_republish_interval;
protected void checkCacheExpiration
 (boolean force ){
 //...
 int life_hours = value.getLifeTimeHours();
 int max_age;
 if ( life_hours < 1 )
  max_age = original_republish_interval;
 //...
}
```

Figure 3: Code illustrating a complex refactoring.

- Q2. What would you do after the situation in Q1 happens? Options: (1) I finish the change without a refactoring tool; (2) Back out of the change and redo the change using a refactoring tool; or (3) Other.

C. Results

1) RQ1. Refactoring Correctness: Our previous research suggests that developers may rely on compilation errors to locate the related code to update [17]. For example, in Figure 1, Grace is using the compilation errors to determine what parts of the code needs to be updated. However, this strategy is sometimes error prone because compilation errors do not indicate every location that needs to be updated for certain complex refactorings. We had participants perform three of these complex refactorings to determine to what extent they used this strategy. Overall, participants inadvertently changed behavior in eleven of the fourteen refactorings, and very few finished participants completed complex refactorings them correctly.

Complex Refactoring 1: We asked participants to perform the complex extract method refactoring shown in Figure 2. If a developer extracts the last *for* loop into a new method, the new method should return the value of *pos* because the extracted *for* loop modifies the value of *pos* and the code in the original method later reads the value. However, compilation errors do not result if *pos* is not returned, so relying on error messages for this part of the refactoring is not sufficient for a correct refactoring.

Only one participant correctly performed the complex extract method refactoring, while seven performed it incorrectly because of failing to return the value.

Complex Refactoring 2: We asked participants to perform a rename local variable refactoring on code that is summarized in Figure 3. We asked the participants to rename *life_hours* to *original_republish_interval*, which is also the name of a field of the containing class. If a developer renames *life_hours* to *original_republish_interval* in the declaration, names in *checkCacheExpiration(...)* that originally bound to the field *original_republish_interval* now bind to the local variable. No compilation error is generated when the rebinding occurs.

Table I: Questionnaire results.

Pre-study questionnaire			Post-study questionnaire			
Coding experience	Java proficiency	Refactoring familiarity	Late awareness	Handle late awareness		
				Manual	Tool	Others
80%	4	4	sometimes	100%	0%	0%
90%	4	3	sometimes	100%	0%	0%
40%	3	2	often	80%	20%	0%
75%	5	3	sometimes	65%	35%	0%
100%	4	4	rarely	80%	20%	0%
70%	5	5	often	40%	30%	30%

No participant correctly performed complex rename local variable refactoring, while nine performed it incorrectly.

Complex Refactoring 3: We asked participants to perform a change a method signature refactoring by swapping the order of two parameters. More specifically, we asked them to refactor *setUserData(Object key, Object value)* to *setUserData(Object value, Object key)*). If a developer swaps these parameters in the method declaration, no compiler errors are shown because all invocations of this method still had arguments of correct types.

One participant performed this complex refactoring correctly, while two performed it incorrectly.

Summary: Across the three complex refactorings, 90% were performed incorrectly. It appeared that developers relied on compiler errors for refactoring, even in situations where that reliance was misplaced. Somewhat surprisingly, participants also made mistakes in the non-complex refactorings. In total, participants completed 96 non-complex refactorings, and 21 (22%) of them were incorrect. Most of these incorrect refactorings resulted from participants' failing to address all compiler errors.

In addition to these results, we also observed another problematic manual refactoring technique. When performing the rename field refactoring, four of the eleven participants invoked the "find and replace all" tool, replacing all of the occurrences of the original field name in the file. Although this technique did not happen to modify behavior in the given code, this technique is generally unsafe because the original name may also occur in other places not referring to the field, such as in a method name.

2) RQ2. Late Awareness: How significant is the late awareness problem? We answer this question by using participants' post-study questionnaires.

Among the 12 participants recruited in our formative study, six returned their post-study questionnaires. Table I shows the answers of these participants to each question we asked.[2] Five of the six participants indicated that late awareness of refactoring tools happens to them at least "sometimes." Two of these five participants indicated that

---

[2] The first row shows the refactoring researcher's answers.

late awareness of refactoring happens to them "often". This suggests that the late awareness problem may happen to a variety of programmers.

Participants reported that when late awareness occurs, a median of 80% of the time they finish the refactoring manually. One participant indicated that she handles late awareness through other means, but did not elaborate.

