Toward Practical Application of Program Refactoring

A Dissertation Submitted to
The Graduate School of Information Science and Technology
of Osaka University
for the degree of

Doctor of Philosophy

in

Information Science and Technology

by

Yoshio Kataoka

January 16, 2006

Abstract

Program refactoring — transforming a program to improve readability, structure, performance, abstraction, maintainability, or other characteristics — is not applied in practice as much as might be desired. Although there are various possible reasons why program refactoring is not so popular, one deterrent is the cost of detecting candidates for refactoring and of choosing the appropriate refactoring transformation. Another problem is that there are few quantitative evaluations of its impact to the software maintainability. It is sometimes difficult to judge whether the refactoring in question should be applied or not without knowing the effect accurately.

This series of researches mainly focuses on how to support program refactoring to encourage practical applications of the technique. The first half of this thesis demonstrates the feasibility of automatically finding places in the program that are candidates for specific refactorings. The approach uses program invariants: when particular invariants hold at a program point, a specific refactoring is applicable. Since most programs lack explicit invariants, an invariant detection tool called Daikon is used to infer the required invariants. We developed an invariant pattern matcher for several common refactorings and applied it to an existing Java code base. Numerous refactorings were detected, and one of the developers of the code base assessed their efficacy.

The latter half proposes a quantitative evaluation guideline to measure the maintainability enhancement effect of program refactoring. We focused on the module coupling metrics and the module cohesion metrics to evaluate the refactoring effect. Comparing the metrics before and after the refactoring, we could evaluate the degree of maintainability enhancement. As for coupling metrics, we carried out an experiment and found our method was really effective to quantify the refactoring effect and helped us to choose appropriate refactorings.

List of Major Publications

1. Yoshio Kataoka, Shinji Kusumoto and Katsuro Inoue, "Supporting Refactoring Using Invariant," IPSJ Transaction, Vol.46, No.5, pp.1211–1221, May 2005.

2. Yoshio Kataoka, Shinji Kusumoto and Katsuro Inoue, "A Quantitative Evaluation of Refactoring Effect," Transactions on Information and Systems (Submitted).

3. Yoshio Kataoka, Takeo Imai, Hiroki Andou and Tetsuji Fukaya, "A Quantitative Evaluation of Maintainability Enhancement by Refactoring," Proceedings of International Conference on Software Maintenance, pp.576–585, Montreal, Canada, October 2002.

4. Takeo Imai, Yoshio Kataoka and Tetsuji Fukaya, "Evaluating Software Maintenance Cost Using Functional Redundancy Metrics," Proceedings of 26th International Computer Software and Applications Conference, pp.299–306, Oxford, England, August 2002

5. Yoshio Kataoka, Michael D. Ernst, William G. Griswold and David Notkin, "Automated Support for Program Refactoring using Invariants," Proceedings of International Conference on Software Maintenance, pp.736–743, Florence, Italy, November 2001

6. Yoshio Kataoka, Masayuki Hirayama, Jiro Okayasu and Tetsuji Fukaya, "An Approach to Reverse Quality Assurance with Data-Oriented Program Analysis," Proceedings of Asia Pacific Software Engineering Conference 1995, pp.324–332, Brisbane, Australia, December 1995

Acknowledgements

During the course of this work, I have been fortunate to have received assistance from many individuals. I would especially like to thank my supervisor Professor Katsuro Inoue and Professor Shinji Kusumoto for their sustainable supervision. I could not have come this far without their support and encouragement for this work.

I am also very grateful to Professor Tohru Kikuno who is the member of my thesis review committee for his invaluable comments and helpful advices.

I also would like to thank Professor Koji Torii for his initiation to this research field and his supervision during my undergraduate years and master course. My perspective had been broaden by his advices and encouragements from time to time.

I also would like to thank Professor David Notkin for his supervision during my undergraduate years and my visiting University of Washington. My original motivation of research in this field was his word, "make software engineering true engineering."

I also would like to thank Professor Mike Ernst for his continuous advice and collaboration during my visiting University of Washington. He always give me very productive and insightful advices on my research.

I would also like to thank Ms. Yoshimi Katagiri for her administrative support during my attending a Ph.D. course. I would also like to appreciate all the member of Inoue laboratory at Osaka University for their support of my research.

I would like to appreciate Dr. Masami Akamine, Dr. Shun Egusa and Dr. Naoshi Uchihira for their understanding and support for my attending a Ph.D. course.

I would like to thank to Research Planning Office members, System Engineering Laboratory members and Software Engineering Center members of Toshiba Corporation for their support of my research.

Last but not least, I would like to thank to my family for every support and encouragement. Nobuko, she has been literally supporting me for everything. And

Yusuke, he has always been cheering me up. There's no proper word to express my gratitude to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Background

Nowadays softwares are getting more and more important for various systems and devices around us. At the same time, required functionalities and performance are also getting higher and higher. In other words, it is getting more and more important to develop higher-quality software as quick as possible. Naturally speaking, developing all those softwares from scratch is not a realistic idea at all. We do have some reasonable method to develop software systems without writing a lot of new pieces of code.

Generally speaking, however, it is rather hard to utilize existing software codes. Most of the case those source codes are not supposed to be used for different systems other than the original target system. As for rather reusable source codes, there could always be an "unpredictable" need of customize because anticipating any possible applications at the development stage is impossible. For instance, an application software installed in a cell phone system would interact not only with those softwares in the cell phone but also a number of software components and/or hardware devices over the network. It is obviously impossible to presume those every potential interaction opponents at the development stage. Therefore customizing existing softwares is inevitable today's software system development.

In conventional way of software development, especially in enterprise software development sites, nothing more important than implementing required functionality as quickly as possible. In other words, so called non-functional requirements such as maintainability, portability, etc. are less cared or even ignored intentionally due to delivery constraint. Such softwares often require enormous amount of time to acquire necessary information, or even have developer extract wrong information to customize. Moreover, if the modularization is not done properly, there might be an unexpected side effect to other modules/functionalities which cannot be traced very easily.

There is another problem. Customizing an existing software induces a potential deterioration problem. Although softwares are free from aged-deterioration, qualities of software can be deteriorated by various modification including adding a new functionality, fixing a bug and other maintenance activities. Among a number of qualities, non-functional qualities like maintainability and portability are often impaired because most modifications are based on functional quality requirements. As a result, such deterioration often makes software customization rather hard.

1.2 Program Refactoring

Program refactoring is a technique in which a software engineer applies well-defined source-level transformations with the goal of improving the code's structure and thus reducing subsequent costs of software evolution. Initially developed in the early 1990s [32, 18, 31, 21], refactoring is increasingly a part of mainstream software development practices [16]. As just one example, one of the basic tenets of Extreme Programming [4] is to refactor on a continual basis, as a fundamental part of the software development process.

Program refactoring is a technique to improve readability, structure, performance, abstraction, maintainability, or other characteristics by transforming a program. Program refactoring can be used to diagnose the quality and improve customizability of the target software. Highly motivated and skillful software developers actually apply this technique in practice. And, of course, Fowler's work organizing the program refactoring catalog encourages many software developers to apply program refactoring to their software development processes.

On the other hand, program refactoring is not a very simple technique to apply. Firstly, it is often very difficult to identify which part of the program should be refactored without understanding basic notion of software quality. Secondly, softwares in practice are so large in scale that developers have to spend decent amount of time to inspect the whole target software. In addition, it is hardly possible to secure that there is no slipped-out. And finally, program refactoring potentially introduces a new bug in the source code because it does require a source code modification.

In fact, refactoring is not applied in practice as frequently as might be beneficial. There are a number of reasons for this, including managerial (such as, "we need to add features to ship the product, and refactoring doesn't directly contribute to that") and technical (such as, "refactoring might break a subtle property of the system, which is too dangerous"). There are a number of tools to help overcome some of these problems: most of these automate the process of safely applying a refactoring that an engineer has determined is appropriate (see Section 3.2).

The main objective of this thesis is to overcome those potential problems regarding program refactoring and make program refactoring more and more practical technique. In this thesis, two major topics will be discussed. One is how to identify which part of the program to be refactored. The other one is how to evaluate the effectiveness of the program refactoring.

1.3 Refactoring Process Model

We analyzed the program refactoring process and found we need at least the following three phases to perform appropriate refactorings.

1. Identification of refactoring candidates
2. Application of refactoring
3. Validation of refactoring effect

Fig. 1.1 shows the diagram of the refactoring process.

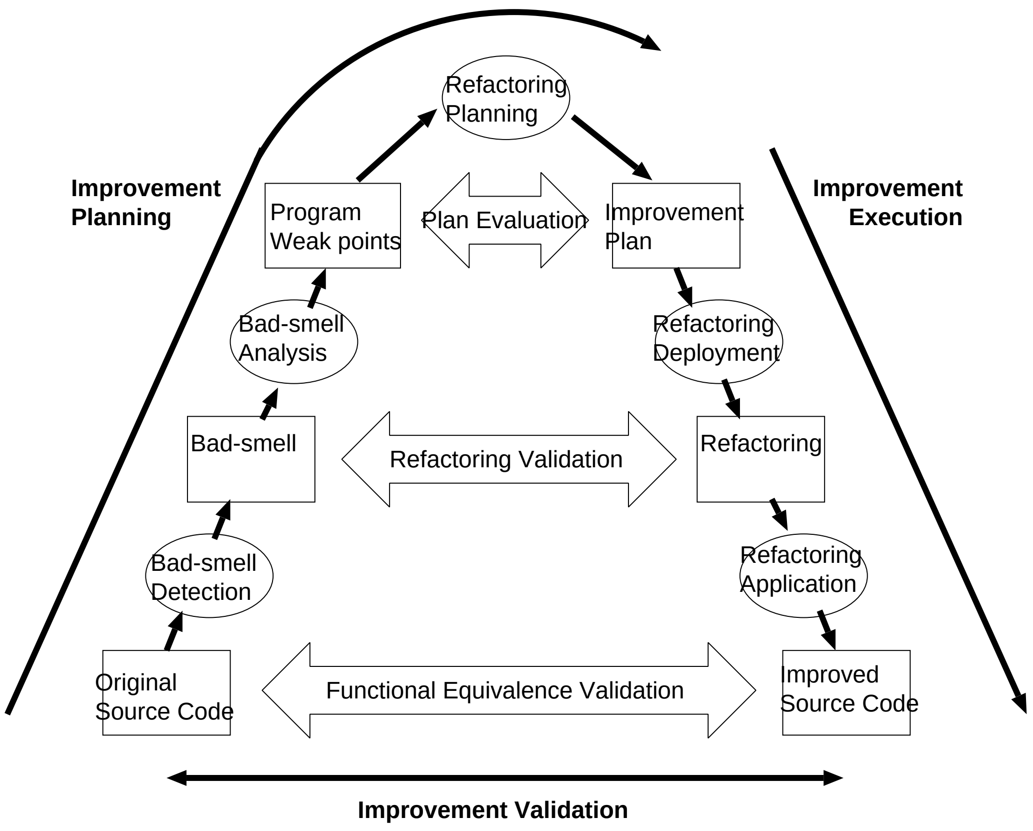

Figure 1.1: Refactoring Process

There are two axes in the diagram. The horizontal axis stands for the process timeline and the vertical axis stands for the degree of abstraction of the product (or roles in charge). Refactoring process always begins from the source code level analysis. Verification and validation of refactorings have to be evaluated at the source code level. This level is also referred as *developer level.* Developers need to confirm that the observable behavior is the same before and after the refactoring by testing and other means(*functional equivalence validation*).

The next level is referred as *analyst level.* At this level analysts verify the refactoring against "bad-smell" detected by the source code analysis(*refactoring validation*).

The uppermost level is referred as *manager level.* The manager has to strategically decide which refactoring should be carried out and which should not(*plan evaluation*). Actually there is a strict trade off between the cost to apply the refactorings and the effect derived from the refactoring. From practical point of view, it is very important to emphasize that the strategic decision made at this level would greatly effect the tactical approaches of refactoring application at the lower levels. A manager should organize the possible improvement alternatives submitted by analysts to establish the refactoring plan. Otherwise the total quality improvement activity will lead nowhere near the desirable result.

As introduced earlier, the refactoring process consists of the following three subprocesses.

1. Identification of refactoring candidates
   This subprocess appears as the bottom–left to top–middle flow in Fig. 1.1. Starting with the identification of program points to be refactored, it includes organization of refactorings and selection of refactorings to be applied.

2. Application of refactoring
   This subprocess appears as the top–middle to bottom–right flow in Fig. 1.1. It includes the ordering of each refactoring according to the priority in terms

of cost–effect trade–off by the analyst and the actual code modification by the developer.

3. Validation of refactoring effect

   This subprocess corresponds to the arrows connecting the identification flow and the application flow in Fig. 1.1. At the developer level, mainly the functional equivalence before and after the refactoring should be verified. At the analyst level, the intended effect should be validated. At the manager level, the cost–effect trade–off should be substantiated.

1.4 Major Activities in Refactoring Process

1.4.1 Planning — Identification of Refactoring Candidates

Extract Bad-smells (developer level)

First of all, developers should identify the causes that deteriorate readability and/or maintainability of their own source code. "Bad-smell" is a program property or characteristic that implies one or more (potential) problems in the source code[16]. There are various smells from simple ones like "Large Class" and "Long Method" to fairly complicated ones like "Parallel Inheritance Hierarchies" and "Inappropriate Intimacy."

Identify Bad-smells' causes (analyst level)

Next thing to be done is to identify the causes of bad-smells. This might be trivial for some simple bad-smells like "Large Class." In some cases, however, it requires some deeper analysis of the target code to identify what causes a certain bad-smell after all. The cause does not necessarily exist anywhere near the smell itself. It is often observed that introducing just one new method solves several bad-smells at once.

Design quality improvement plan (manager level)

As mentioned above, the cost required to eliminate single bad-smell is different from one another. One bad-smell will be eliminated by just introducing a new private method while other bad-smell will require a number of modifications here and there. Designing appropriate improvement plan often requires fairly high level decision making concerning with cost and delivery.

1.4.2 Improvement — Execution of Refactoring

Select optimal refactoring set (analyst level)

Once the improvement plan is established, next thing to be done is to identify an optimal set of refactoring since some of them may conflict each other. Based on the target source code analysis, analyst should find an optimal set of refactoring and parcel out the refactorings to the developers properly.

Modify source code (developer level)

Each developer applies the apportioned refactorings to her/his own code. Althogh the refactoring set is supposed to be consistent by analyst, each developer should be careful not to violate each other.

