Manual Refactoring Changes with Automated Refactoring Validation

Xi Ge Emerson Murphy-Hill
Department of Computer Science
NC State University, Raleigh, NC, USA
xge@ncsu.edu, emerson@csc.ncsu.edu

ABSTRACT

Refactoring, the practice of applying behavior-preserving changes to existing code, can enhance the quality of software systems. Refactoring tools can automatically perform and check the correctness of refactorings. However, even when developers have these tools, they still perform about 90% of refactorings manually, which is error-prone. To address this problem, we propose a technique called GhostFactor separating transformation and correctness checking: we allow the developer to transform code manually, but check the correctness of her transformation automatically. We implemented our technique as a Visual Studio plugin, then evaluated it with a human study of eight software developers; GhostFactor improved the correctness of manual refactorings by 67%.

Categories and Subject Descriptors

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

General Terms

Design, Experimentation, Languages, Human Factors

Keywords

Refactoring, Restructuring, Tool, IDE

1. INTRODUCTION

Refactoring is the process of altering software's internal structure without modifying its external behavior [13]. Studies show refactoring can improve cohesion [20], maintainability [20], evolvability [31], and reusability [24] of existing software systems. Because of these benefits, refactoring is an important part of modern software development. According to Cherubini and colleagues' survey of 427 developers at Microsoft, developers consider refactoring as important as or more important than understanding code and producing documentation [10]. Refactoring is also an integral part of agile development processes such as Extreme Programming [9].

Developers can perform refactorings manually or with automated tools. Manual refactorings are error-prone: according to our previous study of twelve professional developers, one third of their manually performed refactorings inserted defects to the software system [15]. Automated refactoring tools will perform refactorings for developers and automatically check their correctness, enabling developers to quickly and safely refactor their code.

Although automated refactoring tools refactor more correctly than developers do, developers rarely use them. According to existing studies, only 11% of 145 refactorings in real-world open source systems were performed automatically [27, 37].

To solve this underuse problem, researchers have proposed novel tools to encourage developers to refactor automatically. For instance, BeneFactor and WitchDoctor automatically finish refactorings after developers start refactoring manually [15, 12]. These tools significantly reduce, but do not completely remove, the barriers to using refactoring tools. For instance, developers must still explicitly invoke most refactoring tools. In addition, researchers found that developers do not trust automatic refactorings to be correct [25, 11]. Existing tools remove neither of these barriers: developers are unlikely to change their behavior to use a tool they distrust [25].

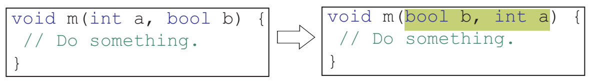
To address these problems, in this paper, we propose a novel static analysis technique. We make the following contributions:

- A technique called GhostFactor that can detect manually performed refactorings and check their correctness. GhostFactor is novel for combining light-weight static analysis with refactoring detection algorithms to quickly detect refactoring errors. Section 4 describes the design of GhostFactor.
- We implemented our technique in an open-source plug-in for the Visual Studio IDE [5]. This plugin, also called GhostFactor, instantly notifies developers when they refactor incorrectly and suggests ways to fix the error. Unlike previous refactoring tools, GhostFactor integrates into the IDE's notification system, a familiar mode of interaction for developers. Section 5 describes the implementation.
- We evaluated GhostFactor by conducting a human study with eight developers. In this study, we compared how participants refactored with or without GhostFactor. GhostFactor improved the correctness of their manual refactorings by 67%. Section 6 presents the design and results of the study.

Before describing our technique, we first provide further motivation for the technique in Section 2 and describe related work in Section 3.

2. MOTIVATION

One desirable property of software development tools, such as refactoring tools, is that they should accommodate the developer,


```
void m(int a, bool b) {
 // Do something.
}
```
```
void m(bool b, int a) {
 // Do something.
}
```
(a) A developer changes a method declaration.

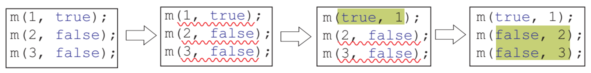

```
m(1, true);
m(2, false);
m(3, false);
```
```
m(1, true);
m(2, false);
m(3, false);
```
```
m(true, 1);
m(2, false);
m(3, false);
```
```
m(true, 1);
m(false, 2);
m(false, 3);
```
(b) A developer relies on compiler errors to update call sites.

Figure 1: Refactoring with compiler warnings.

rather than the developer having to accommodate the tools. In this section, we give two common patterns developers use when refactoring, show how existing refactoring tools do not accommodate this behavior, and describe how our proposed approach accommodates this behavior.

Developers' Refactoring Patterns. Our previous work found two common patterns in developers' refactoring behavior. We found developers transformed their code manually nine out of ten times, rather than using a refactoring tool [27]. Part of the reason for this is that developers do not trust the changes refactoring tools make [25].

The second pattern we found in developers' refactoring behavior was using compiler errors to guide and check the correctness of manual refactorings [27]. For example, suppose a developer is going to change the signature of method `m()` in Figure 1a by exchanging the positions of its first and second parameters. She might first change the method declaration, as illustrated in Figure 1a. The compiler raises errors at all the method's call sites, because the types of their arguments no longer match the declared type signature. The developer can use these compiler errors to find all the call sites, then correct them to finish the refactoring, as illustrated in Figure 1b.

In some cases, such as the example in Figure 1, developers can use this pattern to finish refactorings completely and correctly. However, this is not always the case. For example, changing the parameters of `void n(int a, int b)` to `void n(int b, int a)` will not raise any compiler errors because the exchanged parameters share the same type. Our previous study found developers who rely on such compiler errors make more refactoring errors [15].

Existing Tools Do Not Accommodate These Patterns. Existing refactoring tools do not accommodate these common refactoring patterns: they require developers to manually invoke them, so developers must change their workflow to use them. However, we found some developers do not know what refactoring tools are available. Even when they do, developers sometimes complete part of a refactoring before realizing they are refactoring. They often choose to simply complete the refactoring manually, rather than undoing previous changes and invoking a tool [15]. Thus, requiring explicit invocation is a barrier to the use of traditional refactoring tools.

The refactoring tools BeneFactor [15] and WitchDoctor [12] address these awareness problems, but do not accommodate developers' existing refactoring patterns. Specifically, both tools require the developer to trust the transformations the tools make. As mentioned previously, many developers do not trust tools to make correct changes [25].

Our New Approach Accommodates These Patterns. GhostFactor fits into more developers' workflows because it does not assume that developers know about refactoring tools or trust automatic code transformations. Our approach separates automatic transformations from automatic correctness checks so developers can benefit from the second without adopting the first. GhostFactor uses familiar notifications, similar to compiler warnings, to inform the developer of errors made while manually refactoring.

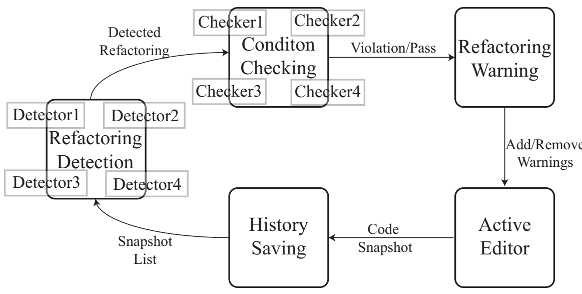


Figure 2: GhostFactor components.