3) RQ3. Refactoring Workflow Patterns: What are the developers' manual refactoring workflows? To investigate this question, we studied the videos of participants performing the refactorings. We distilled a set of widely adopted refactoring workflows for each refactoring type, which we refer to as refactoring workflow patterns. We modeled these patterns using finite-state machines (FSMs) over a set of parameterized elementary operations to facilitate representation and interpretation.

We use operations on abstract syntax tree (AST) nodes to model refactoring workflow patterns. ASTs are tree representations of the syntactic structure of source code written in a programming language. Nodes inside an AST represent software entities at various levels, such as variables, fields, statements, methods, and classes. We use the following operations on the AST nodes to model our refactoring workflow patterns:

- $COP(x)$; copy node x's source code to the clipboard.
- $CUT(x)$; cut node x's source code and keep it in the clipboard.
- $INS(x)$; insert node x into the AST, possibly via a Paste command.
- $UPD(x)$; update the value of node x.

Inspired by regular expressions, we also define the following quantifiers:

- $OP(x*)$ indicates performing operation OP on zero or more nodes x simultaneously.
- $OP(x)*$ indicates performing OP on zero or more nodes x sequentially.
- $OP(x+)$ indicates performing operation OP on one or more nodes x simultaneously.
- $OP(x)+$ indicates performing OP on one or more nodes x sequentially.

Due to space limitations, we present the refactoring workflow patterns for only rename field and extract method, two common refactorings [17]. For refactoring workflow patterns for other refactoring types, the reader can refer to our study material website mentioned in Section III-B.

Rename Field: Participants had similar workflows when manually performing the rename field refactoring. Eleven rename field refactorings that were considered usable (that is, correct and incorrect). Each participant used one of two patterns:

- Seven participants first updated the name of the field in its declaration, and then iteratively updated the names of all the references to this field. We modeled this workflow by the upper transitions (red) in Figure 4. In the Figure, the number on a transition indicates the number of participants following that transition, while the square brackets indicate a conditional transition.
- Four participants invoked the "find and replace" tool to automatically replace all of the occurrences of the field's name. We modeled this workflow by the lower transition (blue) in the Figure 4.

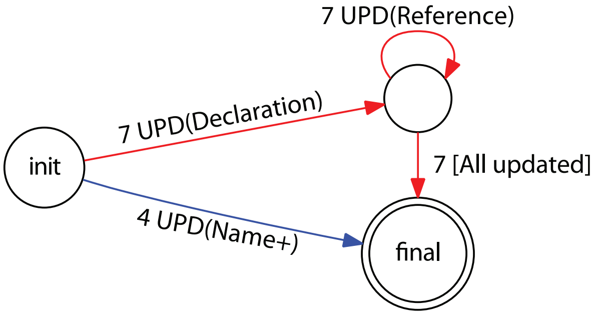


Figure 4: Refactoring patterns for rename field.

Extract Method: We collected ten usable extract method refactorings. Participants adopted several workflows when performing extract method. Three workflows were common to multiple participants:

- Two participants first copied the statements to extract, made a new method declaration, pasted the statements into the new method body, added parameters to the new method declaration, and finally replaced the statements to extract with the new method invocation. We modeled this workflow by the upper transitions (red) in Figure 5.
- Two participants first cut the statements to extract, made a new method declaration, pasted the statements into the new method body, added parameters to the new method declaration, and finally inserted the new method invocation to the place where the extracted statements were. We modeled this workflow by the middle transitions (blue) in Figure 5.
- Two participants first added a new method invocation near the statements to extract, cut the statements to extract, made the new method's declaration, pasted the statement into the new method body, and finally added parameters to the new method declaration. We modeled this workflow by the lower transitions (green) in Figure 5.

We also observed the other four patterns (not shown in Figure 5) adopted by one participant each:

- One participant first added a new method declaration after the method of the statements to extract, copied the statements to extract, pasted these statements into the new method body, replaced these statements with the

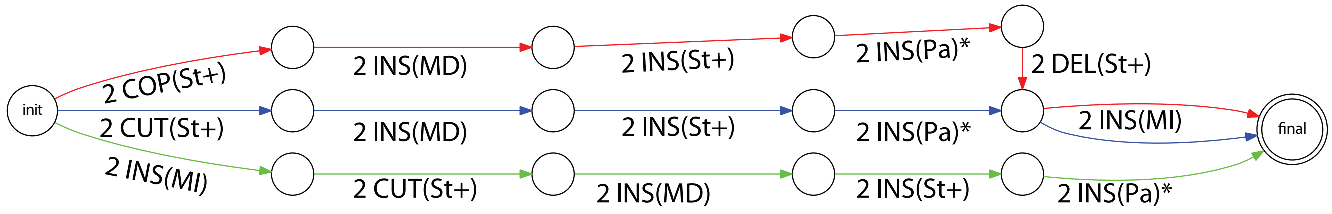


Figure 5: Refactoring patterns for extract method.

