Automated Support for Program Refactoring using Invariants

Yoshio Kataoka,[♠] Michael D. Ernst,[♡] William G. Griswold,[◇] David Notkin[♠]

[♠]Dept. of Computer Science & Engineering
University of Washington
Box 352350, Seattle WA 98195-2350 USA
{kataoka,notkin}@cs.washington.edu

[◇]Dept. of Computer Science & Engineering
University of California San Diego, 0114
La Jolla, CA 92093-0114 USA
wgg@cs.ucsd.edu

[♡]MIT Lab for Computer Science
545 Technology Square
Cambridge, MA 02139 USA
mernst@lcs.mit.edu

Abstract

Program refactoring — transforming a program to improve readability, structure, performance, abstraction, maintainability, or other features — is not applied in practice as much as might be desired. One deterrent is the cost of detecting candidates for refactoring and of choosing the appropriate refactoring transformation. This paper demonstrates the feasibility of automatically finding places in the program that are candidates for specific refactorings. The approach uses program invariants: when a particular pattern of invariant relationships appears 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.

1 Introduction

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 [OJ90, Gri91, Opd92, GN93], refactoring is increasingly a part of mainstream software development practices [FBB+99]. As just one example, one of the basic tenets of Extreme Programming [Bec99] is to refactor on a continual basis, as a fundamental part of the software development process.

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 2).

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 [GHJV95]. 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.

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 [Ern00, ECGN01].

In the following we discuss prior work in refactoring (Section 2), Daikon's approach to detecting invariants (Section 3), our approach to finding refactoring candidates (Sections 4 and 5), and a case study of its use (Section 6). We close with a comparison of dynamic and static refactoring detection (Section 7) and a discussion of contributions and future work (Section 8).

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 paper 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 seem to dovetail quite naturally.

Opdyke [OJ90, Opd92] and Griswold [Gri91, GN93] 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 [RBJ97], which automatically performs a set of refactorings taken primarily from 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 [Rob99] 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 [Moo96b, Moo96a] 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 [US87]. Using a similar algorithm, Guru can also automatically extract shared expressions from methods.

Bowdidge's Star Diagram [Bow95, BG98] 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 [GCB+98]. The tool does not recommend specific refactorings, and the tool user must identify the variables or data structures that are candidates for refactoring.

[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 [Gri91] and Roberts [Rob99].

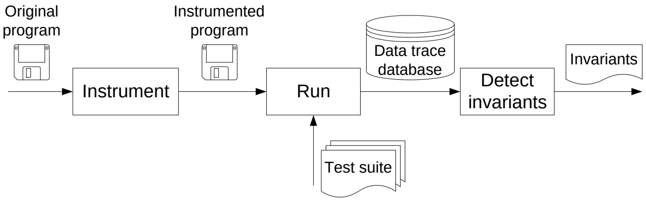

Figure 1: An overview of dynamic invariant inference as implemented by the Daikon tool.

Finding Duplication in Software. Other related work identifies potential duplication in software systems. Baker [Bak95], 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. [KDM+96] 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 Invariant Discovery

Dynamic invariant detection [ECGN01] 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 1). The inference step tests a set of possible invariants against the 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 node = node.child.parent (for all nodes), Daikon detects global invariants over pointer-directed data structures, such as mytree is sorted by $\leq$. Finally, Daikon can detect conditional invariants that are not universally true, such as if $\mathsf{p} \neq \mathsf{NULL}$ then $*\mathsf{p} > \mathsf{x}$ and $\mathsf{p.value} > \mathsf{limit}$ or $\mathsf{p.left} \in \mathsf{mytree}$. 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 [EGKN99].

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 $\mathsf{i} < \mathsf{size}(\mathsf{A})$ 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 [ECGN00].

The Daikon invariant detector is available for download from `http://sdg.lcs.mit.edu/~mernst/daikon/`.

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" [FBB+99, 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 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" [FBB+99, 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 potential refactoring candidate and the refactoring to be applied to it.

To discover candidate refactorings using invariants, we studied the refactorings in Fowler et al. [FBB+99] to determine whether there were invariants that would indicate the applicability of the refactoring.

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 there is an invariant over the parameters that implies one 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.)

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. (Section 7 compares dynamic and static analysis for identifying candidate refactorings.)

5 Candidate Refactorings Discoverable from Invariants

This section describes the refactorings (largely from Fowler et al. [FBB+99]) that our tool detects, and specifies the invariants that indicate their applicability. Section 6 presents a case study over Nebulous [GYK01], including quantitative and qualitative results. We use some examples from this case study to clarify the refactorings discussed here.