3. RELATED WORK

Before delving into the detail of GhostFactor, we first review the existing research related to our work. There is a huge body of existing works related to ours. We briefly summarize them in the following categories: refactoring studies, improvement of refactoring tools, and refactoring detection.

Refactoring Studies. Researchers have conducted multiple studies related to refactorings. Kim and colleagues' study suggested that developers avoid refactoring tools in spite of the fact that they know their intended refactorings can be automated [19]. Murphy-Hill and colleagues showed how refactoring tools are used and that they are significantly underused [27]. Based on this study, Vakilian and colleagues further investigated how refactoring tools are misused in various situations [35]. The study conducted by Tokuda and Batory suggested that software evolution could be significantly accelerated by applying a set of general-purpose refactorings [34]. In contrast to these studies, our study in this paper investigates whether developers can perform manual refactorings more correctly with assistance of enhanced compiler-like warnings.

Improvement of Refactoring Tools. Researchers have proposed various ways to improve automated refactoring. For instance, Mens and colleagues formalize refactoring by using graph transformations [23]. Bavota and colleagues automatically identify method chains and refactor these method chains to cohesive classes [6]. Extending refactoring to the aspect-oriented programming, Hannemann and colleagues introduced an interactive way of transforming code base[17]. To improve the quality of refactoring tools, Daniel and colleagues proposed an automatic testing framework [11]. Lee and colleagues proposed a drag-and-drop refactoring tool that liberates developers from remembering different refactoring types [21]. The aim of these approaches is to provide more powerful refactoring capabilities or user interfaces; the aim of our approach is to align refactoring tools with the way developers manually refactor.

BeneFactor [15] and WitchDoctor [12] can both detect manual refactorings and finish them automatically, relieving the programmer from having to recognize that the task she is about to perform is a refactoring. To further reduce the barriers of automatic refactorings, GhostFactor can check the correctness of manually performed refactorings without being invoked explicitly. Although GhostFactor and BeneFactor share similar user interfaces, they are significantly different from each other. Firstly, BeneFactor looks for incomplete manual refactorings; GhostFactor, to the contrary, detects manually finished refactorings. Secondly, BeneFactor ap-

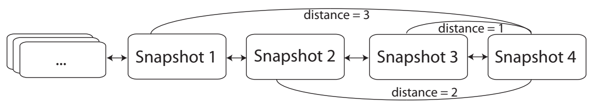


Figure 3: Snapshots compared in a snapshot list.

plies quick fix changes to finish refactorings, whereas GhostFactor applies quick fixes to remedy existing refactoring errors.

Drag-and-drop refactoring tool infers developers' refactoring intention by the source and the destination of their drag-and-drop actions and finishes the intention automatically [21]. Different from the tool, GhostFactor seeks to correct manual refactorings, instead of eliminating the obstacle to invoking automatic refactorings.

Refactoring Detection. Researchers have proposed many refactoring detection techniques for various purposes. Prete and colleagues proposed REF-FINDER to identify complex refactorings by using template logic rules [30]. Bavota and colleagues proposed refactoring detection techniques by using semantic measurement [7] and game theory [8]. BeneFactor [15] and WitchDoctor [12] detect ongoing manual refactorings in order to finish them automatically. Although in this paper we implemented algorithms for detecting refactorings from scratch, reusing these existing techniques is an alternative way to implement the refactoring detection component of GhostFactor.

Refactoring Conditions. Conventional refactoring tools adopt a heuristic-based technique to ensure the correctness of automatic refactorings, namely refactoring condition checking [4]. As alternatives, Schäfer and colleagues proposed the concepts of dependency notions and microrefactorings to implement automatic refactorings more understandably and correctly [32, 33]; Overbey and colleagues proposed a language-agnostic technique called differential precondition checking [29]. In contrast, this paper applies the conventional technique of refactoring condition checking to detect defects introduced by manual refactorings.

4. APPROACH

GhostFactor detects developer-introduced defects to ensure the correctness of manual refactorings. In this section, we describe its design. GhostFactor has several independent components, each of which handles a different task. Figure 2 illustrates these components and how they interact with each other.

The first component is the **history saving** component. This component records the change history of different files that a developer has worked on. It registers a listener to content change events issued from the active editor. When an event occurs, the component takes a snapshot of the source file currently opened in the active editor and saves its content in memory. The change history of a specific source file is maintained as a snapshot list to facilitate sequential access, where the head of the list is the most recent snapshot. After saving the source file's content and updating its snapshot list, the component feeds the list to the next component.

The **refactoring detection** component detects refactorings that the developer completes manually. This component takes the snapshot list for a source file as input, then dynamically loads available refactoring detectors and applies them to the snapshot list. This component uses each detector to compare the latest snapshot in the snapshot list with one of the previous snapshots. For instance, Figure 3 depicts a snapshot list where the latest node is *Snapshot 4*. We show the compared snapshots by drawing arcs between them. In this figure, we also label the distance between compared snapshots, where the distance is one more than the number of snapshots between the two. GhostFactor does not compare snapshots if the distance between them is larger than some constant maximum; we have set this maximum 30 in our implementation. This constraint helps ensure that the detection finishes before another snapshot arrives.

All the refactoring detectors share a common interface that developers can use to implement their own detectors. The interface takes two abstract syntax trees (ASTs) of the same source file as inputs and generates outputs that indicate whether a refactoring is detected, the detected refactoring's type, and the AST nodes affected by the refactoring. For example, the affected AST nodes of an extract method refactoring are the newly declared method and the invocations of that method.

If GhostFactor detects a manual refactoring, it feeds the refactoring to the **condition checking component**. This component dynamically loads available condition checkers for the given refactoring's type. These condition checkers can check for both preconditions and post-conditions for the given refactoring [28]. All of the condition checkers for a refactoring type share a common interface with which developers can implement their own condition checkers to add to GhostFactor. The inputs of this interface include the detected refactoring and the ASTs in which the refactoring is detected. The output indicates whether the refactoring violates or passes the condition.

The last component,**refactoring warning component**, keeps track of detected condition violations. Taking a condition violation as input, the component first checks whether a warning has already been issued for this violation. If one has, this component discards the redundant violation; otherwise, the component saves it and issues a new refactoring warning. Taking a passing condition as input, the component will check whether this passing condition resolves an existing refactoring warning. If it does, the component dismisses this warning. For each refactoring warning, GhostFactor also provides quick fixes with which the violation can be automatically resolved.

To illustrate how the last component works, we next give an example. Suppose a developer is refactoring on a source file F, consecutively generating four different snapshots F_1, F_2, F_3, and F_4. F_1 is the original file; F_2 contains an erroneous refactoring R that violates a condition C; F_3 also contains R violating the condition C; F_4 contains the corrected R that fixed its violation of C. In this example, the refactoring warning component will consecutively receive three different condition checking results, which are R's violation of C, R's violation of C again and finally R's pass of C. According to the mechanism of the refactoring warning component, GhostFactor will issue a refactoring warning at F_2 and F_3, while removing the warning at F_4.