new method's invocation, and finally added parameters to the new method declaration.

- One participant first cut the statements to extract, added a new method invocation at these statements' original place, added a method declaration after the statements' original method, pasted the statements to extract into the new method body, and finally add parameters to the new method declaration.
- One participant first cut the statements to extract, pasted these statements after their original method, surrounded these statements by a new method name and brackets, added parameters to the new method declaration, and finally inserted the new method invocation at the extracted statements' original place.
- One participant first copied the statements to extract, added a method declaration below these statements (inside their containing method), pasted the statements into the new method declaration, cut the new method declaration, pasted the new method declaration after the method containing the statements to extract, added parameters to the new method declaration, and finally replaced the statements to extract with the invocation of the new method.

These patterns suggest that developers manually refactor using only a few different basic actions, but interleave those actions in a variety of ways.

D. Threats to Validity

Although our formative study provides data on how developers manually refactor, there are several threats to validity that should be considered when interpreting our results.

The first threat is the criteria we used to choose refactoring types. We chose the refactoring types whose tools are used more frequently, yet these types of refactorings were not always the most *underused* refactoring tools. In the future, we would like to perform the study with refactoring types for which the corresponding tool is the most underused.

Another threat is the number of participants in our study, and that not all participants produced usable data for every refactoring. Thus, we cannot make strong claims about how our results generalize to other developers. For example, some additional refactoring patterns may exist that were not exhibited by our participants. However, we believe that the results provide a sufficient starting point to begin building a proof-of-concept refactoring tool, as described in Section IV. Similarly, the selected refactorings may not be representative of all refactorings performed in the wild.

Another significant threat is that participants' attitude toward refactoring third-party software may be different than their attitude towards refactoring their own software. Specifically, because participants have no investment in the third-party code, they may be less concerned about introducing behavior-modifying changes, as compared to their own code. This would cause our study to overestimate how often refactoring errors are made. It is also possible that participants exercised more diligence in refactoring correctly in the study, since they were not distracted by other coding tasks. This would cause our study to underestimate how often refactoring errors are made. In either case, a field study may be able to shed more light onto the frequency of errors made during manual refactoring.

IV. APPROACH

According to our formative study, manual refactoring can be an error-prone task (RQ1). Although conventional refactoring tools are available to assist the developers performing safe and efficient refactoring, they are significantly underused [17]. This underuse problem partially results from the developers' late awareness of refactoring (RQ2). In order to tackle this problem, we propose a novel refactoring tool called BeneFactor that is built on our distilled refactoring workflow patterns (RQ3). Unlike conventional refactoring tools, BeneFactor automatically detects an ongoing manual refactoring, reminds the developers that automatic refactoring is available, and can finish the manual refactoring after the developer's explicit invocation without requiring her to undo any code changes. Implemented as a plug-in for the Eclipse IDE, a prototype of BeneFactor can be downloaded here: http://code.google.com/p/flexible-refactoring-tools/. BeneFactor has two major components: Refactoring detection and code modification.

A. Refactoring Detection

In order to assist the developer in recognizing that she is refactoring, we designed a refactoring detection component. In contrast to existing refactoring detection tools (for example, RefacLib [22] and REF-FINDER [18]), our refactoring detection component detects refactoring while the developer is programming, rather than in the version control system. Moreover, BeneFactor detects ongoing and unfinished refactorings, rather than completed ones.

The refactoring detection component runs in the background of the IDE and continuously captures the developer's operations on the code base. It collects two kinds of operations:

- Code-change based operations are detected by comparing subsequent snapshots of code base. These snapshots are captured after certain developer events, such as adding a statement, deleting a field declaration, and updating a variable name.
- Action-based operations are detected by monitoring the commands a developer executes, such as copying a statement and cutting a method declaration.

The refactoring detection component detects an ongoing refactoring by matching the operations performed by a developer against the transitions in the manual refactoring workflow patterns from our formative study. If a developer's operations match with a prefix of refactoring type R's workflow pattern, she may be performing a refactoring of type R. If her operations continue to match the same pattern, our confidence that she is actually refactoring increases; otherwise the confidence decreases. When the confidence exceeds a predefined threshold T, the refactoring detection component concludes that the developer is manually performing a refactoring of type R and offers a quick-fix to help her finish it.