5.1 Remove Parameter

Remove Parameter is intended to apply when "a parameter [is] no longer used by the method body" [FBB+99, 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.

5.2 Eliminate Useless Return Value

Eliminate Useless 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 [FBB+99], its rationale and mechanics are similar to those of Remove Parameter (Section 5.1).

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

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

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.

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" [FBB+99, 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 $\mathsf{return} = \mathit{constant}$, 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 Useless Return Value refactoring (Section 5.2) will be recommended. Postconditions can imply $\mathsf{a} \neq \mathsf{orig}(\mathsf{a})$ by, for example, containing $\mathsf{a} = \mathit{func}(\mathsf{a})$, where *func* is not the identity function.

Nebulous's `CursorHistory.GetNextItem` method, which returns a `CursorHistoryItem` object, 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.

5.4 Encapsulate Downcast

Encapsulate Downcast is intended for "a method that returns an object that needs to be downcasted by its callers" [FBB+99, 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:

- $\mathsf{LUB(return.class)} \neq$ *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. $\mathsf{return.class} =$ *constant*, and
2. $\mathsf{return.class} \neq$ *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.

5.5 Replace Temp with Query

Replace Temp with Query is intended for "a temporary variable that holds the value of an expression" [FBB+99, 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:

- $\mathsf{temp} = \mathsf{orig(temp)}$, and
- $\mathsf{a} = \mathsf{orig(a)}$, $\mathsf{b} = \mathsf{orig(b)}, \ldots$ for all variables `a`,`b`,... in the (side-effect-free) initializer for `temp`

Section 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 define temps via parameter passing, thus making them visible to Daikon. Using the Daikon output of 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.

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 [GYK01]. Nebulous is a software tool that employs simple pattern matching and the 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 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 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 Useless Return Value	1	2	4	7
Separate Query from Modifier	0	2	0	2
Encapsulate Downcast	1	1	0	2
Total	9	8	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 [FBB+99, 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 Useless 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's more obscure features. (This weakness is specifically due to our use of a dynamic method for discovering invariants; that is, 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 [Gri98] 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.

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, rely on 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 2). Deferring checking should be acceptable when most candidates are indeed applicable, as is the case for our tool.

A combination of underlying analyses seems best for automated refactoring. Consider again the question of when the refactoring Remove Parameter 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 [FBB+99, 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.

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 two especially attractive properties of our approach: first, the computation of the invariants and the identification of refactoring candidates is automatic; and 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.

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 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 [ECGN00] 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.

Acknowledgments

The authors thank Wesley Leong for evaluating our tool's recommendations on Nebulous, and the anonymous referees for comments that improved the presentation. This research was supported in part by NSF grant CCR-9970985, an IBM Graduate Fellowship, and gifts from Edison Design Group, Microsoft Corporation, and Toshiba Corporation. Kataoka is currently visiting UW from the System Engineering Laboratory, Research and Development Center, Toshiba Corporation.

References

[Bak95] Henry G. Baker. 'Use-once' variables and linear objects – storage management, reflection and multi-threading. *SIGPLAN Notices*, 30(1):45–52, January 1995.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[BG98] Robert W. Bowdidge and William 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.

[Bow95] Robert 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. Technical Report CS95-457.

[ECGN00] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, Limerick, Ireland, June 7–9, 2000. To appear in *ACM Transactions on Software Engineering and Methodology*.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001. A previous version appeared in *Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[EGKN99] Michael D. Ernst, William G. Griswold, Yoshio Kataoka, and David Notkin. Dynamically discovering pointer-based program invariants. Technical Report UW-CSE-99-11-02, University of Washington, Seattle, WA, November 16, 1999.

[Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[GCB+98] William G. Griswold, Morison I. Chen, Robert W. Bowdidge, Jenny L. Cabaniss, Van B. Nguyen, and J. David Morgenthaler. Tool support for planning the restructuring of data abstractions in large systems. *IEEE Transactions on Software Engineering*, 24(7):534–558, July 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[GN93] William G. Griswold and David Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, July 1993.

[Gri91] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, Dept. of Computer Science & Engineering, August 1991. Technical Report No. 91-08-04.

[Gri98] William 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).

[GYK01] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 2001 International Conference on Software Engineering*, May 2001.

[KDM+96] K. A. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and Others. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1–2):77–108, June 1996.

[Moo96a] Ivan 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.

[Moo96b] Ivan R. Moore. *Automatic Restructuring of Object-Oriented Programs*. PhD thesis, University of Manchester, 1996.

[OJ90] William F. Opdyke and Ralph 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.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[RBJ97] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–63, 1997.

[Rob99] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 227–241, 1987.