5. IMPLEMENTATION

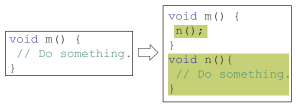
Built on the Microsoft Roslyn project [3], we implemented GhostFactor as a plug-in for the Visual Studio IDE to help refactor C# software; the source code of this plug-in is open to the public[1]. In this section, we describe this implementation.

5.1 Supported refactoring types

The GhostFactor plug-in currently supports the following three refactoring types.

- **Extract method**: According to Fowler's refactoring catalog, extract method is a type of refactoring that "turns a set of statements into a new method whose name clearly explains that purpose of the method" [13], as illustrated in Figure 4a.

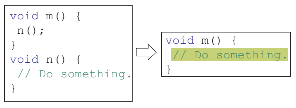
[1] `https://github.com/nkcsgexi/GhostFactor2`



```
void m() {
 // Do something.
}
```

```
void m() {
 n();
}
void n(){
 // Do something.
}
```

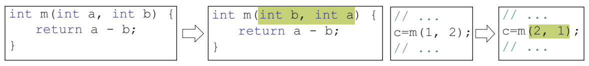
(a) Extract method.



```
void m() {
 n();
}
void n() {
 // Do something.
}
```

```
void m() {
 // Do something.
}
```

(b) Inline Method.



```
int m(int a, int b) {
    return a - b;
}
```

```
int m(int b, int a) {
    return a - b;
}
```

```
// ...
c=m(1, 2);
// ...
```

```
// ...
c=m(2, 1);
// ...
```

(c) Change method signature.

Figure 4: Refactoring examples.

Fowler recommends Extract method to eliminate code smells such as long method and duplicated code [18].

- **Change method signature**: Change method signature is a refactoring type that changes the parameter list of a given method without modifying the method's functionality, as illustrated in Figure 4c. According to Fowler's list, change method signature can be further divided into subcategories such as add parameter, remove parameter, and introduce parameter object [13]. Change method signature is mainly for eliminating code smells such as long method signature, data clumps, and long method [18].

- **Inline method**: The opposite of extract method, inline method is a refactoring type that "puts a method body into its caller and remove the method" [13], as illustrated in Figure 4b. The refactoring is intended to eliminate methods that are short and unnecessary [18].

We selected these refactoring types for two reasons: (1) according to Murphy-Hill and colleagues' previous study of refactoring tool usage, these refactoring types are among the most frequently performed ones [27]; and (2) in our previous study about manual refactorings, developers introduced the most defects when manually performing extract method and change method signature [15]. For inline method, although we do not have evidence, we speculate that it is as error-prone as extract method because of both refactorings' similar complexity.

5.2 Refactoring Detectors

To check the correctness of refactorings, GhostFactor needs to detect them first. Taking two versions of a source file as an input, each detector outputs whether a refactoring is performed and describes the detected refactoring for later analysis. In this subsection, we present our currently implemented detectors.

Extract Method. Our detecting algorithm for extract method is based on the observation that after a refactoring finishes, a new method declaration will contain part of a previously existing method. By comparing the new method against the removed part of the previously existing method, this detector determines whether an extract method refactoring took place. The details of this detection algorithm could be found in Appendix [16].

Change Method Signature. For change method signature, GhostFactor warns developers about call sites that fail to be updated after a method signature changes. In most cases, conventional compiler warnings are enough to achieve this, as the example in Section 2 illustrated. However, when the changed signature has exactly the same parameter count and types with its previous version, conventional compiler warnings are not able to help, as illustrated in the manual refactoring study [15]. Our algorithm detects these situations; the detail of the algorithm can be found in Appendix [16].

Inline Method. Our inline method detection algorithm is based on the observation that simply replacing the invocations of n with n's method body in m's body before the refactoring will result in a method that is similar to m after the refactoring. Details of this algorithm can be found in Appendix [16].

5.3 Condition Checkers

After detecting a manual refactoring, we next check whether the refactoring preserves the software's external behavior. Multiple ways exist to check correctness, such as formal verification, testing, and checking refactoring conditions [28]. Formal verification proves the behavior-preserving of software before and after refactorings, however this technique is prohibitively heavy, potentially jeopardizing the goal of providing developers instant feedback [14]. Testing, or more specifically regressing testing, can expose software's behavior changes, though a recent study from Kim and colleagues shows that the test cases associated with a project are often insufficient to expose refactoring errors [19]. As another option, automatically generating tests instead of using the existing ones likely leads us to the other problems like the object-creating issue [36].

Therefore, we chose to use similar condition checking to existing refactoring tools [4]. Checking refactoring conditions, although ad hoc in nature, has two advantages that other ways do not: (a) checking refactoring conditions is more time efficient, aligning with the requirement of instant feedback; and (b) condition checking allows us to issue informative refactoring warnings convenient for developers to resolve, rather than only telling that the external behavior of the software has been changed.

Similar to the refactoring tools integrated into the mainstream IDEs, condition checkers in GhostFactor check whether a given manual refactoring violates a predefined set of conditions. However, not every condition requires an implementation in GhostFactor, because some violations can trigger conventional compiler errors. Before implementing GhostFactor, we categorized the refactoring conditions into three categories: (1) those whose violations trigger conventional compiler errors; (2) those whose violations do not; (3) those whose violations sometimes do and sometimes do not. Interested readers can find the categorization in our project website[2]. In summary, we manually surveyed the 10 most frequently used refactoring types, and none of them have all its conditions in category (1).

GhostFactor dynamically loads condition checkers at runtime to allow developers to easily add new checkers by implementing our predefined interfaces. We selected three refactoring conditions to implement. The criteria for selecting these conditions were: (1) the condition should be associated with a refactoring type that our implementation can detect; (2) violation of the condition does not trigger conventional compiler errors; and (3) violation of the condition should happen frequently according to our previous study [15]. In the rest of this section, we present the implementation of the checkers for these three conditions.

5.3.1 Return Value Checker

When performing extract method, statements to extract may modify the values of some local variables that are accessed after the extracted statements. The modified values of these variables need to be returned from the extracted method; otherwise the code after the extracted statements in the original method will get incorrect values for these variables. Take the code in Figure 5 as an example, if we intended to extract statements from line 4 to line 8, the

[2]`https://sites.google.com/site/ghostfactorstudy/`

```
1  private double InternalFunction(double[] X) {
2    // ...
3    for (int i = 0; i < this._SimplexVariableList.Length; i++) {
4      if (this._SimplexVariableList[i].Fixed == false) {
5        this._ExternalVariables[i] = X[varFreeVarIndex] * this.
6          _SimplexVariableList[i].ScaleFactor;
7        varFreeVarIndex++;
8      }
9    }
10   // ...
11 }
```

Figure 5: Code before extract method refactoring.

```
private double InternalFunction(double[] X) {
  // ...
  for (int i = 0; i < this._SimplexVariableList.Length; i++) {
    LoopBody(X, varFreeIndex);
  }
  // ...
}
private void LoopBody(double[] X, int varFreeIndex) {
  if (this._SimplexVariableList[i].Fixed == false) {
    this._ExternalVariables[i] = X[varFreeVarIndex] * this.
      _SimplexVariableList[i].ScaleFactor;
    varFreeVarIndex++;
  }
}
```

Figure 6: Missing return value error.

updated value of `varFreeVarIndex` needs to be returned and assigned back to the local variable `varFreeVarIndex`. Without returning this value, as illustrated in Figure 6, although incorrect as a refactoring, the change will not result in compiler errors. To help developers recognize such errors, we implement a checker for missing return values.