We next illustrate our refactoring detection algorithm through a pseudo code example shown in Program 1. This example outlines detecting the rename field refactoring by using the refactoring workflow pattern illustrated in the upper transitions (red) in Figure 4. The value of *confidence* indicates the numbers of consecutive matches between a developer's operations and the states of the workflow pattern. The algorithm detects an ongoing rename refactoring when the value of *confidence* is above a predefined *Threshold* (line 22). We currently use a threshold value of 1, resulting in the highest sensitivity of the detection algorithm.

If the refactoring detection component detects an ongoing refactoring, our Eclipse plugin adds a quick-fix button at the line of code where the developer made the latest change. The quick-fix button offers a user interface affordance to allow BeneFactor to finish the refactoring.

B. Code Modification

If the developer invokes BeneFactor to finish her manual refactoring, BeneFactor's code modification component makes the requested change. At a high level, the code modification component captures any information the developer supplied while manually refactoring, recovers the code to its original state by rolling back the manual refactoring, and then re-applies the refactoring automatically. We discuss each of these steps below.

```
1 confidence = 0;
2 current_state = init;
3 while(true){
4  operation = WaitforOperation();
5  if(operation.isUpdate()
6 && operation.Node.isName()
7 && operation.Node.Parent.
8  isFieldDeclaration()){
9   confidence++;
10  current_state = two;
11  declaration = operation.Node;
12 }
13 else if (current_state == two
14 && operation.isUpdate()
15 && operation.Node.isName()
16 && !operation.Node.Parent.
17 isFieldDeclaration()
18 && operation.bindsTo(declaration))
19  confidence++;
20 else if (confidence > 0)
21  confidence --;
22 if(confidence > Threshold)
23  DetectedRefactorings.add(
24  new RenameRefactoring(binding));}
```

Program 1: refactoring detection pseudo code example

1) Configuration Information Collection: The configuration information collection step collects the necessary information to perform an automatic refactoring. Some examples of the configuration information include the visibility modifier of the extracted method and the position of the method in the containing class. The equivalent step in conventional refactoring tools is to ask for this information via a dialog box. But rather than asking for configuration information a second time using a dialog box, BeneFactor automatically collects this information from the code changes that the developer has already made.

While it is convenient to collect configuration information this way, it is not always possible to collect all information necessary to complete a refactoring. For example, if a developer invokes BeneFactor right after she cuts the statements to be extracted, the new method's name is not yet known. In this situation, BeneFactor uses a default value to finish the refactoring first and then allows the developer to modify the default afterwards. However, sometimes even this is not enough. For example, when performing the move static field refactoring, the tool may not know the developer's planned destination class. In those situations, BeneFactor does not offer to finish a refactoring until this critical information is supplied. An alternative approach would be to collect the missing information using a dialog box or wizard.

2) Code Recovery: The code recovery step recovers the code base to before the manual refactoring was applied. Undoing a manual refactoring is not a trivial task, because

```
1 void selectiveUndo(
2 Stack<Operation> operation_stack,
3 Refactoring refactoring){
4  Stack<Change> undo_operation_stack;
5  While(!refactoring.performedEnoughUndos()){
6   Operation operation=operation_stack.pop();
7   operation.undo();
8   undo_operation_stack.push(operation);
9  }
10 while(!undo_operation_stack.isEmpty()){
11  Operation operation=
12  undo_operation_stack.pop();
13  if(refactoring.isRefactoringOperation())
14   operation.skip();
15  else
16   operation.getPatch.applyPatch();}}
```

Program 2: selective undo algorithm pseudo code example

the developer's manual refactoring effort may interleave with other kinds of code changes [17]. For example, when a developer is performing a rename field refactoring, after renaming the field's declaration, she may add new statements to the code. In this situation, arbitrarily undoing all code changes until the beginning of the manual refactoring causes the developer's non-refactoring work to be lost. To tackle this problem, we designed an algorithm called *selective undo*.

We illustrate selective undo by using the pseudo code snippet shown in Program 2. Our selective undo algorithm takes two parameters as input. The first is the stack of operations performed by the developer from the starting of the Eclipse IDE to the moment of the algorithm's execution, with the latest code change at the top. The second input is the refactoring to be completed.

In Program 2, the algorithm first declares a local variable *undo_operation_stack* to store all the operations that have been undone. After the declaration, *performedEnoughUndos* at line 5 checks whether it needs to undo more code changes. This method returns true if the existing automatic refactoring infrastructure can be applied to finish the refactoring, otherwise it returns false. The implementation of *performedEnoughUndos* depends on the type of refactoring. For example, for rename refactoring, *performedEnoughUndos* returns true when the name's declaration has not been renamed; for extract method refactoring, it returns true when the statements to be extracted appear in their original method.