1.4.3 Evaluation — Validation of Refactoring

Validate improvement plan (manager level)

Manager should validate that designed improvement plan correctly grasp the targeted weak points.

Validate refactoring candidates (analyst level)

Analyst should validate that derived refactorings correctly mend the targeted bad-smells.

Validate functional equivalence (developer level)

Developer should validate that their program does not change its observable behavior after the refactoring.

1.5 Research Objective

The main objective of this series of researches is how to support the refactoring process effectively so that program development projects can adopt the process easily and enhance the maintainability of the target software. Although there are several distinguished research results for some activities of the process, it is still far from being the comprehensive support of the refactoring process we stated above.

For instance, there are some fundamental studies on the refactoring application phase including Griswold and Notkin's work[21]. Table 1.1 summarizes the current status of the refactoring process support.

Among these research areas, Chapter 3 introduces a tool to support engineers in refactoring software: automatically finding candidate refactorings. The recommended manual method of identifying beneficial refactorings is to observe design

Table 1.1: Status quo of refactoring process support

Subprocess	Activity	Support
Planning	bad–smell detection	partly
	bad–smell analysis	partly
	refactoring planning	partly
Validation	plan evaluation	no
	refactoring validation	no
	functional equivalence validation	no
Execution	refactoring deployment	no
	refactoring application	partly

shortcomings manifested during development and maintenance [17]. Unfortunately, design problems may be overlooked or ignored by a programmer, particularly under deadline pressures and the intellectual demands of implementing correct changes.

The technology introduced in Chapter 3 for identifying refactoring candidates uses program invariants: a particular pattern of invariants at a program point indicates the applicability of a specific refactoring. This use of program invariants is complementary to other approaches such as human examination or pattern-matching over the source code.

Chapter 4 introduces a quantitative evaluation guideline to measure the maintainability enhancement effect of program refactoring. This research focused on the module coupling metrics and the module cohesion metrics to evaluate the refactoring effect. Comparing the metrics before and after the refactoring, we could evaluate the degree of maintainability enhancement.

Chapter 2

Program Refactoring

This chapter introduces the basic idea of program refactoring with several examples which are actually referred and used in this series of researches more precisely.

2.1 Defining Refactoring

Fowler defines refactoring as follows [16].

Refactor(verb): *to restructure software by applying a series of refactorings without changing its observable behavior.*

Refactoring(noun): *a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.*

Fowler clarifies these definitions in two points. First, the purpose of refactoring is to make the software easier to understand and modify. Although there are a number of possible changes one can make in software that make little or no change in the observable behavior, only changes made to make the software easier to understand are refactorings. Fowler contrasts refactoring with performance optimization.

That is, neither refactoring nor performance optimization changes the behavior of a component; it only changes the internal structure. The major and very important difference is that performance optimization often makes code harder to understand while refactoring does make code easier to understand.

Second, refactoring does not change the observable behavior of the software. Fowler emphasizes this point like this; any user, whether an end user or another programmer, cannot tell that things have changed. Fowler also introduces Kent Beck's metaphor of *two hats* to support this point. A developer cannot add function and refactor at the same time. She or he have to wear one hat at a time, either "adding function" hat or "refactoring" hat. As the development progresses, the developer should swap the hats frequently. Unless such an obvious change of state of mind, neither adding function nor refactoring will take effect.

2.2 Motivations for Refactoring

Fowler introduces several motivations for refactoring. First, refactoring will improve the design of software. As mentioned earlier, the design of the program may be deteriorated as people change code — changes to realize short-term goals or changes made without a full comprehension of the design of the code. Refactoring helps reorganizing the code and its design so that non-functional qualities like maintainability and portability are improved and enhanced.

Second, refactoring will make software easier to understand. This notion implies two different meanings. One perspective is very straightforward from the definition. Since refactoring is supposed "to make it(program) easier to understand," the resulted program after refactoring will become easier to understand. As for the other perspective, Fowler introduces the following story[16].

This understandability works another way, too. I use refactoring to help

> me understand unfamiliar code. When I look at unfamiliar code, I have to try to understand what it does. I look at a couple of lines and say to myself, oh yes, that's what this bit of code is doing. With refactoring I don't stop at the mental note. I actually change the code to better reflect my understanding, and then I test that understanding by rerunning the code to see if it still works.

Third, refactoring will help find bugs. As seen in the previous quoted story, refactoring itself is a process to understand what the code does. That does not necessarily means that one can only understand what the code actually does but also what the code should do, which means one has spotted a bug.

And finally, refactoring will help program faster. All the earlier points come down to this conclusion. One of the reason why refactoring is not so popular is that most of developers feel that refactoring is an extra work other than their required development task. In fact, however, appropriate refactoring will actually save maintenance effort greatly, since refactoring stops the design of the system from deterioration.

2.3 Refactoring Catalog

The original Fowler's refactoring catalog in [16] contains 72 refactorings classified into seven large categories. After the publication of the text book, a number of additional refactoring techniques have been proposed. There also several information update those in the original text book. Today total 93 refactorings have been defined. Those refactorings can be referred at `http://www.refactoring.com/catalog`.

Since each refactoring name is fairly self-descriptive, here only shows the list of refactorings for each category save those refactorings actually used in this research.

2.3.1 Composing Methods

Refactorings belonging to this category basically intend to package code properly. Most refactorings in this category are useful when one found large methods.

There are two refactorings that are dealt in this thesis: "Extract Method" and "Replace Temp with Query."

Extract Method

"Extract Method" is intended for "a code fragment that can be grouped together." [16, p. 110] The refactoring turns the fragment into a method whose name explains the purpose of the method. The rationale is that if there is a very large method that does require some comment to understand its purpose, turning a fragment of code into its own method will increase the understandability of the method. Fowler also emphasize the importance of selecting an appropriate short name for the extracted method.

Replace Temp with Query

"Replace Temp with Query" is intended for "a temporary variable that holds the value of an expression" [16, p. 120]. The refactoring extracts the expression into a method and replaces uses of the temporary by method calls. The rationale is that the expression may be used in multiple places and that eliminating temporary variables can enable other refactorings. In essence, this is the user-level inverse of a compiler's application of the common subexpression elimination optimization.

The other refactorings belonging to this category are the followings.

- Inline Method
- Inline Temp

- Introduce Explaining Variable
- Split Temporary Variable
- Remove Assignments to Parameter
- Replace Method with Method Object
- Substitute Algorithm

2.3.2 Moving Features Between Objects

It is often rather difficult to place a certain functionality and data into an appropriate class/object for the first time. Refactorings belonging this category help developers to maintain their objects well-organized throughout the development phase.

"Extract Class" is dealt in this thesis as an example.

Extract Class

"Extract Class" is intended for "one class doing work that should be done by two." [16, p. 149] This refactoring create a new class and move the relevant fields and method from the old class into the new class. A large class with a lot of data and methods is very difficult to understand and reuse. Another potential problem is that such a class will increase inter-class coupling complexity. The rationale is that by separating such a large class into two (or more), you will get two (or more) handy classes which are easier to understand. In addition, that separation will decrease coupling complexity among classes.

The other refactorings belonging to this category are the followings.

- Move Method

- Move Field
- Inline Class
- Hide Delegate
- Remove Middle Man
- Introduce Foreign Method
- Introduce Local Extension

2.3.3 Organizing Data

Refactorings belonging to this category let developers gradually used to the basic object-oriented programming paradigm. Typical examples may be "Replace Data Value with Object" and "Replace Array with Object." "Replace Data Value with Object" let developer turn dumb data into a sophisticated object with appropriate methods. When there is an array acting as a data structure, "Replace Array with Object" will help turning the array into an appropriate object.

Although this thesis does not deal with any of them, there are following refactorings in this category.

- Self Encapsulate Field
- Replace Data Value with Object
- Change Value to Reference
- Change Reference to Value
- Replace Array with Object
- Duplicate Observed Data

- Change Unidirectional Association to Bidirectional
- Change Bidirectional Association to Unidirectional
- Replace Magic Number with Symbolic Constant
- Encapsulate Field
- Encapsulate Collection
- Replace Record with Data Class
- Replace Type Code with Class
- Replace Type Code with Subclasses
- Replace Type Code with State/Strategy
- Replace Subclass with Fields

2.3.4 Simplifying Conditional Expressions

Refactorings belonging to this category might be familiar to many developers. Actually they are all conventional programming technique before the appearance of object-oriented paradigm save a few refactorings like "Replace Conditional with Polymorphism" and "Introduce Null Object." The basic idea is to simplify the conditional expressions so that one can understand the logic easily and clearly.

Although this thesis does not deal with any of them, there are following refactorings in this category.

- Decompose Conditional
- Consolidate Conditional Expression

- Consolidate Duplicate Conditional Fragments
- Remove Control Flag
- Replace Nested Conditional with Guard Clauses
- Replace Conditional with Polymorphism
- Introduce Null Object
- Introduce Assertion

2.3.5 Making Method Calls Simpler

Refactorings belonging in this category basically make interface more straightforward. Actually there are variety of refactoring techniques from a very trivial but reasonable one like "Rename Method" to highly sophisticated and rather difficult one like "Separate Query from Modifier."

There are three refactorings that are dealt in this thesis: "Remove Parameter," "Separate Query from Modifier" and "Encapsulate Downcast."

Remove Parameter

"Remove Parameter" is intended to apply when "a parameter [is] no longer used by the method body" [16, p. 277]. A parameter can also be removed when its value is constant or can be computed from other available information. The refactoring eliminates the parameter from the declaration and all calls. The rationale is that extraneous parameters are confusing and burdensome to users of the code.

Separate Query from Modifier

"Separate Query from Modifier" is intended for "a method that returns a value but also changes the state of an object" [16, p. 279]. The refactoring converts a single routine into two separate routines, one of which returns the query result and the other of which performs the modification. The rationale is to give each routine a single clearly defined purpose, to permit clients to perform just the query or just the modification, and to create side-effect-free procedures whose calls may be freely inserted or removed.

Encapsulate Downcast

"Encapsulate Downcast" is intended for "a method that returns an object that needs to be downcasted by its callers" [16, p. 308]. The refactoring changes the return type and moves the downcast inside the method. The rationale is to reduce the static number of downcasts and to simplify implementation and understanding for clients. It can also permit type checks to be performed statically (at compile-time) rather than dynamically (at run-time), which has the dual benefits of early error detection and of improved performance.

The other refactorings belonging to this category are the followings.

- Rename Method
- Add Parameter
- Parameterize Method
- Replace Parameter with Explicit Methods
- Preserve Whole Object
- Replace Parameter with Method
- Introduce Parameter Object
- Remove Setting Method

- Hide Method
- Replace Constructor with Factory Method
- Replace Error Code with Exception
- Replace Exception with Test

2.3.6 Dealing with Generalization

Refactorings belonging to this category often change the class hierarchy. In that sense, these refactorings are very difficult to apply. These refactorings will, however, greatly improve the non-functional qualities such as maintainability, understandability and portability once appropriately applied.

Although this thesis does not deal with any of them, there are following refactorings in this category.

- Pull Up Field
- Pull Up Method
- Pull Up Constructor Body
- Push Down Method
- Push Down Field
- Extract Subclass
- Extract Superclass
- Extract Interface
- Collapse Hierarchy

- Form Template Method
- Replace Inheritance with Delegation
- Replace Delegation with Inheritance

2.3.7 Big Refactorings

Refactorings belonging to this category are different from those introduced in preceding sections. These are more like refactoring strategies than mere refactoring techniques. As for deeper discussion about this category will be found in [16], which is beyond the scope of this thesis.

- Tease Apart Inheritance
- Convert Procedural Design to Object
- Separate Domain from Presentation
- Extract Hierarchy

Chapter 3

Supporting Refactoring Using Invariants

3.1 Introduction

Our research shows the feasibility of another kind of tool to support engineers in refactoring software: automatically finding candidate refactorings. The recommended manual method of identifying beneficial refactorings is to observe design shortcomings manifested during development and maintenance [17]. Unfortunately, design problems may be overlooked or ignored by a programmer, particularly under deadline pressures and the intellectual demands of implementing correct changes.

Our technology for identifying refactoring candidates uses program invariants: a particular pattern of invariants at a program point indicates the applicability of a specific refactoring. This use of program invariants is complementary to other approaches such as human examination or pattern-matching over the source code.

Furthermore, the invariants of the refactored (modified) program indicate which properties were maintained and which were changed, which helps to check that the refactoring was applied properly.

To broaden the applicability of our approach beyond programs for which engineers have written invariants explicitly, we automatically infer the invariants used to find candidate refactorings. In particular, we use the Daikon tool for dynamically discovering program invariants [12, 13].

In the following we discuss prior work in refactoring (Section 3.2), Daikon's approach to detecting invariants (Section 3.3), our approach to finding refactoring candidates (Sections 3.4 and 3.5), and a case study of its use (Section 3.6). We close with a comparison of dynamic and static refactoring detection (Section 3.7) and a discussion of contributions and future work (Section 3.8).

3.2 Related Work

Refactoring Tools. Refactoring is ideally suited to automation: engineers want to apply refactorings, but applying them manually is error-prone (as are all manual software modifications). This research focuses on identifying candidates for refactoring and, to a lesser degree, on checking that manually applied refactorings preserve meaning. In contrast, nearly all the related work focuses instead on automatically applying refactorings once an engineer has identified a candidate. These two styles of automation dovetail quite naturally.

Opdyke [32, 31] and Griswold [18, 21] defined early tools to apply refactorings and ensure that the meaning of the program was left unchanged by the refactoring.[1]

A number of more recent tools also support refactoring: the Smalltalk Refactoring Browser [33], which automatically performs a set of refactorings taken primarily from

[1]Precisely defining what it means to leave the meaning of the program unchanged is important and challenging. For example, the functional properties may be kept stable, but performance properties may change — indeed, that might be one of the motivating reasons to apply the refactoring. The details of this issue are beyond the scope of this paper; they are addressed by Griswold [18] and Roberts [34].

Opdyke's original work; the IntelliJ Renamer tool (www.intellij.com), which supports renaming of packages, variables, etc. and moving of packages and classes for Java; and the Xref-Speller (www.xref-tech.com/speller/), which extends the Emacs editor to support a set of refactorings for C and for Java.

