

# BeneFactor: a Flexible Refactoring Tool for Eclipse

Xi Ge Emerson Murphy-Hill

Department of Computer Science, NC State University, Raleigh, NC, USA 27695

xge@ncsu.edu, emerson@csc.ncsu.edu

## Abstract

Although broadly available in major software development environments, refactoring tools are still underused. One of the reasons for this underuse is that existing refactoring tools assume that a developer recognizes that she is going to refactor before she even begins. In this paper, we present a flexible refactoring tool called BeneFactor that can be invoked after refactoring begins to safely complete a refactoring change.

**Categories and Subject Descriptors** D.2.3 [Software Engineering]: Coding Tools and Techniques—Object-oriented programming; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering

**General Terms** Languages

**Keywords** refactoring

## 1. Introduction

Refactoring is both an effective and commonplace way of improving non-functional requirements. Empirical studies of refactoring have found that it can improve maintainability [2] and reusability [3]. Not only does existing work suggest that refactoring is useful, but it also suggests that refactoring is a frequent practice [4]. Cherubini and colleagues' survey indicates that developers rate the importance of refactoring is equal to or greater than that of understanding code and producing documentation [1].

However, refactoring by hand is labor-intensive and 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. These tools have been integrated into most popular development environments, making them available in a variety of programming languages to a large population of developers. In spite of the wide availability, our previous study shows that refactoring tools are underused; according to two case studies, about 90% of refactorings are performed by hand [4]. As a result, this underuse of refactoring tools unnecessarily slows down software development and increases the potential for introducing new software defects.

One of the reasons for this underuse is that existing refactoring tools assume that a developer realizes that she is going to refactor

before she begins. Our previous study indicates that this assumption may be false because a developer may have already started a refactoring manually by the time she realizes that she is refactoring. One software developer outlined this situation in our interview [4] as, "my hands start doing copy-paste... 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... I just finish it and continue." In these situations, existing refactoring tools fail.

To address this issue, we present a novel refactoring tool named BeneFactor that can be invoked after manual refactoring begins. BeneFactor detects manual refactoring behavior, prompts developers to use automatic refactoring, and finishes the incomplete refactoring automatically. Currently, BeneFactor supports the rename refactoring and the extract method refactoring. Let us compare conventional refactoring tools against BeneFactor using an example.

Suppose Grace is a developer who works on the Mylyn project [5]. To improve understandability, she wants to change the name of the local variable *event* to *intEvent* in the code snippet showed in Figure 1. She starts doing this task manually from the last reference of *event* backwards through her code. After changing seven references to *intEvent*, she realizes that she is refactoring, but in order to use a conventional refactoring tool, she needs to either back out and undo her changes, or finish the refactoring manually.

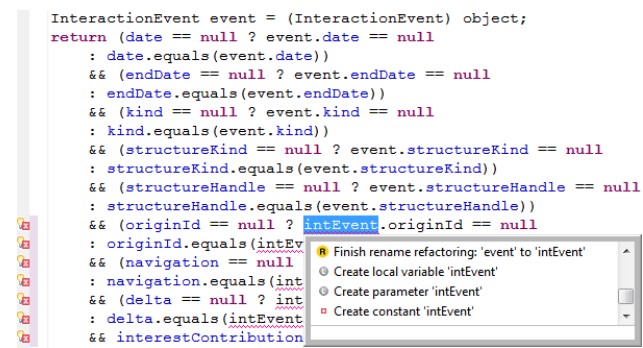


Figure 1. BeneFactor in use.

On the other hand, with BeneFactor, after Grace changes references of *event* to *intEvent*, BeneFactor detects that she is renaming a local variable. Therefore, it places a quick fix refactoring option on the problem markers at the lines where Grace did name changes, as illustrated in Figure 1, suggesting that she allow BeneFactor to finish this rename refactoring automatically. When Grace realizes that she is performing a rename refactoring, she then clicks the quick fix refactoring option. Finally, BeneFactor automatically renames all references of *event* to *intEvent*.

## 2. Approach

BeneFactor has two components: refactoring detection and code modification. The refactoring detection component runs silently in background of Eclipse, while monitoring code changes and a developer's editing actions. If a potential refactoring behavior is detected, the refactoring detection component adds a quick fix to automatically finish the refactoring that she has started. The code modification component takes the responsibility of finishing the refactoring automatically and safely.