When *performedEnoughUndos* returns false, the selective undo algorithm enters a loop. From line 6 to line 7, the algorithm pops the code change at the top of *operation_stack* and undoes it, regardless of whether or not the operation is a refactoring operation. For every operation that has been undone, the algorithm pushes it back to another stack *undo_operation_stack* at line 8. The selective undo algorithm continuously executes the code from line 6 to line 8 until *performedEnoughUndos* returns true.

After arbitrarily undoing all of these operations, the algorithm redoes only the non-refactoring operations among them. For each undone operation, the algorithm invokes *refactoring.isRefactoringOperation()* at line 13 to check whether or not it belongs to the manual refactoring workflow. *refactoring.isRefactoringOperation()* performs this check by comparing the input operation with the operations in our collected refactoring workflow patterns.

If *refactoring.isRefactoringOperation()* returns true, the algorithm skips the operation because BeneFactor will reapply the refactoring later, as shown at line 14. If *refactoring.isRefactoringOperation()* returns false, the algorithm reapplies this operation. Directly applying this operation is unsafe because the previously skipped operations may change the position where this operation should be applied to. Therefore, we use the patch technique to re-apply this operation. Widely used in source control systems, patch allows one version to a file to be applied to the right location in the modified version of the same file [27]. The algorithm repeats this process from line 11 to 16 until no operation is left in *undo_operation_stack*.

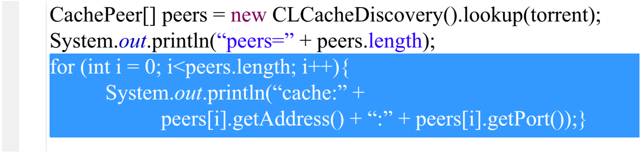
3) Change Creation: After the code base has been recovered, we perform the last step: automatically finishing the refactoring. Eclipse ships with a set of automatic refactoring application programming interfaces (APIs) called the language toolkit (LTK) [9]. Feeding the collected configuration information and the recovered code base as input, LTK APIs are now able to automatically perform the intended refactoring.

C. Example

We next illustrate how BeneFactor works by using a real-world code example. Suppose Grace is using Eclipse with BeneFactor installed. To enhance maintainability, she intends to extract the *for* loop in the code snippet of Figure 6 into a new method. Grace first cuts the statement of the *for* loop. BeneFactor captures this operation and detects that the developer is possibly refactoring, and therefore adds a marker at the line of code where Grace cuts the statement, proposing to help her finish this refactoring, as illustrated in Figure 7. Grace next intends to print a blank line before the printed information in the *for* loop, so she inserts *System.out.println()*, as illustrated by Figure 8.

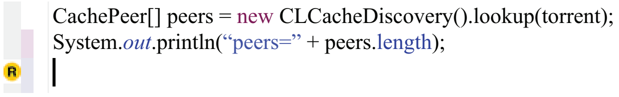
Grace continues her extract method refactoring by adding a new method declaration. She specifies the visibility modifier, method name, and return type of the new method declaration, as shown in Figure 9. Although interrupted by a non-refactoring operation, Grace has performed a sequence of operations following the refactoring workflow pattern modeled by the middle transitions (blue) in Figure 5. Therefore, BeneFactor still detects that Grace is performing an extract method refactoring. Reminded by the BeneFactor icon, Grace realizes she can finish her manual refactoring by invoking the tool. Hence, she chooses the quick-fix item in Figure 9 to extract the method. Finally, BeneFactor finishes her refactoring, resulting in the code snippet shown in Figure

10. The parameter of the extracted method is inferred by the refactoring tool, and thus does not need to be explicitly specified by the developer.



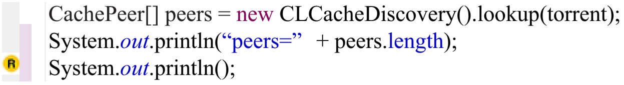
```
CachePeer[] peers = new CLCacheDiscovery().lookup(torrent);
System.out.println(“peers=” + peers.length);
for (int i = 0; i<peers.length; i++){
        System.out.println(“cache:” +
                peers[i].getAddress() + “:” + peers[i].getPort());}
```

Figure 6: Code before extracting method.