Roberts [34] discusses analyses to support refactoring, especially those defined by Opdyke. Roberts observes that few of Opdyke's refactorings are applied on their own, so he defines the postconditions that hold after a refactoring is applied. This definition allows precise reasoning of postcondition–precondition dependencies among refactorings, which in turn allows compositions of refactorings to be defined. Furthermore, Roberts discusses dynamic refactoring in which the program, while running, checks for certain properties, applies appropriate refactorings, and then can retract those refactorings if the required conditions are later violated. Although Roberts's effort has similarities to our result — it aids in refactoring by exploiting program predicates obtained by dynamic analysis — it, like the tools mentioned above, focuses on the application of refactorings, while we focus on locating where refactorings might apply.

Moore's Guru tool [30, 29] automates two specific and somewhat more global refactorings. It employs a graph-based inheritance hierarchy inference algorithm that can automatically restructure an object-oriented hierarchy for programs written in Self [36]. Using a similar algorithm, Guru can also automatically extract shared expressions from methods.

Bowdidge's Star Diagram [7, 8] hierarchically classifies references to chosen variables or data structures, and provides a tree-based graphical visualization to highlight redundant patterns of usage, facilitating an appropriate object-oriented redesign. The visualization has been used both as the front-end to an automated refactoring tool and a refactoring planner [20]. The tool does not recommend specific refactorings, and the tool user must identify the variables or data structures that are candidates for refactoring.

Finding Duplication in Software. Other related work identifies potential duplication in software systems. Baker [2], for example, locates instances of duplication or near-duplication in a software system by checking for sections of code that are textually identical except for a systematic substitution of one set of variable names and constants for another. Further processing locates longer sections of code that are the same except for other small modifications. Experimental results showed the approach to be effective and fast. There are at least two distinctions between Baker's approach and ours. First, Baker's approach identifies the similarities but does not suggest specific refactorings. Second, it identifies only refactoring candidates involving redundancy, but many refactorings apply to code patterns that appear only once or to code patterns that differ.

Kontogiannis et. al. [28] describe three analysis techniques — source code metrics, dynamic programming, and statistical matching — for finding patterns in code. Experimental results have been promising on moderately-sized systems such as several Unix shells. These techniques — and similar ones such as plan recognizers — are aimed at supporting general program understanding, reverse engineering, and architectural (and design) recovery activities. Although our work has a high-level relationship to efforts like these, our work is distinct due to our use of invariants and our focus on refactoring.

3.3 Invariant Discovery

Dynamic invariant detection [13] discovers likely invariants from program executions by instrumenting the target program to trace the variables of interest, running the instrumented program over a test suite, and inferring invariants over the instrumented values (Figure 3.1). The inference step tests a set of possible invariants against the

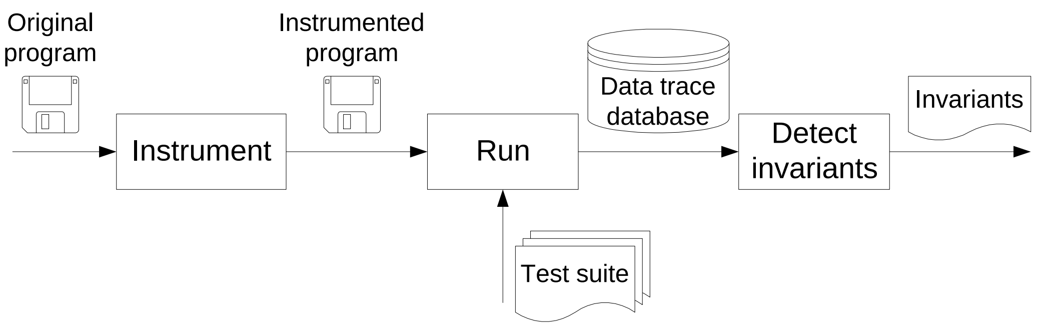

Figure 3.1: An overview of dynamic invariant inference as implemented by the Daikon tool

---

values captured from the instrumented variables; those invariants that are tested to a sufficient degree without falsification are reported to the programmer. As with other dynamic approaches such as testing and profiling, the accuracy of the inferred invariants depends in part on the quality and completeness of the test cases. The Daikon invariant detector is language independent, currently supporting instrumenters for C, Java, and Lisp.

Daikon detects invariants at specific program points such as loop heads and procedure entries and exits; each program point is treated independently. The invariant detector is provided with a variable trace that contains, for each execution of a program point, the values of all variables in scope at that point. Each of a set of possible invariants is tested against various combinations of one, two, or three traced variables.

For variables x, y, and z, and computed constants a, b, and c, some examples are: equality with a constant ($\mathsf{x} = \mathsf{a}$) or a small set of constants ($\mathsf{x} \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$), lying in a range ($\mathsf{a} \leq \mathsf{x} \leq \mathsf{b}$), non-zero, modulus ($\mathsf{x} \equiv \mathsf{a} \pmod{\mathsf{b}}$), linear relationships like $\mathsf{z} = \mathsf{ax} + \mathsf{by} + \mathsf{c}$, ordering ($\mathsf{x} \leq \mathsf{y}$), a range of functions ($\mathsf{x} = \mathsf{fn}(\mathsf{y})$), and invariant combinations ($\mathsf{x} + \mathsf{y} \equiv \mathsf{a} \pmod{\mathsf{b}}$). Also sought are invariants over a sequence variable such as minimum and maximum sequence values, lexicographical ordering, element ordering, invariants holding for all elements in the sequence, or membership ($\mathsf{x} \in \mathsf{y}$). Given two sequences, some example invariants are elementwise linear relationship, lexicographic comparison, and subsequence relationship.

In addition to local invariants such as $\mathsf{node} = \mathsf{node.child.parent}$ (for all nodes), Daikon detects global invariants over pointer-directed data structures, such as $\mathsf{mytree\ is\ sorted\ by} \leq$. Finally, Daikon can detect conditional invariants that are not universally true, such as $\mathsf{if\ p} \neq \mathsf{NULL\ then} *\mathsf{p} > \mathsf{x}$ and $\mathsf{p.value} > \mathsf{limit\ or\ p.left} \in \mathsf{mytree}$. Pointer-based invariants are obtained by linearizing graph-like data structures. Conditional invariants result from splitting data into parts based on the condition and comparing the resulting invariants; if the invariants in the two halves differ, they are composed into a conditional invariant [15].

For each variable or tuple of variables, each potential invariant is tested. Each potential unary invariant is checked for all variables, each potential binary invariant is checked over all pairs of variables, and so forth. A potential invariant is checked by examining each sample (i.e., tuple of values for the variables being tested) in turn. As soon as a sample not satisfying the invariant is encountered, that invariant is known not to hold and is not checked for any subsequent samples. Because false invariants tend to be falsified quickly, the cost of computing invariants tends to be proportional to the number of invariants discovered. All the invariants are inexpensive to test and do not require full-fledged theorem-proving.

To enable reporting of invariants regarding components, properties of aggregates, and other values not stored in program variables, Daikon represents such entities as additional derived variables available for inference. For instance, if array `a` and integer `lasti` are both in scope, then properties over `a[lasti]` may be of interest, even though it is not a variable and may not even appear in the program text. Derived variables are treated just like other variables by the invariant detector, permitting it to infer invariants that are not hardcoded into its list. For instance, if `size(A)` is derived from sequence A, then the system can report the invariant $i < \mathsf{size}(A)$ without hardcoding a less-than comparison check for the case of a scalar and the length of a sequence. For performance reasons, derived variables are introduced only when known to be sensible. For instance, for sequence A, the derived variable `size(A)` is introduced and invariants are computed over it before `A[i]` is introduced, to ensure that `i` is in the range of A.

An invariant is reported only if there is adequate evidence of its plausibility. In particular, if there are an inadequate number of samples of a particular variable, patterns observed over it may be mere coincidence. Consequently, for each detected invariant, Daikon computes the probability that such a property would appear by chance in a random input. The property is reported only if its probability is smaller than a user-defined confidence parameter [14].

The Daikon invariant detector is available for download from `http://pag.csail.mit.edu/daikon`.

3.4 Finding Refactoring Candidates

The goal of this work is to aid the engineer in finding some of the nearly two dozen "bad smells in code" [16, Chap. 3] that motivate refactorings. Human judgment is still required to determine whether a candidate refactoring should be applied: the engineer would apply the refactoring — either manually or using a tool such as those discussed in Section 3.2 — if it was judged to be of value.

Identifying refactoring candidates generally requires a semantic analysis of a program. (In some cases, such as "Large Class" [16, p. 78], a weaker analysis may identify some candidates.) Our approach is to use program invariants to automatically identify candidate refactorings. A particular pattern of invariants identifies a candidate refactoring and where to apply it.

We selected candidate refactorings from among those in Fowler et al. [16] and manually determined the invariants that indicate their applicability.

As an example, the "Remove Parameter" refactoring can be applied if a parameter is not needed. This can arise, for example, when a parameter can be computed from other parameters: a method is a candidate if an invariant over the parameters indicates that one parameter is a function of the others. As one example, this approach allowed us to find a situation in a substantial program in which a potentially rectangular icon was always square: the `height` and `width` parameters were always equal. Automatic detection of this property allowed the engineer to then decide whether or not to apply the refactoring, depending on whether there was still a desire to keep the original flexibility. (Even if the engineer chose not to apply the refactoring, the relationship between the parameters could be retained, perhaps as a comment or annotation, to provide additional documentation about the program; this annotation could be mechanically checked from time to time to identify if and when the potential flexibility was being utilized.)

There are three major steps to perform refactoring.

1. Identifying possible refactoring
2. Applying refactoring
3. Verifying refactoring

As for 2, several useful preceding researches exist [18][21][33][30][36][29]. They often include 3 as a part. For instance, Refactoring Browser [34]

1. checks the possibility of a certain refactoring such as "Inline Temporary," "Remove Parameter from Method," and so on,
2. applies the refactoring to the code, and
3. verifies the refactored code from the specification viewpoint.

Our approach works regardless of how the invariants are created. If the invariants are explicit in the code, then we can analyze those invariants to determine candidate refactorings. In the common case where there are few or no explicit invariants, we dynamically detect program invariants, as described in Section 3.3. (Section 3.7 compares dynamic and static analysis for identifying candidate refactorings.)

3.5 Candidate Refactorings Discoverable from Invariants

This section describes the refactorings (largely from Fowler et al. [16]) that our tool detects, and specifies the invariants that indicate their applicability. Section 3.6 presents a case study over Nebulous [22], including quantitative and qualitative results. We use some examples from this case study to clarify the refactorings discussed here.

3.5.1 Remove Parameter

"Remove Parameter" is intended to apply when "a parameter [is] no longer used by the method body" [16, p. 277]. A parameter can also be removed when its value is constant or can be computed from other available information. The refactoring eliminates the parameter from the declaration and all calls. The rationale is that extraneous parameters are confusing and burdensome to users of the code.

"Remove Parameter" is applicable when either of the following preconditions (invariants at a procedure entry) holds:

- $\mathsf{p} = \mathit{constant}$
- $\mathsf{p} = \mathsf{f}(\mathsf{a}, \mathsf{b}, \ldots)$

where `p` is a parameter, `f` is a computable function, and `a`, `b`, ... are either parameters or other variables in scope at the procedure entry.

In Nebulous one parameter to the `Aspect` constructor was constant:

$$\mathsf{isAutomaticAspect} = \mathsf{true}\ .$$

In another case of a constant parameter, method `SetFirstItemFlag` turned out to have only a single call, and that call passed a literal as the corresponding argument.

The example mentioned in the previous section — when the width and height parameters for a rectangular icon were always equal — illustrates the power of using invariants to find this refactoring. In general, determining equality of two parameters requires nontrivial program analysis.

we found several examples such as the following.

We found the following invariant related with the parameter `flag` of the method `SetFirstItemFlag`.

$$\mathsf{flag} = \mathsf{true}$$

This invariant implies that `flag` is useless parameter in this case and can be eliminated from the parameter list.

```
public void SetFirstItemFlag( boolean flag )
{
  mFirstItem = flag;
}
```

The name implies that the parameter is useless. In fact, there is only one usage of this method.

```
public boolean InsertHistoryItem( int fileID, int lineNum )
{
  ...
  if( -1 == mCurrentIndex )
    item.SetFirstItemFlag( true );
  ...
}
```

Therefore the parameter `flag` can be removed.

3.5.2 Eliminate Return Value

"Eliminate Return Value" is intended for methods that return a trivial value or a value that callers ignore; a value is trivial if it is constant or is computable from other available values. Although this refactoring is not mentioned by Fowler [16], its rationale and mechanics are similar to those of Remove Parameter (Section 3.5.1).

"Eliminate Return Value" is applicable if either of the following postconditions (invariants at a procedure exit) holds:

- $\mathsf{return} = \mathsf{\textit{constant}}$
- $\mathsf{return} = \mathsf{f}(\mathsf{a}, \mathsf{b}, \ldots)$

where `return` stands for the procedure result, `f` is a computable function, and `a`, `b`, ... are in scope at the call site.

In Nebulous, method `makeObjectObey` in class `CollisionCountCommand` had the postcondition

$$\mathsf{return} = \mathsf{true}$$

and in fact, this routine can never return false.

In the Nebulous source code, we found the following example.

```
public boolean makeObjectObey( Object inObject )
```

```
{
  if( inObject instanceof nFlatLandPanel )
  {
    nFlatLandPanel panel =
      ( ( nFlatLandPanel )inObject );
    mCollisionCount += panel.GetNumCollisions();
  }

  return true;
}
```

This method does return a value that is obviously useless because it is always "true."

3.5.3 Separate Query from Modifier

"Separate Query from Modifier" is intended for "a method that returns a value but also changes the state of an object" [16, p. 279]. The refactoring converts a single routine into two separate routines, one of which returns the query result and the other of which performs the modification. The rationale is to give each routine a single clearly defined purpose, to permit clients to perform just the query or just the modification, and to create side-effect-free procedures whose calls may be freely inserted or removed. Another common problem, at least in some of the Nebulous code, is that the procedure name or documentation may not make the side effects clear.

"Separate Query from Modifier" is applicable when two conditions hold at the procedure exit:

- the postconditions do not contain $\mathsf{return} = \mathsf{\textit{constant}}$, even though the procedure returns a value, and

- for some variable a in scope at procedure entry (for instance, a formal parameter), the postconditions imply $\mathsf{a} \neq \mathsf{orig}(\mathsf{a})$

If the return value is constant, the "Eliminate Return Value" refactoring (Section 3.5.2) will be recommended. Postconditions can imply $\mathsf{a} = \mathsf{orig}(\mathsf{a})$ by, for example, containing $\mathsf{a} = \mathit{func}(\mathsf{a})$, where *func* is not the identity function.

