Lightweight Detection of Program Refactorings

Darren C. Atkinson and Todd King
Department of Computer Engineering
Santa Clara University
Santa Clara, CA 95053-0566 USA
{datkinson,tking}@scu.edu

Abstract

Poorly structured code is hard to maintain and read. Program refactoring can improve code structure and thus make it easier to preserve and to discern the underlying design. However, refactoring is a difficult and time-consuming process making it unattractive for many developers. An automated tool that could identify poorly structured code and make suggestions would make the refactoring process easier. Although in general refactorings may be quite difficult to locate automatically, we show that many can be detected using low-cost, syntactic techniques. We have built a tool to locate refactorings in C# programs. Our experiments indicate that the tool has an excellent success rate in identifying refactorings.

1. Introduction

A recent study estimated that 80% of all software development cost is applied toward maintaining software [10]. Software maintenance can take a variety of forms. A programmer may need to incorporate an enhancement requested by the customer. The software may need to be adapted for a new architecture or platform. Defects in the design or implementation may need to be corrected. Finally, the engineer may simply wish to restructure the system, improving its design and organization, to ease incorporation of future changes.

The goal of such preventive maintenance is to reduce future maintenance costs. No matter how good an original design may appear to be, inevitably unexpected problems arise during development and workarounds must be found. Over time, the program code is modified and rearranged, and the code gradually deteriorates from the originally conceived structure. The resulting code is hard to read and understand. Program refactoring attempts to improve the code structure. Specifically, program refactoring is "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [4].

Refactoring improves the readability of code and, by forcing the developer to carefully examine the code, he achieves a new understanding of the code, thus making identifying errors much easier. Refactoring can decrease the development and maintenance time of a project. Unfortunately, refactoring is a difficult and time-consuming process, which makes it an unattractive option for many developers.

Although several refactoring tools exist [8, 14, 16] they tend to focus on applying refactorings rather than on detecting them. The tool user selects a piece of code and a refactoring to perform and the tool will restructure the code automatically, making consistent changes throughout the program. However, we believe that locating the refactorings rather than affecting the changes is more difficult to do by hand. Furthermore, detection is a necessary first step in the refactoring process. Once the refactorings are found, an automated tool can then be used to perform the refactorings. Thus, automated tools for detecting program refactorings are highly desirable. Such tools would examine the source code, looking for locations of possible refactorings.

To derive information from source code, static analysis tools such as program slicing tools [17] are often employed. Such tools analyze the program source, computing information about the program such as data and control dependences, and then present that information in a way that is useful to the tool user. However, such tools are not without their problems. First, they are difficult to build, requiring interprocedural data-flow analyses that go beyond what a professional optimizing compiler might require. Even building parsers for languages such as C++, C#, and Java that have a rich set of features can be difficult. Second, the analyses themselves can be quite slow, often requiring quadratic time in the size of the program. Finally, the results are often less than desirable because the tools are conservative in the sense that their results are correct for any input and execution path [9]. For example, if a function is encountered whose implementation cannot be found, the tool may need to conservatively estimate that any object that could be directly or indirectly referenced by the called function is possibly modified.

We classified the approximately seventy refactorings discussed in [4]. About one third appear to be too high-level for any automated tool to detect (or perform). For example, *substitute algorithm* suggests replacing an existing algorithm with another that is easier to understand. However, we estimate that about half of the refactorings can be discovered using mostly syntactic means. Therefore, rather than develop a static analysis tool that uses data-flow analysis to detect refactorings, we decided to build a tool that uses purely syntactic data such as symbol table information and simple code metrics.

In our experiments, we analyzed a number of public domain C# programs in an attempt to detect a variety of refactorings. Our results indicate that low-cost syntactic techniques have an excellent success rate in detecting many program refactorings. To validate our results, we examined each suggested refactoring by hand to determine if it was valid. We define success rate as the number of correctly identified refactorings. There may be additional refactorings that the tool could not find, but for the purposes of this study we are only concerned with identifying correct refactorings. A tool that simply produces a long list of refactorings, only a small number of which are valid, will most likely not be used. In detail, this paper makes the following contributions:

- We show how many refactorings can be codified using syntactic information and simple code metrics such as line count.
- We report results of analyzing a number of C# programs looking for possible refactorings, thereby assessing their quality.
- We show that for most refactorings, we obtain an excellent success rate for the programs in our test suite.