The idea of the return value checker is straightforward: (1) it first applies data flow analysis to find local variables accessed by the code after the extracted statements in the original method; (2) it applies data flow analysis to find local variables written in the extracted statements; and (3) local variables that are found in both steps (1) and (2) need to be returned from the extracted method. Finally, we check whether the manually extracted method has all these variables returned and assigned back. If not, GhostFactor will issue a refactoring warning for a missing return value at the extracted method's declaration.

GhostFactor provides a quick fix option with the warning. After the developer invokes the quick fix, a return statement will be added automatically and the returned value will also be assigned back to the modified local variable, as illustrated in Figure 7. If multiple return values are needed, GhostFactor makes the parameters pass-by-reference.

5.3.2 *Parameter Checker*

When performing the extract method refactoring, the newly extracted method needs to have correct parameters. Taking the program in Figure 5 as an example, to extract code from line 4 to line 8, the extracted method needs to take `X` as a parameter. Failing to do so changes the code's external behavior. In most cases, if the extracted method accesses variables that are not passed, the undefined symbols will trigger compiler errors. However, compiler errors are not always reliable; consider the case when the extracted method needs a parameter whose name is identical to a field of the containing class. Code in Figure 8 illustrates this situation.

To ensure the extracted method has the correct parameters, the parameter checker gets the needed parameters by applying data flow analysis to the extracted statements. Next, it checks if the extracted method has all of these needed variables as parameters. If not, GhostFactor issues a refactoring warning for the missing parameters.

Similar to the return value warnings, a warning for missing parameters comes with a quick fix option. After the developer invokes

```
private double InternalFunction(double[] X) {
  // ...
  for (int i = 0; i < this._SimplexVariableList.Length; i++) {
    varFreeVarIndex = LoopBody(X, varFreeIndex);
  }
  // ...
}
private int LoopBody(double[] X, int varFreeIndex) {
  if (this._SimplexVariableList[i].Fixed == false) {
    this._ExternalVariables[i] = X[varFreeVarIndex] * this.
      _SimplexVariableList[i].ScaleFactor;
    varFreeVarIndex++;
  }
  return varFreeVarIndex;
}
```

Figure 7: Extract method errors fixed.

```
private double[] X; // A field named X exists.
private double InternalFunction(double[] X) {
  // ...
  for (int i = 0; i < this._SimplexVariableList.Length; i++) {
    varFreeIndex = LoopBody(varFreeIndex);
  }
  // ...
}
private int LoopBody(int varFreeIndex) {
  if (this._SimplexVariableList[i].Fixed == false) {
    this._ExternalVariables[i] = X[varFreeVarIndex] * this.
      _SimplexVariableList[i].ScaleFactor;
    varFreeVarIndex++;
  }
  return varFreeIndex;
}
```

Figure 8: Missing parameter error.

the quick fix, the correct parameters will be added to the declaration of the extracted method and the correct arguments will be added to its call sites. Figure 7 illustrates the code after fixing the error.

5.3.3 *Stale Invocation Checker*

The change method signature detector described in Section 5.2 detects signature changes of a method declaration where a compiler will not issue errors. The stale invocation checker for the change method signature refactoring simply finds invocations of the changed method, and issues warnings to those that have not been updated. The warning also has a quick fix option that can automatically update all of these invocations.

5.3.4 *Modified Variable Checker*

This checker aims at identifying two kinds of errors when performing inline method refactorings, which are *incorrectly updated variables* and *incorrectly non-updated variables*.

When performing the inline method refactoring, after the developer replaces the invocation(s) of the inlined method with its method body, any variable updates that were inside the method body may now change variables' values in its caller, possibly causing the modification to the caller's behavior. We illustrate this *incorrectly updated variable* error in Figure 9. Notice that in the method body of `GoToDefinitionCPlusPlus`, the parameter `target` is updated at line 15. Suppose a developer tries to inline the method `GoToDefinitionCPlusPlus`, after she replaces its invocation in method `GoToDefinition` with its method body, the logged value of `target` at line 6 may differ from its previous version (when `target == null` and `span.IsSome() == true`), introducing a subtle bug that compiler warnings will not alert the developer about. Line 8 in Figure 10 illustrates this refactoring error.

Another error in inline method refactorings is *incorrectly non-updated variables*. Also taking the code in Figure 9 as an example, the return value of `GoToDefinitionCPlusPlus` is assigned to a local variable `successful` at line 3. To guarantee the semantic equivalence, after replacing the invocation with its method body, the developer needs to assign the value of

```
public void GoToDefinition() {
  // ...
  successful = GoToDefinitionCPlusPlus(text, target);
  // ...
  if(successful)
    logger.Info(target);
}
private bool GoToDefinitionCPlusPlus(ITextView textView,
  string target) {
    if (target == null) {
    // ...
    var span = wordUtil.GetFullWordSpan(WordKind.NormalWord,
      caretPoint);
    // Update parameter.
    target = span.IsSome() ? span.Value.GetText() : null;
  }
  // ...
  return SafeExecuteCommand(CommandNameGoToDefinition);
}
```

Figure 9: Code before inline method refactoring.

```
1  public void GoToDefinition() {
2    // ...
3    if (target == null) {
4      // ...
5      var span = wordUtil.GetFullWordSpan(WordKind.NormalWord,
6        caretPoint);
7      // Error 1.
8      target = span.IsSome() ? span.Value.GetText() : null;
9    }
10   SafeExecuteCommand(CommandNameGoToDefinition); // Error 2.
11   // ...
12   if (successful)
13     logger.Info(target);
14 }
```

Figure 10: Inline method errors.

`SafeExecuteCommand` to `successful`. Failing to do so may also cause unintentional changes to the logged message. Line 10 in Figure 10 illustrates this inline method error.

To help developers identify these inline method errors, the modified variable checker again performs data flow analysis, the detail of which can be found in Appendix [16]. After identifying any of these errors, GhostFactor presents a refactoring warning to the developer with the problem declaration, and also provides a quick fix option to help him automatically resolve this issue. After the developer clicks the quick fix option, GhostFactor performs the following changes:

- For any *incorrectly updated variable* v, GhostFactor introduces a new local variable v' to store the value of v before the inlined statements, and assigns the value of v' back to v after the inlined statements. Line 3 and Line 12 in Figure 11 illustrate how GhostFactor resolves the problem in Figure 10.

- For any *incorrectly non-updated variable*, GhostFactor first finds out return statements in the inlined method body before refactoring; next, GhostFactor finds the local variables that are supposed to save these returned values after refactoring; finally, GhostFactor inserts statements that assign the returned values to these local variables at proper positions. Line 11 in Figure 11 illustrates how GhostFactor resolves this problem in Figure 10.

5.4 Limitations

In this subsection, we summarize the known limitations of GhostFactor.