### 2.1 Refactoring Detection

Most existing refactoring tools force a developer to recognize that she is going to refactor even before she begins. In order to shift this cognitive burden from the developer to our refactoring tool, we created a refactoring detection component for BeneFactor. In contrast to existing refactoring detection tools (for example, RefacLib [7] and RefFinder [6]), our refactoring detection component detects refactoring live rather than in the version control system. Once the refactoring detection component detects that a refactoring has begun, it can assist the developer in completing the refactoring automatically. We use two strategies to detect that a developer is refactoring: code-change based detection and action based detection.

**Code-change based refactoring detection:** The code-change strategy detects that a developer is refactoring by analyzing code base changes. We capture the changes as changes to the abstract syntax tree (AST), such as node addition, deletion, update and move. For example, by monitoring AST changes, BeneFactor is able to detect various kinds of refactorings, such as renaming a local variable. Renaming a local variable is detected whenever the developer updates an AST node whose type is local variable. The more references to a local variable that are changed, the more confidence BeneFactor has that the developer is performing a rename refactoring.

**Action based refactoring detection:** The action based strategy detects that a developer is refactoring by analyzing her editing actions upon the code base. Our refactoring detection component monitors a developer's editing actions, such as copy, paste, cut, and select. Once a sequence of editing actions matches a typical manual refactoring workflow, the developer is likely refactoring. For example, if a developer cuts a contiguous set of program statements, BeneFactor infers that she is probably starting an extract method refactoring.

### 2.2 Code modification

Once a developer's manual refactoring has been detected, BeneFactor allows her to automatically finish the refactoring that she has started. This user intervention allows the developer to safely perform the refactoring with the help of a refactoring tool, but without having to undo her changes. After the developer activates the quick fix, BeneFactor invokes the code modification component to finish the refactoring automatically. The code modification component consists of three steps, which are code recovery, information collection, and change creation. This component makes significant use of existing APIs from the Eclipse Language Tool Kit (LTK).

The **code recovery** step must modify the code so that the existing LTK refactoring APIs can be applied. Because the existing LTK refactoring API can only refactor an un-modified code base, BeneFactor must transform the partially-refactored code base to its earlier state before manual refactoring began. For example, when a developer allows BeneFactor to automatically finish an ongoing extract method refactoring, it is likely that the code statements to be extracted have already been cut out of the code base. BeneFactor must undo the cut command before using the existing API.

The **information collection** step collects configuration information when performing automatic refactoring. BeneFactor collects information by traversing ASTs of code base. Examples of such information include original and new name in rename refactoring and the range of code statements to be extracted in extract method refactoring.

The **change creation** step makes the actual code base change. In this step, BeneFactor first performs a set of precondition checks. These checks ensure that automatic refactoring can be successfully finished. After passing all the checks, BeneFactor calculates the actual code change of the refactoring and uses the code change to modify the code base.

## 3. Future Work

We plan to alter current condition checking mechanism of BeneFactor. Rather than checking all the preconditions before a change is made, we plan to adopt a more flexible precondition checking mechanism for BeneFactor. It will check the preconditions live, as changes are being made (either automatically or manually), so that the developer is kept aware of, but not forced to resolve, potentially behavior-modifying changes.

We also plan to add refactoring warnings to BeneFactor. According to our observations, developers sometimes prefer to make a change and let the compiler warn them about what pieces of code must be changed [4]. This is a convenient strategy, yet the compiler's warnings are not sufficient to tell the developer all the places that she must modify to complete the refactoring. In order to complement these insufficient compiler warnings, we plan to add refactoring warnings that augments the development environment by informing the developer of all the code that must be modified.

## 4. Demonstration Script

The presenter will demonstrate BeneFactor by refactoring an open source project using rename local variable and extract method. Each type of refactoring will include several demonstration cases in order to show how BeneFactor can help developers when manual refactoring workflow varies. For each case, the presenter will demonstrate how BeneFactor detects manual refactoring begins, how BeneFactor prompts developers to invoke automatic refactoring, and how the code base will look like after invoking the automatic refactoring.

## Acknowledgments

Work partially supported by an NCSU FRPD grant. Thanks to Moin Ayazifar, Quinton DuBose, and Suprit Patankar for their participation in this project.

## References

- [1] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In Proc. CHI, pages 557-566, 2007.
- [2] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. A case study in refactoring a legacy component for reuse in a productline. In Proc. ICSM, pages 369-378, 2005.
- [3] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi. Does refactoring improve reusability? In Proc. ICSR, pages 287-297, 2006.
- [4] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. IEEE TSE, 2011.
- [5] Eclipse Mylyn Open Source Project, 2011. <http://www.eclipse.org/mylyn/>.
- [6] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In Proc. ICSM, pages 1-10, 2010.
- [7] K. Taneja, D. Dig, and T. Xie. Automated detection of API refactorings in libraries. In Proc. ASE, pages 377-380, 2007.