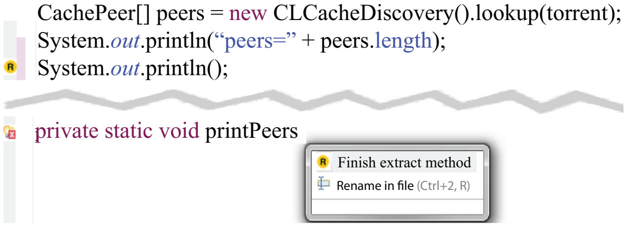
```
CachePeer[] peers = new CLCacheDiscovery().lookup(torrent);
System.out.println(“peers=” + peers.length);
|
```

Figure 7: Code after cutting the statements to extract.



```
CachePeer[] peers = new CLCacheDiscovery().lookup(torrent);
System.out.println(“peers=”  + peers.length);
System.out.println();
```

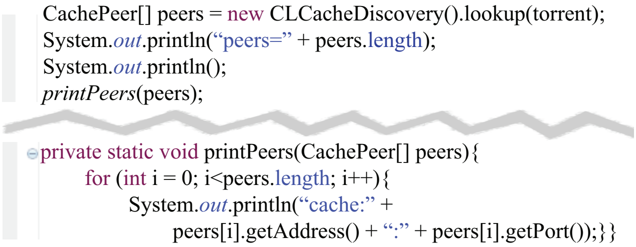
Figure 8: Code after making a non-refactoring change.



```
CachePeer[] peers = new CLCacheDiscovery().lookup(torrent);
System.out.println(“peers=” + peers.length);
System.out.println();

private static void printPeers
```

Figure 9: Code after adding a new method declaration.



```
CachePeer[] peers = new CLCacheDiscovery().lookup(torrent);
System.out.println(“peers=” + peers.length);
System.out.println();
printPeers(peers);

private static void printPeers(CachePeer[] peers){
        for (int i = 0; i<peers.length; i++){
                System.out.println(“cache:” +
                        peers[i].getAddress() + “:” + peers[i].getPort());}}
```

Figure 10: Code after invoking BeneFactor.

D. *Technique Challenges*

While implementing BeneFactor, we faced several challenges. We believe that these challenges must be resolved before BeneFactor can be evaluated.

False Positives: False positives in refactoring detection refer to the situations when BeneFactor falsely assumes that the developer is refactoring manually, but she is actually not. Too many false positives may annoy the developer with an overwhelming number of quick-fix items. The false positives may occur when (1) the developer is performing code changes that serve partially for restructuring purposes, but also are intended to modify the code behavior and (2) the default confidence value is too low, resulting in an overly sensitive refactoring detection algorithm.

False Negatives: False negatives in refactoring detection refer to the situations when BeneFactor fails to detect that a developer is manually refactoring. In this situation, the developer is not able to invoke BeneFactor to finish her ongoing refactoring. The false negatives in refactoring detection may be caused by: (1) A manual refactoring workflow that greatly deviates from our collected manual refactoring workflow patterns; (2) A workflow that is dominated by non-refactoring code changes; (3) A default confidence value that is too high, resulting in an overly insensitive refactoring detection algorithm.

Unresolvable Non-Refactoring Operations: As we have mentioned, a developer's manual refactoring operations may be interleaved with operations serving some other purpose, which we refer to as non-refactoring operations. These non-refactoring operations are preserved during the code recovery step. However, BeneFactor may have difficulty in resolving these non-refactoring operations when they are dependent on the interleaved refactoring.

One example of the unresolvable non-refactoring operation is illustrated by Grace's insertion of *System.out.println();* in Figure 8. BeneFactor must figure out the order of the inserted statement and the invocation of the extracted method. Currently, BeneFactor assumes the invocation happens after all the inserted statements, resulting in the code snippet shown in Figure 10. The developer's intention may have instead been to put the invocation *before* the inserted statements, but BeneFactor currently has no way of determining the developers' intention.

Removing Quick-Fix Items: BeneFactor adds a quick-fix item after detecting a manual refactoring. However, it is not trivial to decide when to remove the item. If BeneFactor removes the item too early, the developer is not able to invoke automatic refactoring when she later wants to. If BeneFactor keeps the quick-fix item for too long, many quick fixes may be strewn around the developer's IDE, long after code changes have been completed.

Scalability: We plan to enhance BeneFactor to support more types of refactorings, and as a developer uses BeneFactor, the developer's operations may simultaneously match with workflow patterns of several different refactoring types. In this case, the developer will face multiple quick fix options to choose from.

V. RELATED WORK

In this section, we summarize several research areas and tools that are closely related to our work.