Nebulous' `CursorHistory.GetNextItem` method, which returns an object of `CursorHistoryItem`, includes the following postcondition:

$$\mathsf{this.mCurrentIndex} = \mathsf{orig}(\mathsf{this.mCurrentIndex}) + 1\ .$$

This method returns an item in the list and also increments an index into the list.

We found the following invariants related with the method in question `GetNextItem()`.

```
CursorHistory.GetNextItem()LCursorHistoryItem;:::EXIT85
this.mCurrentIndex > orig(this.mCurrentIndex)
```

The first line shows that `GetNextItem()` returns non-trivial value, or a `CursorHistoryItem` class value to its callers. The following two lines show that the inner status of this class, or the member `mCurrentIndex` is modified.

In Nebulous, we found a couple of examples, including the following:

```
public CursorHistoryItem GetNextItem()
{
  CursorHistoryItem item = null;

  if( mCurrentIndex < mHistoryItems.size() - 1 )
  {
    mCurrentIndex++;
    item = ( CursorHistoryItem )
      mHistoryItems.elementAt( mCurrentIndex );
  }

  return item;
}
```

This method returns an item in the list and increments the index at the same time. There are two major problems with this implementation. First, the name of the method does not necessarily imply it does, suggesting that it returns the next item in the list and nothing else. In fact, the inner status of the list is changed by invoking this method. Second there might be redundancy when we need a method that performs either getting next item in the list or incrementing the index.

Rather it is preferable that we provide those two method, or "query part" and "modify part" separately.

3.5.4 Encapsulate Downcast

"Encapsulate Downcast" is intended for "a method that returns an object that needs to be downcasted by its callers" [16, p. 308]. The refactoring changes the return type and moves the downcast inside the method. The rationale is to reduce the static number of downcasts and to simplify implementation and understanding for clients. It can also permit type checks to be performed statically (at compile-time) rather than dynamically (at run-time), which has the dual benefits of early error detection and of improved performance.

"Encapsulate Downcast" is applicable when the following postcondition holds:

- LUB(return.class) $\neq$ *declaredtype(return)*

where LUB is the least-upper-bound operator and `declaredtype(return)` is the declared return type of the procedure. Our current implementation approximates this test with the conjunction of the following two conditions:

1. return.class = *constant*, and
2. return.class ≠ *declaredtype(return)*

One example appears in method `ShowAspect` of class `AspectTraverseComboBox`:

elements of this.comboBoxItems have class AspectTraverseListItem

Although `this.comboBoxItems` is declared as a `Vector` (containing `Objects`), its contents are always `AspectTraverseListItem` objects. These elements can be encapsulated in a more specific container, making the intention clearer.

In the Nebulous source code, we found[[**statically?**]] the following in the body of `ShowAspect`:

```
public void ShowAspect( String pat ) {
  HiddenItem h =
     ( HiddenItem )hiddenItems.get( pat );
  ...
}
```

`hiddenItems` is a `Vector` object that holds `HiddenItem` class objects. It seemed that `hiddenItems` holds `HiddenItem` class object only. Nevertheless `hiddenItems` is declared as a general `Vector` object and therefore `hiddenItems.get()` is required to be downcast by its caller.

There were a couple of other program points that have the same structure seen here.

3.5.5 Replace Temp with Query

"Replace Temp with Query" is intended for "a temporary variable that holds the value of an expression" [16, p. 120]. The refactoring extracts the expression into a method and replaces uses of the temporary by method calls. The rationale is that the expression may be used in multiple places and that eliminating temporary variables can enable other refactorings. In essence, this is the user-level inverse of a compiler's application of the common subexpression elimination optimization.

"Replace Temp with Query" is applicable to a temporary variable if neither the temporary variable nor the value of the expression that initialized it is changed during the temporary variable's lifespan. This is guaranteed by the conjunction of the following two postconditions:

- temp = orig(temp), and
- a = orig(a), b = orig(b), ... for all variables `a`, `b`, ... in the (side-effect-free) initializer for `temp`

Section 3.6 does not report any results for "Replace Temp with Query", because Daikon does not currently report invariants over the initial values of temporary variables. Extending Daikon to do so is relatively straightforward, but time-consuming. (Another problem is the need to check values (of `a`, `b`, ... above) at each use of the temporary; checking only at the procedure exit would not preclude one of those values being changed, then changed back. A static analysis or programmer examination could also suffice for this check.)

As a proof of concept of detecting "Replace Temp with Query", we introduced wrapper functions into Nebulous that make temporary variables into parameters, thus making them visible to Daikon. Using the Daikon output for the modified program, our refactoring pattern-matcher was able to detect candidates for "Replace Temp with Query". Consequently, we are confident that this refactoring can be detected automatically when we improve Daikon as described above. As an example, method `getIndex` contained invariant

numAspects = orig(numAspects) = size(this.aspects)

indicating that the computation of `numAspects` could be abstracted or eliminated.

In the Nebulous source code, we found several examples including the following.

```
public Aspect itineraryLine( String pat, String path,
                                int lineNumber, boolean adding ) {
  int index = getIndex( pat );
  Aspect aspect = null;
  if( index >= 0 ) {
    aspect = ( Aspect )aspects.elementAt( index );
    ...
  }
...
}
```

It may be, however, getting more difficult to realize that which value `index` holds if the method implementation is so long that we cannot see the definition and the usage in one screen.

An important precondition to applying this refactoring is to confirm that neither `pat` nor the return value of `getIndex` itself is changed throughout the lifespan of the temporary variable `index`.

This refactoring requires invariant information related with a local variable which current Daikon cannot handle. To demonstrate the availability of dynamic invariants, however, we introduced a dummy wrapping method which enables Daikon to handle such a local variable in question.

For instance, we made the following change to a method in question.

```
  private int getIndex(String aspectPattern) {
    int numAspects = aspects.size();

    return getIndexReal(aspectPattern, numAspects);
  }
  //  private int getIndex( String aspectPattern )
  private int getIndexReal(String aspectPattern, int numAspects)
  {
    int indexFoundAt = -1;

    //    numAspects = aspects.size();
    ...
  }
}
```

This modification allows Daikon to detect variable `numAspects` related invariants. As a result, we found the following Daikon output as a dynamic invariant for `getIndexReal()` method.

numAspects = orig(numAspects) = size(this.aspects)

This implies that the temporary variable `numAspects` in the original `getIndex()` method is never changed during the method scope. Therefore every appearance of `numAspects` in the original `getIndex()` method can be replaced with `aspect.size()` query method to enhance the readability.

3.6 Analysis of Nebulous

To demonstrate the feasibility of our invariant pattern matching method and to gather informal empirical data about its effectiveness, we applied our technique to Nebulous, a component of Aspect Browser [22]. Nebulous is a software tool that employs simple pattern matching and the geographic map metaphor both to visualize how the code of a program feature or property crosscuts the file hierarchy of the program, and to manage changes to that code. Nebulous is written in Java, and consists of 78 files and a similar number of classes, amounting to about 7000 lines of non-comment, non-blank source code. Over its three year history, it has had three different primary developers under the guidance of one of the co-authors.

We applied our approach as follows:

1. We wrote Perl scripts to identify the patterns of invariants that indicate that particular refactorings may apply (Section 3.5).
2. We used Daikon to extract invariants from a typical Nebulous execution. The test runs of Nebulous exercised a variety of features over two inputs — a student assignment from an MIT class and the source code of Nebulous itself.
3. We ran the Perl scripts over the extracted invariants to identify candidate refactorings from among those described in Section 3.5.
4. The current programmer on the Nebulous project evaluated the usefulness of the recommendations. This evaluation occurred in the presence of one of the co-authors so that qualitative issues could be observed.

The Nebulous programmer classified the recommendations into: *yes*, the recommendation is good; *no*, the recommendation is not good at all; or *maybe*, the refactoring might be a good idea, or another refactoring might be better.

Name	yes	maybe	no	total
Remove Parameter	6	4	5	15
Eliminate Return Value	1	2	4	7
Separate Query from Modifier	0	2	0	2
Encapsulate Downcast	1	1	0	2
Total	8	9	9	26

Remove Parameter Most of the yes's in this category are due to the same literal or object being passed in from all call sites. For a single object being passed in, the restructuring is tricky, since the object's data must still reach the method. Since the object is a singleton, the incoming object's class could be refactored to make it a static class, thus making the data readily accessible via static methods. Most of the no's and maybe's in "Remove Parameter" were detected as candidates because in each instance a flag of the same value was passed in on every call, and this flag controls a case statement that is driving a method dispatch. Thus, although the programmer deemed it incorrect or inappropriate to perform "Remove Parameter," he did decide that "Replace Parameter with Explicit Methods" [16, p. 285] would be appropriate, which would push the switching logic outside the method using the flag. Extending the tool's pattern matching to recognize flags — by detecting the passing of a limited range of values to a method — could yield the appropriate recommendation.

Eliminate Return Value The yes here is for a function that always returns true. The four no's are due to the fact that the usage scenario failed to exercise a couple of Nebulous' more obscure features. (This weakness is due to our use of a dynamic method for discovering invariants: the reported invariants were false and would be eliminated through the use of a richer test suite or a sufficiently powerful static technique.) The two maybe's are functionally correct, but their value is dubious. For example, the `createAspect` method of class `AspectBrowser` need not return its value, but the programmer judged it convenient for additional processing after the aspect was created.

Separate Query from Modification The two maybe's for this refactoring are a matter of programming style. The programmer likes a style of iterator (for example) that uses a modify-and-return approach, which inherently combines the query and the modification. It should be straightforward to customize the invariant pattern matcher to account for programmer preferences, for instance disabling patterns that make recommendations that contradict the programmer's preferred style.

Encapsulate Downcast Both recommendations made by the tool are correct. However, in one case ten casts on a vector appear in the code, in the other just two appear. In the latter case the programmer marked this as a maybe since the amount of casting was limited, thus mitigating the benefits of creating a new class to encapsulate the casting.

Overall, the programmer felt that the use of the tool was quite valuable. The tool's recommendations, although not large in number, revealed fundamental architectural features — the programmer would say flaws — of Nebulous. In particular, although the tool did not detect every use of flags in the system to control method dispatch, the programmer used his knowledge of the system to extrapolate from these few cases to the architectural generalization. Also, although a number of the recommended refactorings were not of interest to the programmer, he quickly picked out the gems, wasting little time on the uninteresting recommendations. Moreover, even the no's provided insight about Nebulous, in particular revealing the excessive use of flags.

Several recommended refactorings, although correct, possessed subtleties that would complicate their application, perhaps enough to discourage their application. One example is "Remove Parameter," which in some cases would necessitate converting a singleton object into a static class. In another case, the recommendation, although technically not meaning-preserving, convinced the programmer that the exceptional, falsifying case should be eliminated to simplify the program. Thus, the process was not an exercise in refactoring alone, but also in functional redesign.

Based on these results, the programmer plans to eliminate the prevailing architectural flaw, systematically refactoring the code to largely eliminate the use of flags and to convert key singleton objects into static classes.

Coincidentally, the programmer had recently used a simple clone-detection tool based on text-based pattern matching [19] to ferret out copy-pasted code. The refactoring recommendations derived from Daikon's output are largely orthogonal to the ones found with the clone detector, thus providing real value. This orthogonality is not surprising, since the techniques described here depend upon run-time values not available to the text-based clone detector.

Finally, we observe that some of the maybe's pertain to the refactoring's usefulness in improving the design, rather than its correctness. For "Separate Query from Modification," this was a matter of style. For such cases, a more complete user interface to our tool would permit a programmer to selectively disable classes of recommendations to avoid being bothered by them. In other cases, the usefulness of the refactoring is a matter of degree. For "Encapsulate Downcast," the programmer felt that the number of casts (and how widespread they were) was a determinant of usefulness. This is a static property of the program's design, as it pertains to the way computations are expressed and modularized into classes and methods. Consequently, it seems that the complementary strengths of dynamic and static analysis would best be combined to achieve high accuracy in refactoring recommendations.

Also, in several instances, when we looked at a recommended refactoring, one of the developers of Nebulous said, "Oh, yeah" as a recognition of a problem he'd noticed in passing during other maintenance efforts. It's notable that he had declined to make the changes at the time–some in part because further work would have been required to verify what Daikon confirmed—and had his attention brought back to them by the refactoring recommendations.

We believe that a simple customization to the tool would permit a programmer to control what recommendations are made based on "style" parameters. For example, don't suggest "Inline Temp" when the temp is a loop bound.

Of course, if more cases had been run, we'd see a different set of invariants (fewer false hits, more invariants that didn't make the cut). What's interesting is that invariants from limited usage are valuable, as they convey typical versus general cases, and it leads one to think about whether the exceptional cases could somehow be eliminated.

In fact, recommendations that involve a value being constant or a function of other values are the most useful, as these are the hardest for us to detect. The "Inline Temp" case is a simple one because it's usually all in one method. But when the constant or computed value is dependent of how a number of clients use the service, that's really valuable, because we have to bounce all over the program–and maybe even run it — to verify that indeed the hypothesized limited use is in fact limited. In fact, some of these limited uses were a surprise to us. We found a singleton object (`StatsGenerator`) that should help simplify the code.

We also observed that refactorings will start to cascade once the initial recommendations are made. Making the `StatsGenerator` a singleton will likely eliminate a `case` statement altogether, which in turn may eliminate an entire method. Likewise, there was a recommended refactoring of `KeepStats`, and we think that this functionality is overly complex and can be eliminated by having stats kept all the time. This won't be a simple refactoring, however, as it will involve a small change in behavior that could be tricky.

3.7 Static and Dynamic Approaches

This section briefly compares our technical approach for identifying refactorings to alternative approaches using static rather than dynamic analysis techniques.

Our basic approach is independent of how invariants are found. But our actual tools, and the case study based on those tools, use dynamic techniques — those that run the program and observe its execution. Dynamic analysis is by no means the only reasonable way to detect refactorings; we have suggested it as a complementary technique that can enhance other techniques and enable more refactorings to be performed.

The central alternative to dynamic analysis is, of course, static analysis. (Human analysis is sometimes more powerful than either, allowing deep and insightful reasoning that is beyond hope for automation.)