Snapshots. GhostFactor's detection of manual refactorings relies on the snapshots taken from the developer's code changes, which in turn rely on IDEs' notifications of such event [4, 5]. Different IDEs notify plugins about source code changes at different frequency. If the frequency is too low, some significant snapshots may be missing, leading to manual refactorings remaining undetected. To deal with this limitation, we plan to investigate mechanisms that optimize the likelihood of manual refactorings being

```
1  public void GoToDefinition() {
2    // ...
3    var originalTarget = target; // Fixing error 1.
4    if (target == null) {
5      // ...
6      var span = wordUtil.GetFullWordSpan(WordKind.NormalWord,
7        caretPoint);
8      target = span.IsSome() ? span.Value.GetText() : null;
9    }
10   // Fixing error 2.
11   successful = (SafeExecuteCommand(CommandNameGoToDefinition));
12   target = originalTarget; // Fixing error 1.
13   // ...
14   if (successful)
15     logger.Info(target);
16 }
```

Figure 11: Inline method errors fixed.

detected without losing significant computing resources. Another limitation is that currently GhostFactor only compares the snapshots of a single file to detect refactoring. For some refactoring types, such as move method, the refactorings can only be detected by synergically comparing snapshots across file boundaries. We plan to add this feature in future work.

False Positives. Like most static analysis techniques, false positives happen in GhostFactor. Especially when developers interleave their refactoring changes with non-refactoring changes, a violation detected by GhostFactor may actually be what the developer intended [27]. False positives may also lead to stubborn warnings. As we show in Section 4, the removal of a refactoring warning relies on the successful detection of a corrected refactoring. If GhostFactor successfully detects an erred refactoring while failing to detect its later correction, a refactoring warning may persist longer than appropriate.

To deal with these false positives, we plan to improve GhostFactor in the following two ways: (1) add a user interface affordance that allows developers to dismiss false positive warnings and (2) design more sophisticated algorithms for refactoring detection utilizing data collected in (1).

When developers dismiss false positives in GhostFactor, the refactoring detection algorithms could adapt future violations accordingly. For instance, when several refactoring errors are consecutively accepted by the developer, she is likely performing root-canal refactorings where non-refactoring code changes are less often interleaved [27]. In that situation, GhostFactor can more aggressively assume detected refactorings are actual refactorings. On the other hand, if refactoring errors consistently marked by the developer as false positives, GhostFactor should lower the sensitivity of refactoring detection to maintain the developer's confidence in future error messages.

False Negatives. Developers may perform multiple refactorings together, which may hinder GhostFactor from detecting some of them. One example is when a developer first extracts some statements to a new method and afterwards renames the variables used in these statements. Because GhostFactor detects the extract method refactoring by measuring text similarity, the detection component may not recognize this as a refactoring.

5.5 Example

We next use an example to illustrate how GhostFactor works. Suppose Susan is a C# developer working on the DotNumeratic open source project [2]. She has installed GhostFactor into her IDE. One day, Susan noticed that the method in Figure 12 was undesirably long. Therefore, she decided to extract part of the method into a new method. After she manually extracted the code in the box, Susan transformed the code in Figure 12 to the code in Figure 13. Susan thought she had finished the refactoring correctly. Right before she started coding somewhere else, GhostFactor detected the

```
void ConsoleReadLine(
    Characters destination,
        int length) {
    string s = Console.ReadLine();
    if (s.Length > length)
        s = s.Substring(0, length);
    destination.ToBlanks(length);
    FortranLib.Copy(destination, s);
}
```

Figure 12: Original code.

```
void ConsoleReadLine(
    Characters destination,
        int length) {
    string s = Console.ReadLine();
    TrimString(s, length);
    destination.ToBlanks(length);
    FortranLib.Copy(destination, s);
}
void TrimString(string s,
    int length) {
    if (s.Length > length)
        s = s.Substring(0, length);
}
```

Figure 13: Code after manually extracting method.

refactoring by comparing the code in Figure 13 and Figure 12. Further analysis performed by the return value checker of GhostFactor found that the extracted method failed to return `s`. After that, GhostFactor issued a refactoring warning to the newly extracted method, as illustrated in Figure 14.

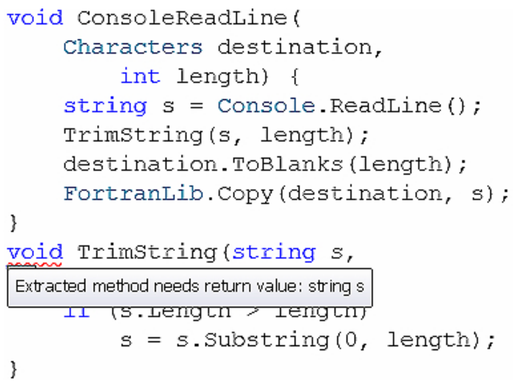
Susan noticed this warning message and was aware of this refactoring error. So she invoked the quick fix options associated with the refactoring warning, as illustrated in Figure 15. GhostFactor automatically added a return statement and assigned `s` back to the proper local variable, resulting in the code shown in Figure 16, where the code automatically changed by GhostFactor is in boxes. Susan refactored correctly this time with the help of GhostFactor, which afforded her the benefit of refactoring tools without requiring her to explicitly use these tools.

6. EVALUATION

To evaluate the effectiveness of GhostFactor in improving manual refactoring's correctness, we conducted a study with participants from both academia and industry. With this study, we intend to answer the following three research questions:

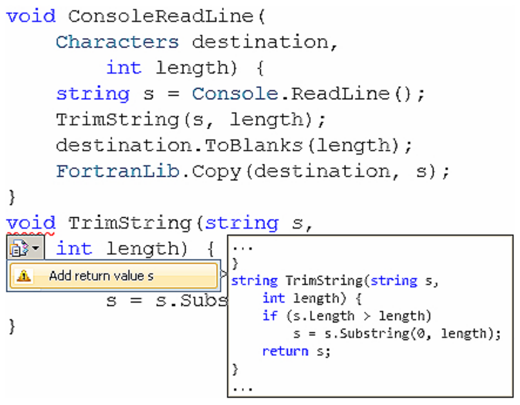
- **Q1.** When manually refactoring, can GhostFactor help developers to refactor more correctly compared to not using it?
- **Q2.** When manually refactoring, can GhostFactor help developers achieve correct refactorings more quickly?
- **Q3.** How can we improve GhostFactor to better assist developers with manual refactorings?

Our study consists of three parts: (1) a pre-study questionnaire that collects participants' demographic data, (2) several refactoring tasks for participants to manually finish, either with or without GhostFactor, and (3) a post-study questionnaire that collects participants' opinions on GhostFactor. In this section, we present the design of the study and summarize the study results.



```
void ConsoleReadLine(
    Characters destination,
        int length) {
    string s = Console.ReadLine();
    TrimString(s, length);
    destination.ToBlanks(length);
    FortranLib.Copy(destination, s);
}
void TrimString(string s,
Extracted method needs return value: string s
    if (s.Length > length)
        s = s.Substring(0, length);
}
```

Figure 14: GhostFactor error message.



```
void ConsoleReadLine(
    Characters destination,
        int length) {
    string s = Console.ReadLine();
    TrimString(s, length);
    destination.ToBlanks(length);
    FortranLib.Copy(destination, s);
}
void TrimString(string s,
    int length) {
Add return value s
        s = s.Subs
}
```

```
...
}
string TrimString(string s,
    int length) {
    if (s.Length > length)
        s = s.Substring(0, length);
    return s;
}
...
```

Figure 15: GhostFactor quick fix.