Finally, although our experiments use C# programs, the detection mechanisms apply almost without modification to other object-oriented languages such as Java.

2. Refactoring

Refactoring [4, 8, 11] (also known as restructuring [2, 3, 6] when applied to non-object-oriented systems) is changing the structure of a system so that future maintenance is easier. Typically, refactoring transformations do not change the semantics of the program, but only improve its structure. The key motivation for refactoring is that the original design of the system was either inappropriate or has degraded over time due to accommodating unexpected features or requirements. Since today's software development is market-driven, little time is typically given to reworking an earlier design even though doing so might reduce future development costs.

A wide range of refactoring algorithms or transformations exist, from the simple to the complex. We present a brief list of refactorings, many of which we considered in our study:

- *encapsulate field*: a public field should be declared either private or protected and a property or method should be used to access the field
- *extract class/method*: a large, complex class/method should be split into smaller, dedicated classes/methods
- *decompose conditional*: a complex conditional expression should be replaced with a self-explanatory (i.e., well-named) method call as shown in Figure 1
- *replace magic number*: a literal with a particular meaning should be replaced with a symbolic constant
- *hide method*: a public method that is not used outside of its class should be declared private or protected
- *move method/field*: a method/field is using/used by more features of another class than the class in which it is declared and therefore should be moved
- *remove unused method/field*: a method/field is not used and should therefore be removed
- *replace parameter with explicit methods*: a method that performs different tasks based on a parameter should be replaced with separate methods for each task
- *split temporary variable*: if a temporary variable is defined more than once, but is not aggregating information, a separate temporary variable for each definition should be used
- *rename method*: the name of a method does not reveal its purpose, so the method should be renamed
- *substitute algorithm*: a complex algorithm should be replaced with one that is clearer

Clearly, some of the refactorings listed such as *substitute algorithm* and *rename method* are sufficiently high-level as to be beyond the scope of any tool to detect, while others such as *encapsulate field* are trivial to detect.

3. Detecting Refactorings

We wanted to determine how well refactorings could be detected using low-cost syntactic techniques. Our belief is that many refactorings can be detected with a high rate of success (i.e., few false positives). We have built a tool called

```
if ((water.getPressure() > THRESHOLDPSI && water.getTemp() >= BOILING)
 || (water.flowRate() > MAXFLOWRATE && this.state == State.OPEN)) ...
```
(a)

```
if (pressureValveFailed(water)) ...
```
(b)

Figure 1. Example of the *decompose conditional* refactoring: (a) before refactoring and (b) after refactoring.

Look# for automatically detecting refactorings in C# programs. We briefly describe our tool architecture to illustrate the kind of information that we have available for the refactoring detection algorithms, which are then described.

3.1. Tool architecture

We used the ANTLR system [12] for constructing a lexical analyzer and parser for C#. The parser constructs an abstract syntax tree (AST) [1] for each file in the program. The ASTs for all files are retained in memory and linked together to form a single AST for the entire program. Several passes are then made over the tree, with each pass requiring linear time in the size of the program:

1. *Scope construction*: Traverse the AST. If a scope delimiter is found, create a new hash table for symbols in the scope. If a symbol is found, insert the symbol into the hash table for the current scope.
2. *Type resolution*: Traverse the AST. If a symbol is found, then lookup the symbol's type in the symbol table and link this node to its declaration node.
3. *Class hierarchy resolution*: Traverse the scope hierarchy. For each node that may be inherited, lookup its base types in the symbol table and add them to this node's base type list, and add this node to the base type's derived type list.
4. *Forward reference construction*: Traverse the AST. If an identifier is found, then lookup its declaration in the symbol table and add this identifier as a reference. Additionally, if the symbol is for a method, then conservatively add a reference in each derived class.