Empirical Studies of Refactoring. Many empirical studies have investigated refactoring. For instance, Xing and Stroulia conducted a case study on the structural evolution of Eclipse and concluded that refactoring is frequent [26]. Murphy-Hill and colleagues' study of how refactoring tools are used [17]. In contrast to existing studies, our formative study focused on the process of manual refactoring.

Refactoring Detection. To help developers maintain software, researchers proposed several algorithms for detecting refactorings. For instance, Taneja and colleagues proposed a technique called RefacLib to detect performed refactorings in different versions of library files [22]. Prete and colleagues proposed REF-FINDER that identifies complex refactorings by using template logic rules [18]. Kim and colleagues proposed an algorithm based on heuristics to detect rename-method refactorings between two versions of software [11]. In contrast to these algorithms, our approach detects ongoing, partially-completed refactorings.

State-of-the-Art Refactoring Tools. Researchers have proposed novel refactoring tools and guidelines for building new tools. For instance, Mealy and colleagues [13] have distilled a list of 38 usability guidelines for building refactoring tools. Developed in parallel with BeneFactor, the WitchDoctor tool also can help a developer automatically finish a refactoring that was started manually [29]. The major difference is that WitchDoctor uses only code changes when specifying refactoring patterns; BeneFactor also includes behavioral patterns, such as copying code. In some cases, this allows BeneFactor to identify manual refactorings earlier than WitchDoctor would.

State-of-the-Practice Refactoring Tools. Several commercial refactoring tools bear some resemblance to BeneFactor. For example, when you rename a variable declaration in Eclipse, you can sometimes update all references to that variable using a quick fix. However, the rename is purely syntactic; no safety checks are performed because the refactoring API is not actually used. Both Refactor! Pro [19] and ReSharper [20] can detect the first step of a manual refactoring workflow for some types of refactorings. However, BeneFactor takes the concept further by allowing a developer to finish her refactoring at any point in her refactoring workflow.

Recommender Systems. The field of recommender systems have been used extensively in software engineering; specific to refactoring, such systems are known as smell detectors [16][23][24]. They identify code that is in need of refactoring and recommends that the developer refactor that code. In contrast, BeneFactor detects actual refactorings, rather than potential refactorings; it detects refactoring by human operations, rather than code metrics.

Search-Based Refactoring. Search-based techniques detect refactoring candidates and can refactor them automatically [28]. Unlike BeneFactor, search-based refactoring is helpful when the developer does not know what she wants to refactor, but does not help when the developer has already begun refactoring.

VI. FUTURE WORK

Currently, BeneFactor only supports three refactoring types: Rename, extract method and move field. In the future, we plan to support more refactoring types. While BeneFactor faces several challenges (Section IV-D), we plan to explore solutions to tackle them. We also plan to conduct an empirical study to investigate how BeneFactor works in practice. We also plan to conduct a formative study with a wider variety of programmers as participants, providing us more confidence in our refactoring workflow patterns.

According to our observations, sometimes developers rely on compiler errors to help them refactor. However, compiler errors do not reflect refactoring errors, making this practice unreliable. In the future, we plan to design a set of refactoring-aware warnings that augment compiler errors. These warnings would warn a developer when she violates refactoring preconditions, in a manner similar to compiler errors. This gives developers some of the benefits of refactoring tools without forcing them to explicitly "use refactoring tools."

VII. CONCLUSION

Although refactoring tools are widely available to developers, a great portion of refactorings are still performed by hand. Developers avoid refactoring tools partly because of the late awareness of refactoring, that is, the situations when a developer recognizes she is refactoring after she has already begun. In order to address this issue, we conducted a formative study about developers' manual refactoring process. Our study suggests that manual refactoring can be an error-prone process and that reliance on compilation errors can hinder developers from refactoring safely. The results confirm that the late awareness of refactoring contributes to the refactoring tool underuse problem in the real world.

Furthermore, our formative study distills a set of manual refactoring patterns. Building on these patterns, we present a novel refactoring tool called BeneFactor. Unlike conventional refactoring tools, BeneFactor detects when a developer is manually refactoring, reminds her that automatic refactoring is available, and can finish her refactoring automatically. We believe that by alleviating developers from the burden of recognizing when they are refactoring upfront, developers will be more likely to use refactoring tools and gain the benefits that those tools provide.

ACKNOWLEDGMENTS

We thank Moin Ayazifar and Suprit Patankar for their participation in this project. For their advice and guidance, we thank Brittany Johnson, Donghoon Kim, Da Young Lee, Yoonki Song, and Shundan Xiao. We also thank the participants in our formative study. This research was partially funded by NCSU's Faculty Research and Professional Development Grant.