6.1 Pre-study Questionnaire

In total, we recruited 8 participants, 6 from the computer science department of North Carolina State University and 2 from local IT companies. We required the participants have a college degree in computer science. As compensation, each participant received a 10 dollar gift card after finishing the study. Before we asked them to perform any refactorings, we asked participants to complete a pre-study questionnaire. The questions collected demographic data relevant to our study. These questions include:

1. How many years have you been a programmer?
2. How familiar are you with Java programming language? Please rate yourself from 1 to 5, 1 for not at all and 5 for expert.
3. How familiar are you with C# programming language? Please rate yourself from 1 to 5, 1 for not at all and 5 for expert.
4. How familiar are you with refactoring practices? Please rate yourself from 1 to 5, 1 for not at all and 5 for "I refactor every time I program".
5. What is the percentage of your programming time involving with refactoring? Please specify the percentage.
6. What is the percentage of your refactorings finished by applying refactoring tools? Please specify the percentage.

Table 1 presents the participants' responses, where the number of opaque stars indicates participants' ratings on scales from 1 to

```
void ConsoleReadLine(
    Characters destination,
        int length) {
    string s = Console.ReadLine();
    s = TrimString(s, length);
    destination.ToBlanks(length);
    FortranLib.Copy(destination, s);
}
string TrimString(string s,
    int length) {
    if (s.Length > length)
        s = s.Substring(0, length);
    return s;
}
```

Figure 16: Code after GhostFactor fixes.

5. The participants had a median of 5.5 years for programming experience, a median Java proficiency of ★★★★☆, a median C# proficiency of ★★☆☆☆, and a median refactoring proficiency of ★★★★☆. All participants considered refactoring an integral part of their programming. Two participants, one in the treatment group and one in the control group, indicated that they do not use refactoring tools at all.

6.2 Refactoring Tasks

After participants finished the pre-study questionnaire, we asked them to manually finish a set of refactoring tasks in Visual Studio 2010. We did not mention that the purpose of the study is to evaluate our tool; we only informed participants that we were interested in how correct manual refactorings are. Afterwards, we randomly assigned participants to the treatment group (T in Table 1) or the control group (C in Table 1). The treatment group refactored with GhostFactor's assistance, while the control group refactored without it. The only difference between these two groups' development environments was whether GhostFactor was running. Both groups of participants were allowed to use conventional compiler warnings. We also disabled GhostFactor's quick fix options to avoid the bias introduced by automatic code completion, which could enable participants in the treatment group achieve correct refactorings faster.

We selected real code examples from an open source project called DotNumerics [2]. Written in C#, DotNumerics implements multiple algorithms for linear algebra, differential equations and optimization problems. We selected this project for two reasons: (1) DotNumerics has a non-trivial code base of over 10K lines of code, in which we easily found real refactoring opportunities; and (2) DotNumerics contains procedures for solving well-known mathematical problems that we expected participants to easily understand.

We selected 6 code examples from DotNumerics where refactoring could be properly performed. We selected these code examples based on five criteria: (1) refactoring these code examples should be particularly error-prone; (2) the refactorings to be performed on these code examples should be of the types detectable by GhostFactor (mentioned in Section 5.2); (3) the errors that participants may make when refactoring these code examples should include those checkable by GhostFactor (mentioned in Section 5.3); (4) the code examples should not be too complex to refactor manually; and (5) the code examples should not manifest any syntactic differences between C# and Java, as all 8 participants had less experience in C# than in Java.

Table 1: Pre-study questionnaire results for both treatment (T) and control (C) group.

	Years as programmer	Java proficiency	C# proficiency	Refactoring proficiency
T	4	★★★★☆	★★☆☆☆	★★★☆☆
	5	★★★★☆	★☆☆☆☆	★★★☆☆
	4	★★★★★	★★★☆☆	★★★☆☆
	4	★★★☆☆	★★★☆☆	★★★★☆
C	9	★★★★★	★★☆☆☆	★★★☆☆
	6	★★★☆☆	★★★☆☆	★★★☆☆
	3	★★★☆☆	★★☆☆☆	★★★★☆
	4	★★★☆☆	★★☆☆☆	★★★☆☆

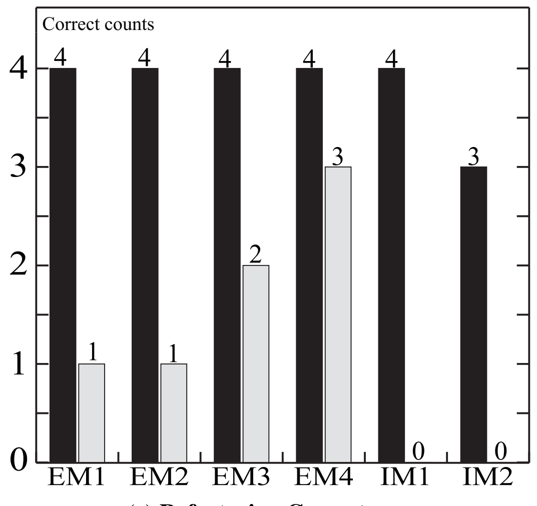
For all of the code examples used in our study, we refer readers to our project website for detail. Of these 6 code examples, we asked participants to perform the extract method refactoring on 4 and the inline method refactoring on 2. We chose more examples for the extract method refactoring because we have implemented twice as many checkers for that refactoring. We did not include any refactoring tasks for the change method signature refactoring. In exploratory iterations of this study, although participants correctly performed change method signature refactorings with GhostFactor's assistance, they refactored exactly as though they were using compiler warnings. This indicated that including these tasks in the study may not produce interesting results, hence we excluded them from the study described here.

The first author led participants through the code examples sequentially and asked the participants to manually complete the refactoring tasks as correctly as possible. Before a participant started to refactor, we reminded her or him of the definition of refactoring correctness mentioned previously in Section 5.2. We recorded the screencast of participants' manual refactorings using CamStudio [1]. We used these recordings to compare the treatment and the control group's refactorings to answer research questions **Q1** and **Q2**.

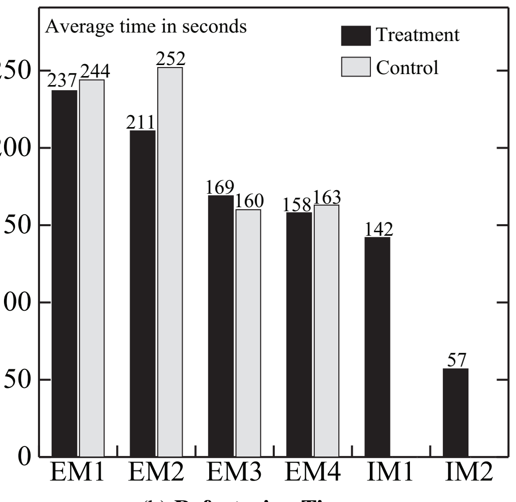
Q1. With GhostFactor, can developers refactor more correctly? We answer this research question by comparing the correctness of refactorings assisted by GhostFactor and those that were not. Figure 17a shows the number of correct refactorings for each task performed by participants in each group. EM stands for extract method and IM stands for inline method. Overall, we collected 24 refactoring assisted by GhostFactor, and 24 refactorings performed without GhostFactor's assistance.