The results of dynamic analyses inherently depend on the test suite used to produce those results; by contrast, static analyses are in general sound with respect to all possible program executions. Our dynamically-based results need to be verified, either by the engineer or by a (static) analyzer such as a meaning-preserving tool for applying the suggested refactoring (Section 3.2). Deferring checking should be acceptable when most candidates are indeed applicable, as is the case for our tool.

The classic distinction between dynamic and static techniques is that the results of dynamic analyses inherently depend on the test suite used to produce those results while, in contrast, static analyses are in general sound with respect to all possible program executions. This implies that our dynamically-based results need to be verified by the engineer to ensure that they are accurate. An alternative in this domain is to use a meaning-preserving tool for applying the suggested refactoring (Section 3.2), which will fail if the suggested refactoring does not in fact apply. That is, checking the soundness of the results can be delayed to a later tool; this may be an acceptable approach when most reported invariants (and thus, in this work, reported candidates for refactoring) are in fact true. This is a qualitative property that seems to hold in our applications of dynamically extracted invariants to date, including the current tool.

A combination of underlying analyses seems best for automated refactoring. Consider again the question of when the "Remove Parameter" refactoring can be applied. A parameter can be eliminated if it is never used, but also if its value can be determined from other available information. The former can be determined in some cases by a trivial lexical analysis (or by a deeper analysis, if the parameter is passed to other routines that can be determined never to use it), and a dynamic analysis is not appropriate. By contrast, determining the run-time relationship among variables or their run-time values is beyond the capability of most static analyses, but can be easily checked dynamically. In some cases, a combined static and dynamic analysis will be most appropriate; for instance, static analysis can detect a pattern in the program text, and dynamic analysis can verify that a necessary relationship holds over the values, as for "Encapsulate Downcast. In other cases, static and dynamic analysis can give different perspectives on a property; for instance, "Move Field" and "Move Method" [16, pp. 142, 146] depend on how much the field or method uses or is used by other classes. The static and dynamic usage counts may be quite different.

The applicability of many refactorings in the literature depends only on information that can be obtained statically or even lexically. (As just one example, Fowler states that "Remove Parameter" [16, p.277] is applicable if the parameter is never used, but he omits cases where the parameter is constant, is computable from other available values, or is only passed to other procedures that do not use it.) We hypothesize that one reason for this bias is that programmers and authors are accustomed to static analysis for program understanding, and a number of such tools exist. Daikon (for invariant detection) and other dynamic analysis tools make a whole new class of information available to programmers. We expect that this will change the way that they think about their code, and in particular it will expand the set of refactorings that people consider reasonable in the future.

3.8 Conclusion

This research indicates that program invariants can be used to effectively discover refactoring candidates. Our results, although preliminary, are compelling: we can extract invariants from a realistic codebase and use them to identify a set of refactorings that the author of the codebase had not previously identified or considered. There are three especially attractive properties of our approach: first, the computation of the invariants and the identification of refactoring candidates is automatic; second, the recommended refactoring is justified in terms of run-time properties of the code (in the form of Daikon invariants) that must hold for the refactoring to be correct; and third, our tool reports the candidate refactoring in terms of the actual source code, which tends to reduce the hesitance of engineers who might wish to use the tool. The results are sufficiently promising to justify further investment in this approach.

A number of questions remain, which we expect to drive our future work in this area. First, in our initial case study about half of the candidate refactorings showed real promise. Ideally, we should reduce the other half of less-promising candidates to relieve the engineer from having to consider them as possible refactorings. Selection of better test suites and complementing our dynamic analysis with static analysis could go a long way towards achieving this goal. Second, we need more empirical assessment of the utility of our approach: our data point may not be representative of programs in general, and determining if our approach works better or worse in other situations is crucial. Third, we need to determine how well our approach dovetails with other approaches for identifying refactoring candidates; the observation that our approach found distinct candidates from some clone-based approaches is promising in this regard. Fourth, we need to take steps towards integrating our tools with those for applying refactorings (Section 3.2), with the intent of giving engineers a powerful suite of refactoring tools.

With respect to the use of dynamically discovered invariants, our work provides another data point for their potential effectiveness for realistic software engineering tasks. This is the first use of invariants in which a tool consumes the invariants as opposed to an engineer. A minor but interesting consequence of this is that our work on improving the relevance of extracted invariants [14] is of less importance, since our refactoring tool would not be distracted nor overwhelmed by additional, less relevant invariants. This kind of observation can help drive the work on dynamically discovering program invariants in new directions, too.

Using program invariants to discover refactoring candidates shows significant promise as a complementary approach to tools that help to automate refactoring itself. That the reported candidate refactorings for Nebulous suggested some broader changes to the codebase was unexpected; it hints that tools like this might address additional problems that arise during software evolution and maintenance.

Acknowledgements

I would like to thank Professor William G. Griswold for his advice and collaboration for the research introduced in Chapter 3. He often join us in a conference call to discuss the research issues.

I would like to thank Wesley Leong for evaluating our tool's recommendations on Nebulous, and the anonymous referees for comments that improved the presentation. The research was also supported in part by NSF grant CCR-9970985, an IBM Graduate Fellowship, and gifts from Edison Design Group, Microsoft Corporation, and Toshiba Corporation.

Chapter 4

A Quantitative Evaluation of Refactoring Effect

4.1 Introduction

According to [16], we can reap the following benefits from refactoring.

- Improve the design of software
- Make software easier to understand
- Find bugs
- Program faster

These statements seem to be intuitively true, and various qualitative evaluations regarding the effectiveness of program refactoring have been proposed so far[23][27]. Experienced programmers easily acknowledge and realize the effectiveness of program refactoring. Actually they are practicing refactoring as a routine work during the program development.

On the other hand, inexperienced programmers cannot afford to apply program refactoring because they are too busy implementing given requirements as fast as they can, nor do managers properly realize the effect of refactoring. They are so occupied to make their project meet the delivery that they dare not encourage programmers to refactor their programs.

As stated in Chapter 2, there are at least the following three phases to perform appropriate refactorings.

1. Identification of refactoring candidates

2. Application of refactoring

3. Validation of refactoring effect

Supporting the identification phase will encourage people to take program refactoring into consideration as a quality enhancement method. There are a few previous works including our previous result.[24][26]

Supporting the application of refactoring will also help people to practice program refactoring. There are several theoretical and practical approaches for the application of refactoring.[18][21][34]

From the viewpoint of project managers, however, it is imperative to quantitatively evaluate the effect of program refactoring before applying it. Without knowing the effect, managers cannot judge whether they should go for refactoring or not because they have to be cost sensitive.

We propose a quantitative evaluation guideline to measure the maintainability enhancement effect of program refactoring. We focused on the module coupling metrics and the module cohesion metrics to evaluate the refactoring effect. Comparing the metrics before and after the refactoring, we could evaluate the degree of maintainability enhancement.

We begin with a description of the refactoring process and the current status of supporting technologies of the process, and indicate what to be studied in Section 4.2. We begin with a definition of refactoring effect as a foundation of the refactoring validation supporting method development in Section 4.3. Based on the discussion in the previous section, we provide a detailed quantitative evaluation model of refactoring effect from module coupling viewpoint in Section 4.4, followed by Section 4.5, which provides the experimental evaluation of the refactoring effect quantification. Our future work, together with some related work information, is described in Section 4.6.

4.2 Supporting Refactoring Process

As stated in Chapter 2, the program refactoring process consists of three major phases: identification of refactoring candidates, validation of refactoring effect, and application of refactoring(see Fig. 1.1).

We are studying how to support the refactoring process effectively so that program development projects can adopt the process easily and enhance the maintainability of the target software. Although there are several distinguished research results for some activities of the process, it is still far from being the comprehensive support of the refactoring process we stated above.

For instance, there are some fundamental studies on the refactoring application phase including Griswold and Notkin's work[21]. The result introduced earlier (Chapter 3) is also an approach to identify refactoring candidates using program invariants[26]. We are developing a refactoring support tool "*Refactoring Assistant*" which implements those existing approaches and others and supports a part of planning and execution of the refactoring process. *Refactoring Assistant* provides bad–smell detection support based on static analysis of a program. It can detect almost half of bad–smells described in [16]. It also provides automatic application of some basic refactorings such as "Move Method," "Extract Method," and "Inline Method."

As shown in Table 1.1 in Chapter 2, we need some effective approach to support the validation subprocess. Experienced developers and analysts who possess fairly reasonable insight into software maintainability will be able to benefit from the current planning subprocess and execution subprocess support. It is not necessarily easy, however, to share such useful insight among all developers and analysts who are involved in a software development project. Quantitative evaluation of those subjective judgments make the information sharing easier. Moreover validation subprocess support allows software development project managers to enhance their projects' maintainability appropriately. Without such a managerial point of view, maintainability enhancement tends to be ad hoc and might even make it worse.

In this research we focus on how to evaluate refactoring effect to support the plan evaluation and refactoring validation.

4.3 Refactoring Effect Evaluation

4.3.1 Definition of Refactoring Effect

One of the major objectives of refactoring is the enhancement of the maintainability of the software. Therefore it is reasonable to evaluate the refactoring effect in terms of maintainability. We first focus on how to quantify the maintainability of the software. There have been many distinguished previous works on the software maintainability quantification.[3][10][5][37] From the refactoring point of view, however, it is reasonable to focus on the maintainability of each part of a program and not of a whole program. Each refactoring technique has its own policy of how to enhance the maintainability of a certain part of a program. For instance, one obvious objective of "Extract Method" refactoring is to decrease the size of the method in question. Reducing the size, of course, is one way to enhance the maintainability of the method. "Replace Temp with Query" refactoring has a different objective to enhance the maintainability. It aims to separate a certain calculation as an independent method in the enhancement of the cohesion of the method.

In this research, we focus on two major maintainability metrics: coupling and cohesion. There are three reasons for selecting these two metrics. First, these two viewpoints are quite essential during the quality control activity in our in-house software development site. In fact, considerable amount of software problems caused by highly coupled modules and low cohesion modules are found during the test and maintenance phases. Second, a number of refactoring techniques introduced in [16] intend to decrease the module coupling and/or enhance the module cohesion as we mentioned above. Therefore evaluating refactoring effect based on these two metrics is reasonable. Third, although these two metrics are essential in terms of maintainability, it is very difficult to grasp the effect intuitively. Our developers require handy evaluation method to judge whether a certain refactoring should be applied or not in terms of cost-effect trade-off.

In the following, we classify several refactoring methods, which are often required in our in-house software development site, introduced in [16] in terms of two metrics and show the guideline to evaluate refactoring effect.

4.3.2 Refactorings enhancing module couplings

As represented by the early work by Stevens, Myers, and Constantine[35], module couplings is an important metrics for maintainability. Stevens et al. illustrated three major contributing factors of module couplings, which are "Interface complexity," "Type of connection," and "Type of communication."(See Table 4.1)

According to Table 4.1, there are three approaches to increase maintainability using refactorings:

Table 4.1: Contributing factors

	Interface complexity	Type of connection	Type of communication
low	simple, obvious	to module by name	data
Coupling			control
high	complicated, obscure	to internal elements	hybrid

---

1. making interface simple and obvious,
2. reducing direct connection into modules, and
3. simplifying exchanged data.

Making interface simple and obvious

One of the typical complicated/obscure interfacing patterns is exchanging data through a shared variable. It is rather complicated because one should browse at least two different objects to clarify the interface of a certain method/class. It is also obscure because the shared variable does not directly appear in the function definition. In that sense, replacing the shared variable with an obvious parameter would loosen the coupling among methods/classes.

The number of data for interfacing also matters. The more data are exchanged, the tighter is the coupling between modules.

We will show the refactoring effect using experimental results later.

Reducing direct connection into modules

An simple example of a direct connection into modules is that a class c_A has a public data d_A, which is accessed by another class c_B through a method m_B. Fowler[16] also mentions that such a direct connection is harmful because "other objects can change and access data values without the owning object's knowing about it."

"Encapsulate Field" is a typical refactoring technique which resolves such a situation. In that sense, "Encapsulate Field" will reduce direct connection into modules and hence increase maintainability from coupling viewpoint.

Simplifying exchanged data

Stevens et al. explain how to loosen the coupling between modules by simplifying exchanged data[35]. For instance, method m_A passes a flag f_A to method m_B which includes two distinct actions, a_{B1} and a_{B2}. In m_B, f_A decides which of a_{B1} and a_{B2} should be executed. Separating those two distinct actions results in a simpler structure. In other words, refactoring techniques such as "Extract Method," "Move Method," etc., will provide a good chance to loosen the coupling.

4.3.3 Refactorings enhancing module cohesions

Cohesiveness is another important metrics for maintainability beside module coupling[35]. Although there are variations in the definition of module cohesion, we can generally classify the cohesiveness in the following six levels:

1. **Coincidental** totally unrelated elements
2. **Logical** sharing a general category only
3. **Temporal** control flow without order
4. **Communicational** data flow without order

5. **Sequential** data flow with order
6. **Functional** contributing to one and only relevant task

Although this graduation itself is very sophisticated, it is not suitable for refactoring effect evaluation. Hence we first define a quantitative model of module cohesiveness.

Quantitative Cohesion Model

To evaluate refactoring effect in terms of cohesiveness improvement, we prepare the following quantitative cohesion model. First of all, we assume the following notions.

(Program) Element Variables, constants, statements, methods and objects.

Cluster A set of elements/clusters.

According to the above definition, a method could be treated as a cluster of variables, constants and statements. At the same time, a method could be a component of a larger cluster, or a class/instance.

Two program elements have a "relation" if there is a certain dependency between them. For instance, method m_A and method m_B have a relation if m_A calls m_B, or vice versa. And class c_A and class c_B have a relation if a method m_A in c_A refers a variable v_B in c_B.

We define a graph-based model of a cluster: assigning each element to a vertex and each relation to an edge connecting corresponding vertices.(See Fig. 4.1) Each edge is assigned a certain weight, which is defined by a number of appearances of the corresponding relation and a coefficient. We define two different types of coefficients: an intra–cluster coefficient w_i and an inter–cluster coefficient w_e. Preferably, w_i is two or three times larger than w_e. For instance, a relation corresponding an edge e is n-ply and the relation is an intra–cluster one, the weight of e is nw_i. We also define a vertex weight, which is either

- 0 if the vertex is primitive (not a cluster), or
- weight of corresponding sub-cluster, if otherwise.

Based on the model, we define a cluster weight (of a cluster c) $CW(c)$ as a sum of edge weights plus sum of vertex weights.