REFERENCES

[1] A. Abadi, Y. A. Feldman, and M. Shomrat. *Code-Motion for API Migration: Fixing SQL Injection Vulnerabilities in Java*. In Proceeding of the Workshop on Refactoring Tools, pages 1-7, 2001.

[2] Apache Tomcat Open Source Project, 2011. `http://tomcat.apache.org/`.

[3] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko. *Let's Go to the Whiteboard: How and Why Software Developers Use Drawings*. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 557-566, 2007.

[4] K. Damevski and M. Muralimanohar. *A Refactoring Tool to Extract GPU Kernels*. In Proceeding of the Workshop on Refactoring Tools, pages 29-32, 2011.

[5] B. Daniel, D. Dig, K. Garcia, and D. Marinov. *Automated Testing of Eclipse and NetBeans Refactoring Tools*. In Proceedings of the Workshop on Refactoring Tools in conjunction with the European Conference on Object-Oriented Programming, pages 42-43, 2007.

[6] B. Daniel, D. Dig, K. Garcia, and D. Marinov. *Automated Testing of Refactoring Engines*. In Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pages 185-194, 2007.

[7] J. R. Foster. *Cost Factors in Software Maintenance*. PhD thesis, University of Durham, 1993.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[9] L. Frenzel. *The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs*, 2006.

[10] Join.me Remote Screen Share, 2011. `https://join.me/`.

[11] S. Kim, K. Pan, and E. J. Whitehead, Jr. *When Functions Change Their Names: Automatic Detection of Origin Relationships*. In Proceedings of the Working Conference on Reverse Engineering, pages 143-152, 2005.

[12] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. *A Case Study in Refactoring a Legacy Component for Reuse in a Product Line*. In Proceedings of the International Conference on Software Maintenance, pages 369-378, 2005.

[13] E. Mealy, D. Carrington, P. Strooper, and P. Wyeth. *Improving Usability of Software Refactoring Tools*. In Proceedings of the Australian Software Engineering Conference, pages 307-318, 2008.

[14] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi. *Does Refactoring Improve Reusability?* In Proceedings of the International Conference on Software Reuse, pages 287-297, 2006.

[15] E. Murphy-Hill and A. P. Black. *Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method*. In Proceedings of the International Conference on Software Engineering, pages 421-430, 2008.

[16] E. Murphy-Hill and A. P. Black. *An Interactive Ambient Visualization for Code Smells*. In Proceedings of the International Symposium on Software Visualization, pages 5-14, 2010.

[17] E. Murphy-Hill, C. Parnin, and A. P. Black. *How We Refactor, and How We Know It*. IEEE Transactions on Software Engineering, 2011.

[18] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. *Template-based Reconstruction of Complex Refactorings*. In Proceedings of the International Conference on Software Maintenance, pages 1-10, 2010.

[19] Refactor! Pro Add-in for Visual Studio, 2011. `http://devexpress.com`.

[20] ReSharper Visual Studio Extension, 2011. `http://www.jetbrains.com/resharper/`.

[21] Skype Voice and Video Call, 2011. `http://www.skype.com`.

[22] K. Taneja, D. Dig, and T. Xie. *Automated Detection of API Refactorings in Libraries*. In Proceedings of the International Conference on Automated Software Engineering, pages 377-380, 2007.

[23] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. *JDeodorant: Identification and Removal of Type-Checking Bad Smells*. In European Conference on Software Maintenance and Reengineering, pages 329-331, 2008.

[24] E. Van Emden and L. Moonen. *Java Quality Assurance by Detecting Code Smells*. In Proceedings of the Working Conference on Reverse Engineering, pages 97-106, 2002.

[25] Vuze Open Source Project, 2011. `http://www.vuze.com/`.

[26] Z. Xing and E. Stroulia. *Refactoring Practice: How it is and How it Should be Supported-An Eclipse Case Study*. In Proceedings of the International Conference on Software Maintenance, pages 458-468, 2006.

[27] Diff, Match and Patch libraries for Plain Text, 2011, `http://code.google.com/p/google-diff-match-patch/`

[28] Mark O'Keeffe and Mel Ó Cinnéide, *Search-Based Refactoring: an empirical study*. In Journal of Software Maintenance and Evolution: Research and Practice, 20: 345-364, 2008.

[29] S. R. Foster, W. G. Griswold, and S. Lerner, *WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings*. In Proceedings of the International Conference on Software Engineering, 2012. To Appear.