23 of the 24 refactorings performed with GhostFactor were correct. In contrast, only 7 of the 24 refactorings performed without GhostFactor were correct. GhostFactor performed especially well when assisting inline method refactoring which no participant in the control group did correctly. GhostFactor improved our participants' likelihood of performing correct manual refactorings by 66.7% $((23-7)/24)$. To further evaluate the improvement, we performed the Mann-Whitney U test on the collected data, which is a statistic test that assumes neither that the data follows the normal distribution nor the minimum sample size [22]. We found p to be .001 (< .01), which indicates that **GhostFactor significantly improves the correctness of manual refactorings**.

Q2. With GhostFactor, can developers achieve correct refactorings more quickly? We answer this research question by comparing the time needed to perform correct refactorings with and without GhostFactor. Figure 17b plots the average time, in seconds, taken by participants in each group to correctly finish tasks. Since no participant in the control group performed the inline method refactoring correctly, Figure 17b does not include these tasks for participants in the control group. According to this data, the participants who used GhostFactor finished faster in three out of four



(a) **Refactoring Correctness.**



(b) **Refactoring Time.**

	EM1	EM2	EM3	EM4	IM1	IM2
Correct counts, Treatment	4	4	4	4	4	3
Correct counts, Control	1	1	2	3	0	0
Average time in seconds, Treatment	237	211	169	158	142	57
Average time in seconds, Control	244	252	160	163		

Figure 17: Study results summary.

tasks. To further evaluate the time difference, we again apply the Mann-Whitney U test. We found p to be .343 (> .01), suggesting that **GhostFactor may not significantly shorten the time needed to perform correct manual refactorings**.

6.3 Post-study Questionnaire

After participants completed the refactoring tasks, we asked them to take a post-study questionnaire. The post-study questionnaires, administered to participants in the treatment group, collected participants' opinions about their experiences with GhostFactor. Before they answered these questions, we used the screencast of their refactorings to show the participants which "IDE warnings" they encountered were issued by GhostFactor. This ensured that participants differentiated GhostFactor's warnings from other warnings in the Visual Studio IDE, such as those for undefined variables and syntax errors.

The post-study questionnaire asked participants to rate their agreement (from 1 to 5, 1 for not at all and 5 for totally agree) with the following statements:

- Refactoring warnings are useful in identifying refactoring errors.
- Refactoring warnings are a desirable feature in IDEs.

Overall, the median rating for participants' agreement with each of these statements was 4, suggesting that developers found GhostFactor useful in helping manual refactorings and that GhostFactor may be a desirable feature in IDEs.

The second part of the post-study questionnaire was an open question about how we could improve refactoring warnings. Participants raised several interesting points. We use their answers to answer the research question **Q3**.

Q3. How can we improve GhostFactor? One participant suggested that "refactoring warnings are not intuitively understandable". Like error messages from conventional refactoring tools [26], the warning messages issued by GhostFactor are difficult to understand in the current implementation. One way to tackle this problem is by giving developers code examples, or by visualizing these errors. To meaningfully compare the time to finish the refactorings, we intentionally disabled GhostFactor's quick fix suggestions during our study, but when developers use GhostFactor in the wild, the quick fix preview (as shown in the gray box in Figure 15) may help developers understand refactoring warnings better. However, better ways to convey warnings' meaning remains an open question for future exploration.

Another participant suggested that "refactoring warnings should come earlier than the refactoring's end". He would prefer if refactoring warnings reminded him of possible refactoring pitfalls before or while he refactored, instead of providing corrective messages after he has already made an error. While we agree that showing potential refactoring errors early may help developers perform correct refactorings faster, knowing a developer's intent to refactor before he actually starts refactoring is technically difficult, if not impossible.

To better answer this research question, we also examined the refactoring videos of those participants who used GhostFactor, but did not refactor correctly. One participant in the treatment group failed to complete IM2 correctly, as illustrated in Figure 17a. Delving into the causes of her error, we found that she accidentally deleted a statement in the method where another method should be inlined to. Even though her refactoring passed all of GhostFactor's condition checkers, our tool failed to detect that the mistakenly deleted statement changed the program's behavior. This observation suggests a need for new condition checkers that guarantee no code has been unintentionally removed by a manual refactoring.

6.4 Discussion

Our study showed that GhostFactor can effectively improve the correctness of manual refactorings. We next discuss some other interesting observations.

Learning Effects. The data in Figure 17 suggests that participants generally performed later refactoring tasks more correctly and faster than earlier ones. This observation holds for participants in both study groups. We speculate that this performance enhancement was due to learning effects: participants may have applied knowledge gained from earlier tasks to the later ones. The knowledge gained might include what code elements need more attention during manual refactoring, the meanings of refactoring warnings, and how to resolve refactoring warnings.

Attitude towards GhostFactor. We observed that participants sometimes over rely on GhostFactor warnings. One participant, when refactoring manually with GhostFactor, told the first author that *"[Warning] messages are quite informative, I feel like I am* *not really thinking"*. This statement suggests that she feels GhostFactor warnings are guiding her refactorings, rather than correcting them. Ironically, one of the reasons we designed GhostFactor was to decrease developers' reliance on conventional compiler warnings in refactoring; for this developer, GhostFactor became the tool that she relied on too much. We believe that manually inspecting GhostFactor warnings before addressing them is a better strategy than totally relying on them while refactoring.

In contrast to the over-reliant participant, another participant became much more careful when performing refactoring tasks after GhostFactor first warned him of a refactoring error. He manually examined the correctness of each complete refactoring even when GhostFactor found no errors. We speculate that using GhostFactor increased his awareness of the error-prone nature of manual refactoring. We did not anticipate this benefit to using GhostFactor.

Refactoring Time Improvement. Although the refactoring data we collected suggests that GhostFactor does not improve the time taken to refactor manually, we speculate that, as developers become expert GhostFactor users, they can use GhostFactor's assistance to refactor faster. Also, to reduce bias introduced by quick fixes, we disabled them in the study. We postulate that developers can correct refactoring errors faster by applying quick fixes. Furthermore, GhostFactor helps developers assess the correctness of refactorings, potentially saving time spent testing and inspecting refactored code. We believe that a more comprehensive, long-term study of GhostFactor use may show that using GhostFactor can improve the efficiency of manual refactorings.

6.5 Threats to Validity

Although the study gives us confidence about the usefulness of GhostFactor, several threats need to be considered when interpreting the study results.

The first threat is the limited number of refactoring tasks as well as the recruited participants (6 tasks for each of 8 participants), which externally threatens the results' generalizability to other developers' refactoring tasks in the wild. Also, by solely studying extract method and inline method refactorings, we cannot conclude that GhostFactor can perform equally well when helping developers perform other types of refactorings.

Another threat is that the tasks we picked can always lead to refactoring mistakes that GhostFactor detects. In spite that these mistakes, according to our previously collected data, are frequent among developers, failing to consider other refactorings performed in the wild may lead to the hasty conclusion [15].