Some passes could probably be combined, but in general multiple passes are necessary for languages such as C# and Java that do not require types to be declared before use. After these passes are complete, we can lookup a declaration of a symbol given a reference (i.e., use or definition) and find all references given a declaration. Similarly, we can find all parent classes of a child class and find all child classes of a parent.

3.2. Detecting refactorings

We describe several refactoring detection algorithms that we have implemented as part of *Look#*. All algorithms that use counts or other metrics have thresholds that can be customized by the tool user. The algorithms are listed in order of complexity.

Encapsulate field: Construct a list of all field members. Any field member that is declared public is flagged. This is easily the simplest refactoring to detect.

```
for $c \in classes[program]$ do
  for $f \in fields[c]$ do
    if $\text{PUBLIC} \in modifiers[f] \wedge \text{CONST} \notin modifiers[f]$ then
      $report(\text{"encapsulate field"}, f)$
```

Decompose conditional: Traverse the AST searching for conditional (i.e., logical) expressions. Count the number of boolean OR and AND operators and the number of statements in the method body containing the expression. If both counts exceed their respective thresholds, then suggest decomposing the conditional.

The motivation for the threshold on the minimum number of statements is to avoid suggesting decomposing conditionals that have already been moved into their own methods. Before adding this simple threshold, the tool was suggesting decomposing conditionals in methods where clearly refactorings had already been performed.

```
for $c \in classes[program]$ do
  for $m \in methods[c]$ do
    $\text{DFS}(m)$

function DFS(n) do
  if $type[n] = \text{AND} \vee type[n] = \text{OR}$ then
    $count := 1$
    for $d \in descendents[n]$ do
      if $type[d] = \text{AND} \vee type[d] = \text{OR}$ then
        $count := count + 1$
    if $count > \alpha \wedge |statements[method\text{-}containing(n)]| > \beta$ then
      $report(\text{"decompose conditional"}, n)$
  else
    for $c \in children[n]$ do
      $\text{DFS}(c)$
```

Replace magic number: This algorithm is complicated by the fact that three different levels need to be maintained: global level, type (i.e., class or struct) level, and method level. The rationale for the different levels is that a literal may be used many times, but if it used in different classes or methods, then it may not in fact represent the same thing and cannot be replaced with a single symbolic constant.

The tool traverses the AST and updates the level when encountering a class or method declaration. Each literal found is added to the current scope. When the end of a construct such as a method is reached, the scope is closed and a literal whose occurrence count exceeds a threshold is reported. (Each scope level has its own adjustable threshold.)

Program	Version	Lines	Methods	Classes	Time	Description
FCKeditor	2.0	809	34	6	0.55	online text editor
MyACDSee	1.3	4118	137	14	1.48	image viewer
AscGen	2.0.2.4	4762	168	17	1.72	image conversion
TVGuide	0.4.3.1.3	6231	296	55	1.78	download and display TV listings
MFXStream	1.0.0	7077	323	24	1.96	streaming media file server
GmailerXP	0.7	9347	288	29	2.86	mail client
3DProS	1.0	13629	381	22	3.81	3D animation
HeroStats	2.2.1	28840	1317	115	6.29	gaming statistics
ZedGraph	1.0	33539	1193	107	4.51	graphing package
NAnt	0.85	80986	3416	408	32.31	Ant-like build tool

Table 1. Description of programs used in the experiments.

The occurrence counts of the current scope are merged with those of the parent scope. This step is done so if a number appears frequently throughout a class but only once or twice per method, we will still detect it as a magic number for the entire class.