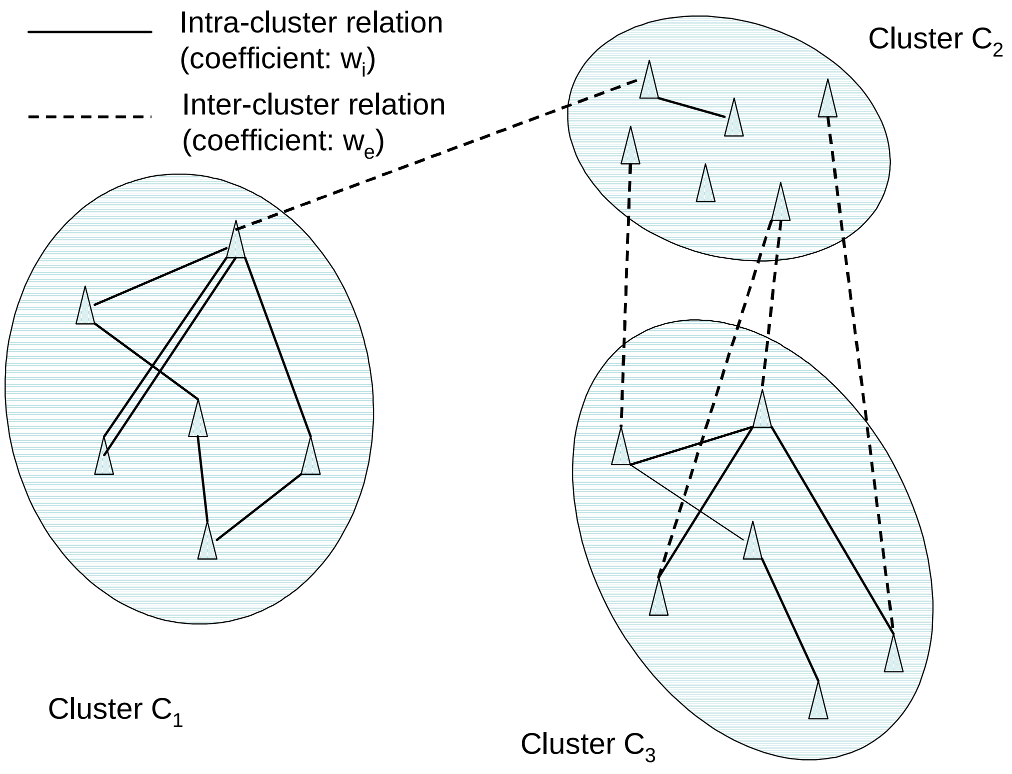

Figure 4.1: Quantitative Cohesion

Now we are ready to define a strength of functional relatedness in a cluster, or cluster cohesion index (of a cluster c) $C_i(c)$ as follows.

$$C_i(c) = \frac{CW(c)}{NV(c)}$$

where $NV(c)$ is a number of vertices in (intra–) or connected to (inter–) cluster c.

Fig. 4.1 includes three clusters, C_1, C_2 and C_3. Assuming $w_i = 2$ and $w_e = 1$, we can evaluate cohesiveness of those three clusters as follows.

C_1 shows the highest cohesion among them due to large weight concentrated on intra–cluster links.

$$CW(C_1) = 15, NV(C_1) = 7 \Rightarrow C_i(C_1) \approx 2.1$$

On the other hand, C_2 shows the lowest cohesion among them as elements are not related much with each other.

$$CW(C_2) = 7, NV(C_2) = 11 \Rightarrow C_i(C_2) \approx 0.6$$

C_3 shows intermediate cohesion, large weight but there are quite a number of inter cluster links.

$$CW(C_3) = 14, NV(C_3) = 9 \Rightarrow C_i(C_3) \approx 1.6$$

4.3.4 Refactoring Effect from Cohesiveness Viewpoint

Based on the previous assumption of quantitative evaluation of module cohesiveness, we examine a number of refactoring techniques which are introduced in Fowler's textbook[16]. We also include some refactoring techniques which are difficult to evaluate the refactoring effect with in terms of previously discussed quantitative model. We believe, however, that even qualitative effect evaluation would help further exploration.

Composing Methods These are good for procedural and temporal modules (control-flow biased) for it concerned mainly with method cohesion.

Extract Method increases method cohesion due to reducing the number of vertices in the method ($NV-$).

Inline Method does not change to method cohesion, but increases module cohesion ($NV+$).

Moving Features Between Objects These are mainly concerned with class cohesion and are effective with functional, sequential and communicational types.

Move Method, Move Field increase class cohesion ($NV-$) and module cohesion ($CW+$ as coefficient w_i replaces coefficient w_e).

Extract Class increases sub-cluster (new class) cohesion and hence module cohesion.

Organizing Data These are not quite effective with procedural and temporal modules for it is a data-centered technique, but would be effective at a lower level of cohesiveness.

Self Encapsulate Field reduces class cohesion ($NV+$) for highly cohesive classes.

Replace Array with Object increases class and module cohesion; an isolated array $\rightarrow$ a cohesive class.

Simplifying Conditional Expressions These are mainly concerned with complexity of conditional expressions, not much relevant to software cohesion.

Making Method Calls Simpler These may be suitable to all categories but not to a great extent for it is about method re-arrangement with respect to class and method itself.

Add Parameter increases method, class cohesion ($CW+$, $NV+$).

Separate Query from Modifier increases sub-cluster (new method) cohesion and hence class cohesion.

Dealing with Generalization It is hard to evaluate cohesion change with these refactorings due to possible dynamic binding.

Pull Up Field/Method/Constructor Body remove code redundancy; often subclass cohesion down ($CW-$, $NV-1$); super-class cohesion up ($CW+$, $NV+1$).

Push Down Field/Method usually subclass cohesion up; super-class cohesion down.

Extract Subclass increases cohesion of a parent class and a new sub-class substantially.

Big Refactorings These affect on system cohesion to a great extent for it is a large scale system reorganization.

Tease Apart Inheritance converts a class hierarchy into two very cohesive small class hierarchies, especially relevant for top 3 categories.

Convert Procedural Design into Objects is suitable with procedural and temporal levels and new module (class) cohesion can be substantially higher.

Extract Hierarchy will lead to a highly cohesive method and class for it replaces a class with many conditional flags into a hierarchy, and each sub-class is responsible to a condition.

Example: Separate Query from Modifier

Fig. 4.2 shows an example of "Separate Query from Modifier" refactoring. Method m_{qm} includes two elements unrelated with each other, or a query e_q and a modifier e_m, and m_{qm} is called from n (> 0) methods. The cohesion index for m_{qm} is:

$$C_i(m_{qm}) = \frac{nw_e}{2+n}.$$

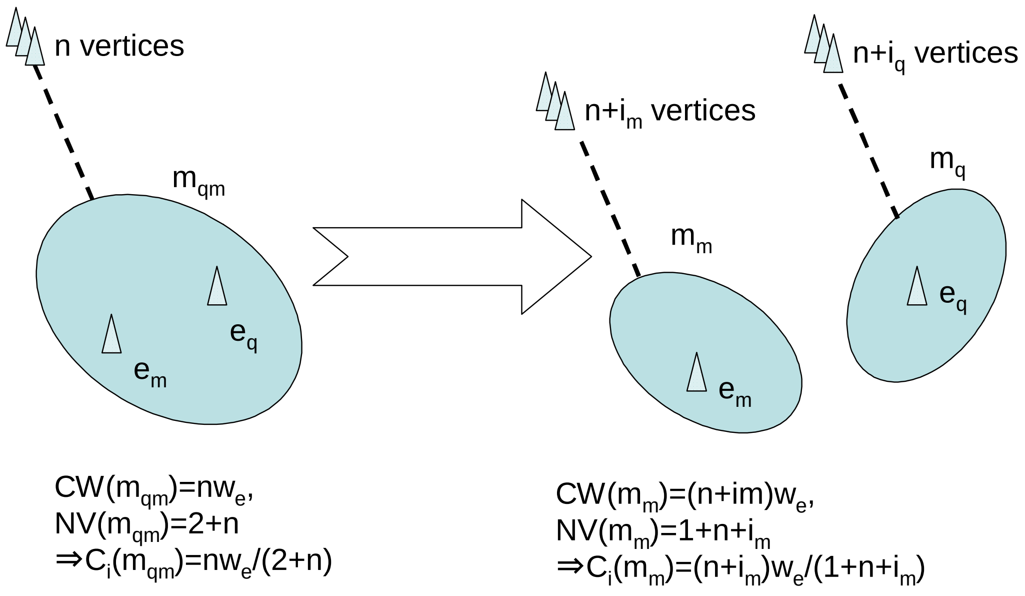

Figure 4.2: Example of "Separate Query from Modifier

After the refactoring, m_{qm} is divided into two methods, m_q and m_m. To keep the semantic unchanged, both m_q and m_m are called from n methods just as m_{qm} was. Moreover, there might be additional calls to both methods. Suppose we have additional i_q (≥ 0) callers and i_m (≥ 0) callers for m_q and m_m, respectively. As for m_m, the cohesion index is:

$$C_i(m_m) = \frac{(n+i_m)w_e}{1+n+i_m}.$$

We can easily get the same result for $C_i(m_q)$.

Assume $C_i(m_m)$ (and $C_i(m_q)$, as well) is larger than $C_i(m_{qm})$, or:

$$\frac{nw_e}{2+n} < \frac{(n+i_m)w_e}{1+n+i_m}.$$

Solving the inequality, we can get the following condition for n.

$$n > -2i_m$$

Since the right-hand side is negative (because $i_m \geq 0$) and $n > 0$, the above inequality is always true. Therefore we can conclude that both $C_i(m_m)$ and $C_i(m_q)$ are always larger than $C_i(m_{qm})$. And the result implies that "Separate Query from Modifier" refactoring actually increases method cohesion.

4.4 Quantitative Coupling Evaluation

In this experiment, we focus on refactorings that enhance the maintainability of a method in terms of coupling. We examine those refactorings that are supposed to reduce coupling among methods such as "Extract Method," "Extract Class," or "Move Method." In the following, we are using Java and a method as examples of a programming language and a unit of a part of a program, respectively.

4.4.1 Definition of Coupling

First we define the coupling between methods. Although various definitions and calculation policies have been previously proposed, we developed our own definition based on Allen et al.'s [1], Briand et al.'s [9], and others. Our definition is based on our original handy metrics which are used in our in-house software development site.

We classified coupling into three major categories.

Return value coupling

When method A uses a return value from method B, we define that A has a *return value coupling* with B. When method A provides a return value to method B, we also define that A has a *return value coupling* with B.

The whole return value coupling of method A can be defined as a total of those related values. Hence we define return value coupling of A, or $C_{rv}(A)$ as follows:

$$C_{rv}(A) = \sum_{m_\rho \in \rho(A)} K_{rv}(m_\rho) + \sum_{m_\sigma \in \sigma(A)} K_{rv}(m_\sigma),$$

where,

$\rho(A)$:	a set of methods whose return value is used by A,
$K_{rv}(m)$:	return value coupling inter–class coefficient, $= 1$ when m is in the same class as A, $= \kappa_{rv} > 1$ when m is in a different class,
$\sigma(A)$:	a set of methods that use A's return value.

$K_{rv}(m)$ has been introduced to reflect that an inter–class coupling could be a greater maintenance obstacle than an intra–class coupling. Hence we apply $\kappa_{rv}(> 1)$ coefficient to inter–class parameter couplings.

Parameter coupling

When method A receives n parameters from method B, we define that A has n *parameter coupling* with B. When method A passes m parameters to method C, we define that A has m *parameter coupling* with C.

The whole parameter coupling of method A can be defined as a total of those related values. Hence we define parameter coupling of A, or $C_{pp}(A)$ as follows:

$$C_{pp}(A) = \sum_{m_\zeta \in \zeta(A)} K_{pp}(m_\zeta) p_{m_\zeta} + \sum_{m \in \xi(A)} K_{pp}(m_\xi) p_{m_\xi},$$

where,

$\zeta(A)$:	set of methods that invoke A,
$\xi(A)$:	set of methods that are invoked by A,
$K_{pp}(m)$:	parameter coupling inter–class coefficient, $= 1$ when m is in the same class as A, $= \kappa_{pp} > 1$ when m is in a different class,
p_m:	# of parameters of m.

$K_{pp}(m)$ has been introduced for the same reason as $K_{rv}(m)$ has been introduced.

Shared variable coupling

When method A uses n class/instance variables in common with another method B, we define A has n *shared variable coupling* with B.

The whole shared variable coupling of method A can be defined as a total of those related values. Hence we define shared variable coupling of A, or $C_{sv}(A)$ as follows:

$$C_{sv}(A) = \sum_{m_\chi \in \chi(A)} K_{sv}(m_\chi) v_{m_\chi},$$

where,

$\chi(A)$:	a set of methods that use class/instance variables in common with A,
$K_{sv}(m)$:	shared variable coupling inter–class coefficient, $= 1$ when m is in the same class as A, $= \kappa_{sv} > 1$ when m is in a different class,
v_m:	number of class/instance variables appearing in both method A and method m.

$K_{sv}(m)$ has been introduced for the same reason as $K_{rv}(m)$ has been introduced.

4.4.2 Combining Three Couplings

Three couplings introduced above are not equally important from the maintainability viewpoint. For instance, when we compare accessing an instance variable of another class and receiving a parameter from a method of another class, the former case might have considerably more severe influence than the latter case.

Therefore we have to apply proper coefficients when combining those three coupling metrics into one that would evaluate the maintainability of a certain method. We prepare three coefficients $K_{T_{rv}}$, $K_{T_{pp}}$, and $K_{T_{sv}}$ for the parameter coupling, the shared variable coupling, and the return value coupling, respectively. Although there is no reasonable clue to deciding the exact value for each of those coefficients, we dare to assume the following inequality relationship among them:

$$0 < K_{T_{rv}} \leq K_{T_{pp}} < K_{T_{sv}} \quad (K_{T_{rv}} + K_{T_{pp}} + K_{T_{sv}} = 1)$$

The whole coupling metrics value for the method A $C_T(A)$ is then calculated in the following formula:

$$C_T(A) = K_{T_{rv}} C_{rv}(A) + K_{T_{pp}} C_{pp}(A) + K_{T_{sv}} C_{sv}(A)$$

4.5 Experiment and Result

4.5.1 Example of Quantification

We will show the example of refactoring effect quantification using a refactoring called "Move Method." "Move Method" is supposed to be applied when "a method is, or will be, using or used by more features of another class than the class on which it is defined."[16]

Fig. 4.3 and 4.4 show program codes before and after the "Move Method," respectively.