The third threat lies in other reasons why participants in two groups refactor with varied correctness and speed. GhostFactor may not be the only reason for the differences. Since we randomly assign participants to the two groups, the participants in the treatment group may by themselves have better programming skills than those in the control group, allowing them to refactor more correctly and quickly regardless of GhostFactor's existence. To eliminate this threat, in the future, we plan to assign participants according to their reported expertise to the groups under study so that the knowledge gap between the groups are minimized.

7. FUTURE WORK

We have implemented the GhostFactor technique as a plug-in Visual Studio IDE. Although the initial study yields promising results, we plan to further explore the possibility of adopting GhostFactor as an integral part of modern IDEs. In this section, we summarize possible future work.

Better tool. At this point, GhostFactor only supports three types of refactorings, namely extract method, change method signature and inline method. Existing IDEs usually support more than 20 different refactoring types [4]. To improve the usefulness of GhostFactor, adding more refactoring types is necessary. Also, as the cornerstone of GhostFactor, we plan to improve the refactoring detection algorithms in terms of reducing the false negative and the false positive rates; thus developers can benefit from the tool more often without being frequently disrupted by spurious warnings.

Currently, GhostFactor only supports checkers for refactoring errors that were manifested in our previously conducted studies. However, these errors may be only a small fraction of all the refactoring errors developers could make. In order to better guarantee refactoring correctness, we need to investigate more refactoring error patterns. We plan to apply data mining techniques to software repositories to find these error patterns. In the future, we also plan to summarize a catalog for frequent refactoring errors like FindBugs does for commonly occurring defects.

Richer Study. To evaluate GhostFactor, we conducted an in-lab study session participated by 8 developers and collected limited amount of data. To further investigate the benefit of GhostFactor as well as its limitations, a field study to observe what happens when professional developers use GhostFactor under in real-world development can be beneficial. In such a setting, developers frequently interleave refactorings with non-refactorings, potentially leading to GhostFactor's false positives and false negatives; measuring these two indicators are important to evaluate GhostFactor's effectiveness.

Another goal of this richer study is to investigate the refactoring detection algorithms. In our controlled human study, the detection algorithms successfully detected all of the manual refactorings performed by the participants. However, when used in the wild, the detection algorithms' performance is still an open question, such as their precision and recall. The richer study may also compare GhostFactor with other tools assisting developers' refactorings, such as DNDRefactoring [21] and WitchDoctor [12], in terms of the usability and the usefulness. Although these tools assist refactoring in different ways, such a study could compare developers' effort saved during refactoring.

8. CONCLUSION

Manual refactorings are error-prone. Although refactoring tools are available to help, they require the developer to adapt herself to the tool. To help create a tool that is adapted to the developer, we proposed and implemented a novel technique called GhostFactor that runs in the background of IDEs and warns developers about refactoring errors in the same manner of compiler warnings, a way that most developers accustom to. GhostFactor allows developers to benefit from refactoring tools without explicitly invoking these tools, and works independent of how developers refactor by hand. To evaluate the effectiveness of GhostFactor, we conducted a human study with eight software developers who performed manual refactorings with or without the help of GhostFactor. Our evaluation results provide evidences that GhostFactor improves the correctness of manual refactorings and suggests promising future directions.

9. ACKNOWLEDGMENTS

We thank the study participants for participating. We also thank the comments from the Development Liberation Front group members Titus Barik, Michael Bazik, Brittany Johnson, Kevin Lubick, John Majikes, and Yoonki Song. Special thanks to Jim Witschey. This work is funded by a Microsoft Software Engineering Innovation Foundation Award and a Google Research Award.

10. REFERENCES

[1] CamStudio: Free Streaming Video Software. http://camstudio.org/, 2013.

[2] DotNumerics Open Source Project. http://www.dotnumerics.com/, 2013.

[3] Microsoft Roslyn CTP. http://goo.gl/u5XHgq, 2013.

[4] The Eclipse Foundation. http://www.eclipse.org/, 2014.

[5] The Visual Studio IDE. http://www.visualstudio.com/, 2014.

[6] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. A two-step technique for extract class refactoring. In *Proceedings of the international conference on Automated software engineering*, pages 151–154, 2010.

[7] G. Bavota, A. De Lucia, and R. Oliveto. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, 84(3):397–414, 2011.

[8] G. Bavota, R. Oliveto, A. De Lucia, G. Antoniol, and Y. Gueheneuc. Playing with refactoring: Identifying extract class opportunities through game theory. In *Proceedings of the International Conference on Software Maintenance*, pages 1–5, 2010.

[9] K. Beck. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[10] 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.

[11] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proceedings of the ACM SIGSOFT symposium on The foundations of software engineering*, pages 185–194, 2007.

[12] 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*, pages 222–232, 2012.

[13] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[14] A. Garrido and J. Meseguer. Formal specification and verification of java refactorings. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation*, pages 165–174, 2006.

[15] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *Proceedings of the International Conference on Software Engineering*, pages 211–221, 2012.

[16] X. Ge and E. Murphy-Hill. Appendix to Manual Refactoring Changes with Automated Refactoring Validation. http://goo.gl/0ty6J1, 2014.

[17] J. Hannemann, T. Fritz, and G. C. Murphy. Refactoring to aspects: an interactive approach. In *Proceedings of the OOPSLA workshop on eclipse technology eXchange*, pages 74–78, 2003.

[18] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004.

[19] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 50:1–50:11, 2012.

[20] 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 International Conference on Software Maintenance*, pages 369–378, 2005.

[21] Y. Y. Lee, N. Chen, and R. E. Johnson. Drag-and-drop refactoring: Intuitive and efficient program transformation. In *Proceedings of the International Conference on Software Engineering*, pages 23–32, 2013.

[22] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Annals of Mathematical Statistics*, 18:50–60, 1947.

[23] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Journal of software maintenance and evolution: Research and practice. *J. Softw. Maint. Evol.*, 17(4):247–276, 2005.

[24] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi. Does refactoring improve reusability? In *Proceedings of International Conference on Reuse of Off-the-Shelf Components*, pages 287–297, 2006.

[25] E. Murphy-Hill. Programmer-friendly refactoring tools. Dissertation Proposal, 2007.

[26] E. Murphy-Hill and A. P. Black. Programmer-friendly refactoring errors. *IEEE Transactions on Software Engineering*, 38(6):1417–1431, 2012.

[27] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[28] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[29] J. L. Overbey and R. E. Johnson. Differential precondition checking: A lightweight, reusable analysis for refactoring tools. In *Proceedings of the International Conference on Automated Software Engineering*, pages 303–312, 2011.

[30] 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.

[31] J. Ratzinger, M. Fischer, and H. Gall. Improving evolvability through refactoring. In *Proceedings of the international workshop on Mining software repositories*, pages 1–5, 2005.

[32] M. Schäfer and O. de Moor. Specifying and implementing refactorings. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, pages 286–301, 2010.

[33] M. Schäfer, M. Verbaere, T. Ekman, and O. de Moor. Stepping stones over the refactoring rubicon – lightweight language extensions to easily realise refactorings. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 369–393, 2009.

[34] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8(1):89–120, 2001.

[35] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the International Conference on Software Engineering*, pages 233–243, 2012.

[36] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise identification of problems for structural test generation. In *Proceedings of the International Conference on Software Engineering*, pages 611–620, 2011.

[37] 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.