These programs are for the bank account management. Class `AccountType` deals with various types of bank account such as a regular account, a premium account and some others. In this case we assume that there are going to be several new account types, each of which has its own rule for calculating the overdraft charge. Therefore we plan to move `overdraftCharge()` method to the `AccountType` class from `Account` class.

We can expect that the coupling of the method `overdraftCharge()` decreases after the refactoring. Let n be a number of account types dealt with `AccountType` class.

```
class Account...
  double overdraftCharge() {
    if (_type.isPremium()) {
      double result = 10;
        if (_daysOverdrawn > 7)
          result += (_daysOverdrawn - 7) * 0.85;
        return result;
    }
  }

  double bankCharge() {
    double result = 4.5;
    if (_daysOverdrawn > 0)
      result += overdraftCharge();
    return result;
  }
  private AccountType _type;
  private int _daysOverdrawn;
```

Figure 4.3: Before "Move Method"

Before the refactoring, `overdraftCharge()` provides its return value to `bankCharge()` and uses the return values of n account type decision methods such as `isPremium()` in `AccountType` class. In other words, the coupling of `overdraftCharge()` consists of one intra–class return value coupling and n inter–class return value couplings. Therefore the whole coupling of `overdraftCharge()` is $K_{T_{rv}}(1 + n\kappa_{rv})$.

After the refactoring, on the other hand, `overdraftCharge()` provides its return value to `Account.bankCharge()`, receives a parameter from `Account.bankCharge()`, and uses the return values of n account type decision methods such as `isPremium()`. In other words, the coupling of `overdraftCharge()` consists of one inter–class return value coupling, n intra–class return value couplings, and one intra–class parameter

```
class AccountType...
  double overdraftCharge(int daysOverdrawn) {
    if (isPremium()) {
      double result = 10;
      if (daysOverdrawn > 7)
        result += (daysOverdrawn - 7) * 0.85;
      return result;
    }
    else return daysOverdrawn * 1.75;
  }

class Account...
  double bankCharge() {
    double result = 4.5;
    if (_daysOverdrawn > 0)
      result +=
        _type.overdraftCharge(_daysOverdrawn);
    return result;
  }
  private AccountType _type;
  private int _daysOverdrawn;
```

Figure 4.4: After "Move Method"

---

coupling. Therefore the whole coupling of `overdraftCharge()` is $K_{T_{rv}}(\kappa_{rv}+n)+K_{T_{pp}}$.

The refactoring effect is then calculated as follows:

$$K_{T_{rv}}(1+n\kappa_{rv})-\{K_{T_{rv}}(\kappa_{rv}+n)+K_{T_{pp}}\}$$

$$=K_{T_{rv}}(\kappa_{rv}-1)(n-1)-K_{T_{pp}},$$

and this value is positive under the following condition:

$$(\kappa_{rv} - 1)(n - 1) > \frac{K_{T_{pp}}}{K_{T_{rv}}}(> 1).$$

Suppose $\kappa_{rv} > 1.5$, which means that it is at least 1.5 times more costly to maintain inter–class coupling than intra–class coupling. Considering that inter–class coupling involves at least two different classes while intra–class coupling involves only one class, this assumption cannot be so far from the reality. If $\kappa_{rv} > 1.5$, then the refactoring effect is positive when $n \geq 3$, which means that this refactoring is effective if there are three or more account types involved and that is what we have assumed in this example.

4.5.2 Experiment

To evaluate our theory, we implemented the following experiment. We chose one software project written in C++ which had been maintained for almost five years. We realized that the maintainability of the software has been much deteriorated. We applied *Refactoring Assistant* to the software and found many bad–smells in terms of coupling metrics. We also interviewed the developer of the program to recognize the most serious problems in terms of maintainability. Using such information, we have singled out five problems. The developer then performed refactoring to each of those five problems. We did not provide the developer any precise information about the coupling metrics values measured by *Refactoring Assistant* beforehand.

Case 1: Extract Method(1)

In this case, the original method indicated very high coupling with many other methods including the ones in other classes. The developer admitted that the method should be divided into at least two parts: one for manipulating instance variables to update the object's state and the other for collecting required information from other objects. Therefore the developer applied Extract Method refactoring.

Case 2: Extract Method(2)

This case also showed rather high inter–class parameter coupling and intra–class shared variable coupling. However, we could find little effective solution for this problem. We applied the solution anyway only to find that the refactored code was no more maintainable than the original one.

Case 3: Extract Class(1)

In this case, the original method indicated very high shared variable coupling with many other methods in the same class. The method dealt with four instance variables all of which performed very similar roles to one another. Actually those four instance variables all together prescribe a certain state of the object. Hence the developer decided to introduce a new class to manage those four instance variables together with appropriate methods to manipulate them.

Case 4: Extract Method(3)

The method in question is a check method which traverses two instance variables. Those two instance variables are very similar to each other and so are the ways to traverse them. The developer therefore extracted a traverse method. The extraction separated the original method into two parts: one for the query part and the other for the traversing.

Case 5: Extract Class(2)

The original method accessed twelve instance variables. Those instance variables are also accessed by a number of other methods not only in the same class but in many different classes. The developer had realized that some encapsulation should be applied to those instance variables. We introduced a new class that deals with those instance variables and provides methods to access/update those values. As a result, the extracted class became responsible for the shared variable coupling and the method in question became almost free from the shared variable coupling.

4.5.3 Consideration

We collected subjective data from the developer regarding coefficients discussed in the previous section. As a result, we chose these coefficient values for the target system:

$$\kappa_{rv} = 1.5,\ \kappa_{pp} = 2.0,\ \kappa_{sv} = 3.0$$

$$K_{T_{rv}} = 0.2,\ K_{T_{pp}} = 0.2,\ K_{T_{sv}} = 0.6.$$

Table 4.2: Refactoring Experiment Result

case	sbj.	before	after(effect)
1	B	10.4	2.8(7.6)
2	C	12.1	9.0(3.1)
3	B	25.2	9.0(16.2)
4	B	26.4	1.7(24.7)
5	A	126.0	26.0(100.0)

Table 4.2 summarizes the result of our experiment. The column "sbj." shows the subjective evaluations of the refactoring effectiveness by the developer which are:

A considerably effective

B somehow effective

C not necessarily effective.

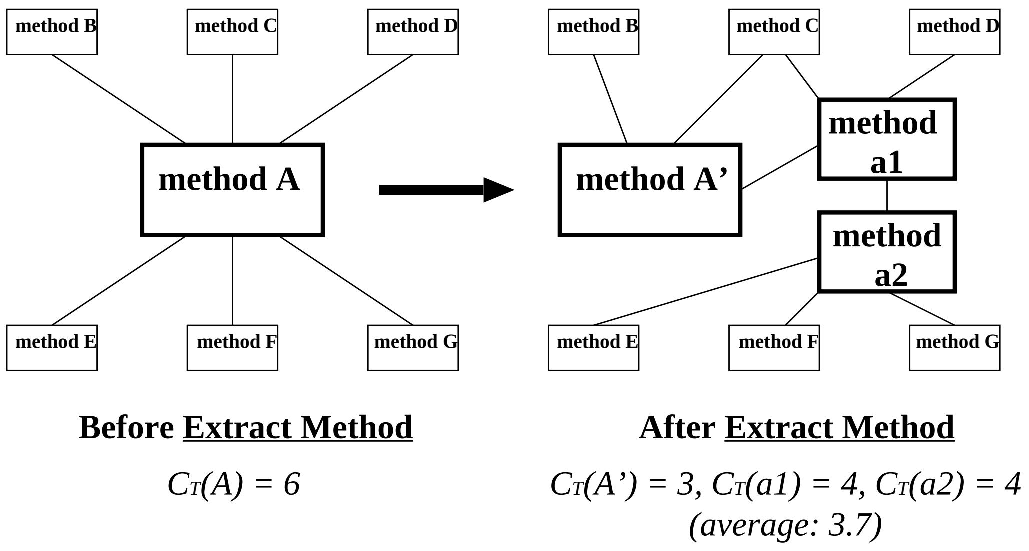

Figure 4.5: Example of Good Refactoring

The column "before" shows the coupling metrics value of the target method before the refactoring. The column "after(effect)" shows the coupling metrics value of the target method after the refactoring and the difference from before–the–refactoring value, or the effect of the refactoring. It is important to evaluate the refactoring effect by these values in the parentheses. However we have to consider other factors as well. Fig. 4.5 and 4.6 describe the situation. A is the original method, and "Extract Method" divides A into three methods: A' which corresponds to A, $a1$ and $a2$ which are derived from A by the refactoring. Fig. 4.5 is an effective refactoring example. After the refactoring, the coupling of A is well balanced among three methods. On the other hand, Fig. 4.6 is an example of ineffective refactoring. Although A' itself has very small coupling, two derived methods ($a1$ and $a2$) have rather big coupling. Therefore we should take those derived methods into consideration as well when we evaluate the refactoring effect precisely. The column "average(effect)"

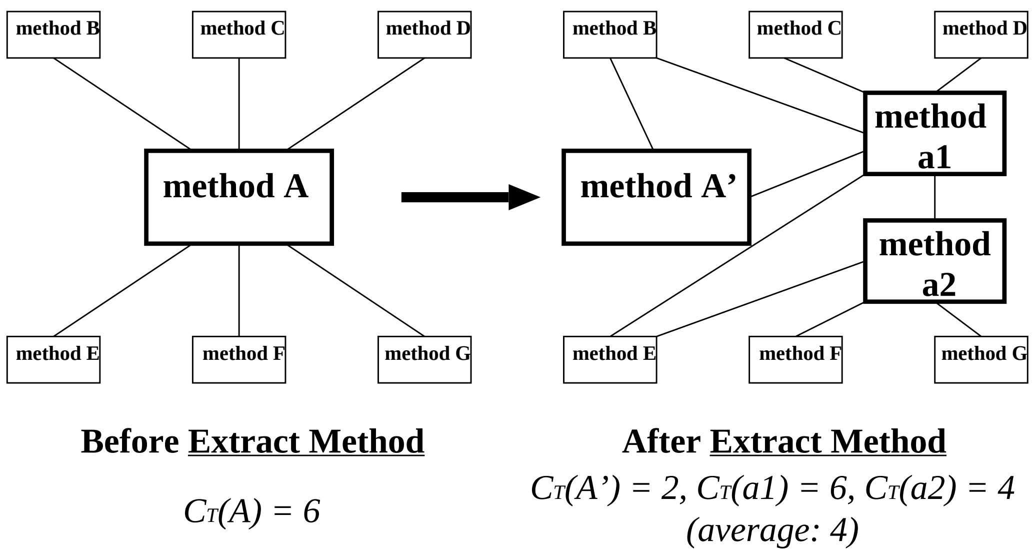

Figure 4.6: Example of Not-So-Good Refactoring

shows the average coupling metrics value of the target method and derived methods added by the refactoring and the difference from before–the–refactoring value, or the effect of the refactoring.

As the result shows, our approach to evaluate the effectiveness of the refactoring using coupling metrics reflects the developer's subjective evaluations quite well.

Regarding case 2, for instance, the developer decided not to refactor the method at least for the time being. He, however, realizes that there is a potential problem which would emerge when he extends the current data structure. The situation is very similar to the bank account program example referred in Section 4.5.

As for case 5, on the other hand, the developer had actually just started considering whether he might introduce a new class for the set of the instance variables. The exceedingly high coupling metrics value encouraged him to implement his idea, and the result was really acceptable.

4.6 Expansion and Related Works

We have shown our first result of quantitative evaluation of the refactoring effect from the coupling metrics viewpoint. Although the result satisfies us, we still have to enhance our method. We need to provide some systematic approach to find appropriate coefficient values to calculate the coupling metrics value. In our experimental case, the developer was both well–experienced and well–trained so that we could rely on his judgment. Working with as many experienced developers as possible is inevitable to define reasonable and versatile coefficient values.

There is another direction we have to explore. As we stated above, there are some other aspects about how to enhance the software maintainability using refactoring, including the size viewpoint, the cohesion viewpoint, the readability viewpoint, and so on. Our experience with various software development projects reveals that some refactorings are so intuitive that almost everyone could agree with the effect of the refactoring in terms of maintainability enhancement. For instance, everyone would admit that a–thousand–LOC–method is much harder to maintain than a–hundred–LOC–method. Another example is that it is much easier if they use reasonable names for their variable, method, and class which reflect their role properly rather than those meaningless identifiers such as "`a`," "`method1`," or "`ClassA`."

On the other hand, the effectiveness of refactorings such as "Pull Up Field" or "Introduce Null Object" is not necessarily obvious. Therefore some quantitative evaluation methods other than the coupling metrics viewpoint are also required to further encourage software development projects to adopt those refactorings.

Our next target will be the cohesion viewpoint. There are various distinguished previous studies on cohesion metrics including Allen et al.'s[1], Briand et al.'s[9], Chae et al.'s[11], Karstu's[25], Bieman et al.'s[6], and so on. As previously discussed, we are currently trying to develop our own metrics as well. Using those existing results and our own approach, we are developing the effect evaluation method for those refactorings intended to enhance the program maintainability in terms of cohesion improvement.

4.7 Conclusion

We have introduced a quantitative method to evaluate the refactoring effect in terms of the degree of maintainability enhancement of the target program. Although the number of refactorings whose effect can be evaluated from the coupling metrics viewpoint is limited, we showed that it is actually useful as an evaluation method of refactoring effect.

Our goal is to establish a comprehensive support of the refactoring process so that the software development project can easily adopt the process and enhance the maintainability of the target software. To achieve our goal, we should provide an effective solution for each subprocess. Implementing the refactoring support tool *Refactoring Assistant*, we have provided solutions for improvement planning subprocess and improvement execution subprocess to a certain extent. In that sense our tool can assist (experienced) developers and analysts who can somehow make reasonable judgment regarding what they should and should not do to improve their products' maintainability. It is essential, however, to provide an effective support for improvement validation to guide project managers toward appropriate direction of the maintainability enhancement. Without assisting project managers responsible for the total quality of the developed software, it would be a far cry from comprehensive improvement of the software maintainability management.

We will continue seeking the way to evaluate refactoring effect based on other metrics. We will also explore how to coordinate those three subprocesses effectively.

Acknowledgements

I would like to thank Mr. Takeo Imai, Mr. Hiroki Andou and Mr. Tetsuji Fukaya of Toshiba Corporation for their contributions to the fundamental result of Chapter 4.

The work of Chapter 4 also owes much to Dr. Nguyen Truong Thang of School of Information Science, Japan Advanced Institute of Science and Technology, who worked with me during his internship program period. And I would like to thank him very much.

Chapter 5

Summary

5.1 Toward Practical Application of Refactoring

As stated very beginning of this thesis, softwares are getting more and more important for various systems and devices around us. That means that the potential risk of social system failure is greatly depends on the quality of softwares. At the same time, the amount of required software is increasing at an exponential rate. Figure 5.1 shows a few example how the total amount of software in an embedded system increases rapidly.

From workforce point of view, on the other hand, there are quite limited number of software developing workforce. According to the surveillance report by Software Engineering Center, for instance, the number of embedded software development workforce is estimated around 175,000. And the increasing rate of the workforce is not very high; only 17% per year. That means we do need any possible software development technologies that drastically enhance the productivity.

We do require software engineering technologies that promise

- securing highly dependable software (components), and

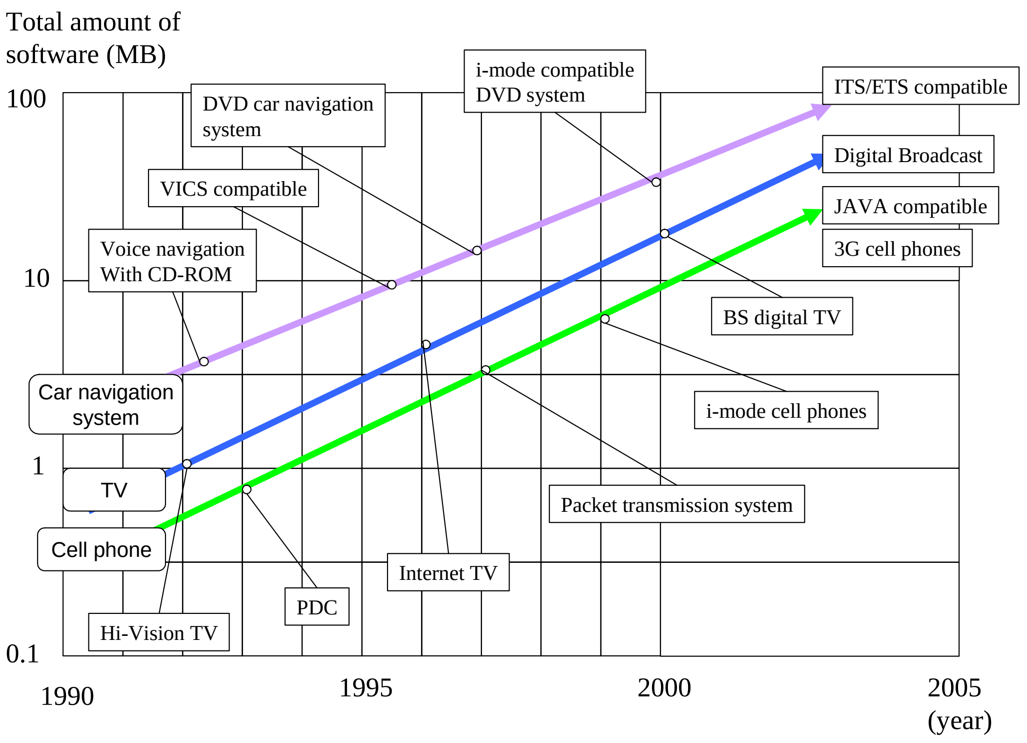

Figure 5.1: Growth of Software Size in Embedded Systems

- increasing productivity.

Program refactoring is one of those promising technology. The problem is that program refactoring itself is not a very handy technology to apply to actual software development, especially those in industrial field where cost and delivery greatly matter. And that is the reason why this series of researches has its value.

Chapter 3 introduced a method and a tool to identify candidates of program refactoring. Making use of this research result, we can greatly lower the first barrier to program refactoring application since identifying where to be refactored is not a very easy task to perform. As for identifying bugs, for instance, there are various technologies including testing, symbolic execution, code review and so on. Although, of course, it is still difficult task to fix those identified bugs, those technologies for identifying bugs have been greatly improving the bug elimination process. Therefore identifying program refactoring candidates might also encourage the practical application of program refactoring as well.

Chapter 4 introduced a method to evaluate the effect of refactorings quantitatively. This result should be a basis for appropriate application of program refactoring. For instance, it is not practical to test all possible usage patterns for a certain software. Especially these days it is not just realistic since softwares can communicate with other systems/devices via networks, which is not necessarily predictable beforehand. Therefore we have to weigh possible use case patterns according to some policy so that we can select appropriate minimum set of test cases. In like wise, evaluating the effect of program refactoring quantitatively, we can weigh possible refactoring patterns so that we can select appropriate minimum set of refactorings.

5.2 Future Works

Table 5.1 shows technology roadmap regarding the refactoring process introduced in Chapter 1. According to the interview on several in-house companies' development sites, there are differences in degree of requirement among the refactoring activities introduced in Chapter 1. The "Needs" column in Table 5.1 indicates the degree of requirement for each activity.

bad–smell detection Many developers agree that it is the first and possibly the highest barrier which makes them hold back from applying refactoring. Although they are very keen to implement required functionality, they usually have little amount of time to consider anything other than that. And after all, most of them do not have slightest idea about what does "non-functional quality" actually mean and how to improve it.

Table 5.1: Technology map in refactoring process

Subprocess	Activity	Support	Needs
Planning	bad–smell detection	**Chapter 3**	High
	bad–smell analysis	partly	Mid
	refactoring planning	partly	Mid
Validation	plan evaluation	**Chapter 4**	High
	refactoring validation	no	High
	functional equivalence validation	no	Low
Execution	refactoring deployment	no	Mid
	refactoring application	partly	High

---

bad–smell analysis Actually most of refactoring patterns we have encountered so far were rather simple refactorings including "Move Method" and "Extract Method," which does not necessarily require very deep analysis. Therefore the support need of this activity is not very high compared with other activities.

refactoring planning Since most of refactoring patterns are rather simple for the time being, required decision making is also rather straightforward. In that sense the support need of this activity is as high as the previous activity.

plan evaluation Although this activity is located in manager level, quantitatively indicating the effect of refactoring is very important to encourage and motivate developers to make use of the merit of refactoring. That is the reason why the need is ranked "High."

refactoring validation Even if the relationship between a certain bad–smell and corresponding refactoring candidate is rather obvious, this activity has to be done appropriately. In addition, as stated below, functional equivalence validation is not a purely refactoring related matter. After all, testing and other software engineering technologies are appropriate to confirm the functionality of a program. From program refactoring viewpoint, this activity is the last stronghold for refactoring application.

functional equivalence validation As mentioned above, this activity is not a purely program refactoring related matter. Although there's no such means from refactoring viewpoint, one can still utilize conventional testing method and others to confirm the functional equivalence before and after the refactoring. After all, developers are never free from any regression tests after any modification to the source code.

refactoring deployment Once the appropriate refactoring plan has been established, most of the refactorings are straightforward as stated in **refactoring planning**. Therefore the support need of this activity is not so high.

refactoring application This is another great barrier which makes developers hesitate to apply refactorings. They dare not modify currently working program. It is sort of their theory not to modify their program unless they encountered bugs. Therefore it will lower the barrier that if there is a technique that secures "a safety program modification."

Chapter 3 and Chapter 4 cover bad–smell detection and plan evaluation respectively. According to the roadmap, next target would be refactoring validation support or refactoring application support. As seen in the previous chapters, there are a number of distinguished preceding researches for refactoring application support. On the other hand, there is no major research for refactoring validation support. Therefore this would be the next target for establishing comprehensive support of program refactoring.

Bibliography

[1] E.B. Allen and T.M. Khoshgoftaar. “Measuring coupling and cohesion: an information-theory approach”. In *Proceedings of the Sixth International Software Metrics Symposium*, pages 119–127, Boca Raton, FL U.S.A., November 1999.

[2] H.G. Baker. “‘Use-once’ variables and linear objects – storage management, reflection and multi-threading”. *SIGPLAN notices*, 30(1):45–52, January 1995.

[3] V.R. Basili, L.C. Briand, and W.L. Melo. “A validation of object-oriented design metrics as quality indicators”. *IEEE Transaction on Software Engineering*, 22(10):751–761, October 1996.

[4] K. Beck. *“eXtreme Programming eXplained: Embrace Change”*. Addison–Wesley, 2000.

[5] P. Bengtsson. “Towards maintainability metrics on software architecture: and adaptation of object-oriented metrics”. In *Proceedings of First Nordic Workshop on Software Architecture*, Ronneby, Sweden, August 1999.

[6] J. M. Bieman and L. M. Ott. “Measuring functional cohesion”. Technical Report CS-93-109, Colorado State University, Fort Collins, CO, USA, June 1993.

[7] R.W. Bowdidge. *"Supporting the Restructuring of Data Abstractions through Manipulation of a Program Visualization"*. PhD thesis, University of California, San Diego, Department of Computer Science & Engineering, November 1995.

[8] R.W. Bowdidge and W.G. Griswold. "Supporting the restructuring of data abstractions through manipulation of a program visualization". *ACM Transactions on Software Engineering and Methodology*, 7(2), April 1998.

[9] L.C. Briand, J.W. Daly, and J.K. Wütt. "A unified framework for cohesion measurement in object-oriented systems". In *Proceedings of the Fourth International Symposium on Software Metrics*, pages 43–53, Albuquerque, NM U.S.A., November 1997.

[10] R.J. Bril and A. Postma. "An architectural connectivity metric and its support for incremental re-architecting of large legacy systems". In *Proceedings of 9th International Workshop on Program Comprehension*, pages 269–280, 2001.

[11] H.S. Chae and Y.R. Kwon. "A cohesion measure for classes in object-oriented systems". In *Proceedings Fifth International Software Metrics Symposium*, pages 158–166, Bethesda, MD U.S.A., November 1999.

[12] M.D. Ernst. *"Dynamically Discovering Likely Program Invariants"*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[13] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. "Dynamically discovering likely program invariants to support program evolution". *IEEE Transaction on Software Engineering*, 27(2):1–25, February 2001.

[14] M.D. Ernst, A. Czeisler, W.G. Griswold, and D. Notkin. "Quickly detecting relevant program invariants". In *Proceedings of 22nd International Conference on Software Engineering*, pages 449–458, Limerick, Ireland, June 2000.

[15] M.D. Ernst, W.G. Griswold, Y. Kataoka, and D. Notkin. "Dynamically discovering pointer-based program invariants". Technical report, University of Washington, Seattle, WA, November 1999.

[16] M. Fowler. *"Refactoring: Improving the Design of Existing Code"*. Addison–Wesley, 1999.

[17] E. Gamma, R. Helm, R.E. Johnson, and J. Vlissides. *"Design Patterns"*. Addison–Wesley, 1995.

[18] W.G. Griswold. *"Program Restructuring as an Aid to Software Maintenance"*. PhD thesis, University of Washington, Dept. of Computer Science & Engineering, Seattle, WA, August 1991.

[19] W.G. Griswold. "Coping with change using information transparency". Technical Report CS98-585, University of California, San Diego, Department of Computer Science and Engineering, April 1998 (Revised August 1998).

[20] W.G. Griswold, M.I. Chen, R.W. Bowdidge, J.L. Cabaniss, V.B. Nguyen, and J.D. Morgenthaler. "Tool support for planning the restructuring of data abstractions in large systems". *IEEE Transactions on Software Engineering*, 24(7):534–558, July 1998.

[21] W.G. Griswold and Notkin D. "Automated assistance for program restructuring". *ACM Transaction on Software Engineering and Methodology*, 2(3):228–269, July 1993.

[22] W.G. Griswold, J.J. Yuan, and Y. Kato. "Exploiting the map metaphor in a tool for software evolution". In *Proceedings of the 2001 International Conference on Software Engineering*, May 2001.

[23] K. Hatano, Y. Nomura, H. Taniguchi, and K. Ushijima. "A mechanism to support automated refactoring process". *IPSJ Journal*, 44(6):1548–1557, June 2003.

[24] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. "Aries: Refactoring support environment based on code clone analysis". In *Proceedings of the 8th IASTED International Conference on Software Engineering and Applications (SEA 2004)*, pages 222–229, 2004.

[25] S. Karstu. "An examination of the behavior of the slice based cohesion measures". Master's thesis, Department of Computer Science, Michigan Technological University, 1994.

[26] Y. Kataoka, M.D. Ernst, W.G. Griswold, and D. Notkin. "Automated support for program refactoring using invariants". In *ICSM 2001, Proceedings of the IEEE International Conference on Software Maintenance*, pages 736–743, Florence, Italy, November 2001.

[27] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. "A quantitative evaluation of maintainability enhancement by refactoring". In *ICSM 2002, Proceedings of the IEEE International Conference on Software Maintenance*, pages 576–585, Montreal, Canada, November 2002.

[28] K.A. Kontogiannis, R. DeMori, E. Merlo, and M. et al. Galler. "Pattern matching for clone and concept detection". *Automated Software Engineering*, 3(1–2):77–108, June 1996.

[29] I. Moore. "Automatic inheritance hierarchy restructuring and method refactoring". In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 235–250, 1996.

[30] I. Moore. *"Automatic Restructuring of Object Oriented Programs"*. PhD thesis, University of Manchester, 1996.

[31] W.F. Opdyke. *"Refactoring Object-Oriented Frameworks"*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[32] W.F. Opdyke and R.E. Johnson. "Refactoring: An aid in designing application frameworks and evolving object-oriented systems". In *Proceedings of SOOPPA '90: Symposium on Object-Oriented Programming Emphasizing Practical Applications*, Sep 1990.

[33] D. Roberts, J. Brant, and R. Johnson. "A refactoring tool for smalltalk". *Theory and Practice of Object Systems*, 3(4):253–63, 1997.

[34] D.B. Roberts. *"Practical Analysis for Refactoring"*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[35] W.P. Stevens, G.J. Myers, and L.L. Constantine. "Structured design". *IBM Systems Journal*, 13(2):115–139, 1974.

[36] D. Ungar and R.B. Smith. "Self: The power of simplicity". In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 227–241, 1987.

[37] S. Wake and S. Henry. "A model based on software quality factors which predicts maintainability". In *Proceedings of the Conference on Software Maintenance*, pages 382–387, 1988.