Certain uses of literals must be ignored such as those in expressions when initializing a constant (i.e., when the symbolic constant itself is declared) and in functions not written by hand but rather generated by the development environment. For example, Microsoft Visual Studio generates a method called InitializeComponent that is blacklisted by default in *Look#*.[1] Furthermore, certain literals such as zero and one must be ignored in most, but not all, circumstances. For example, if such a literal is used an array index, then the use should probably be flagged.

```
for $c \in classes[program]$ do
  for $m \in methods[c]$ do
    for $n \in body[m]$ do
      if $type[n] = \text{LITERAL} \wedge type[parent[n]] \neq \text{INITIALIZER}$ then
        $method\text{-}count[value[n]] := method\text{-}count[value[n]] + 1$
    for $v \in method\text{-}count$ do
      if $method\text{-}count[v] > \alpha$ then
        $report(\text{“magic number”}, v, m)$
      else
        $class\text{-}count[v] := class\text{-}count[v] + method\text{-}count[v]$
    delete $method\text{-}count$
  for $v \in class\text{-}count$ do
    if $class\text{-}count[v] > \beta$ then
      $report(\text{“magic number”}, v, c)$
  delete $class\text{-}count$
```

Move method: This detection algorithm is based on the data class “smell” [4]. A data class is a class that holds data but does not make use of the data itself; it merely holds the data for other classes to use. In our experiments, we found that restricting our detection algorithm to this “smell” helped increase the accuracy of the *move method* detection algorithm, at the cost of ignoring other possible instances where the refactoring should be applied between two non-data classes.

1 The tool user can add functions to the blacklist as necessary. Each refactoring has its own blacklist.

The tool traverses the AST to create a list of all classes and structs in the program. For each class or struct in the list, we count the number of fields and methods declared. If there is a low ratio of methods to fields, then we assume that the class is a data class and target it as a candidate class for *move method*.

Once a candidate class has been found, for each field in the class we determine the set of methods referencing the field by using the forward reference chains previously built. If a foreign method makes too many references to a number of distinct fields, then we suggest moving the method or some of its functionality into the data class. The additional check on the number of distinct fields is needed since some methods may reference a single field many times, but in fact a temporary variable could have been used to reference the field only once.

```
for $c \in classes[program]$ do
  if $|fields[c]| > \alpha \cdot |methods[c]|$ then
    for $f \in fields[c]$ do
      for $r \in references[f]$ do
        $m := method\text{-}containing(r)$
        $fields\text{-}referenced[m] := fields\text{-}referenced[m] \cup \{f\}$
        $total\text{-}references[m] := total\text{-}references[m] + 1$
    for $m \in fields\text{-}referenced$ do
      if $total\text{-}references[m] > \beta \wedge |fields\text{-}referenced[m]| > \gamma$ then
        $report(\text{“move method”}, m, c)$
    delete $total, refs$
```

4. Experimental Results

In order to validate our hypothesis that lightweight techniques can be used to accurately detect program refactorings, we analyzed ten public domain C# programs using *Look#*. General information for each program is given in Table 1. The programs are listed in this and other tables in order of increasing size. All programs are publicly available from SourceForge. The time in seconds required by *Look#* to construct the syntax tree and execute all of the refactoring detection algorithms is also given.

For each program in our test suite, Table 2 gives the number of refactorings detected. Not surprisingly, more refactorings were generally detected for the larger programs. We were surprised by the high detection count for *unused method*, since this refactoring indicates dead code that could be removed. Most of the programs analyzed are immature and thus would not have evolved as to have so much dead code. Rather, we attribute the high count to the fact that these programs *are* immature, and therefore have features that are only partially implemented. Additionally, we found that some programs were providing a more complete interface for a class than was necessary for their own use. For example, *ZedGraph* provides an implementation of a collection with methods for adding and removing elements, but itself only adds elements to the collection.

Of course, all refactorings detected by *Look#* are not necessarily correct. We verified each reported refactoring by hand to determine its correctness. Table 3 lists the percentage of refactorings that were correct for each program. Table 4 provides a summary across all programs.

Encapsulate field: Not surprisingly, *encapsulate field* has a perfect score. However, the high count was slightly surprising. We expected many developers to understand that public fields are a violation of encapsulation. Furthermore, C# provides *properties* that can be used to transparently and safely encapsulate a public field. Our results would seem to indicate that this mechanism is not being used as much as it should be.

Remove empty method: Interestingly, this algorithm does not have a perfect score because although *Look#* correctly detects that the method is empty, the method cannot always be removed because it is required to implement an external interface. This was only a problem for *NAnt*, which uses a large number of external libraries.

Remove unused field/method: *Remove unused field* works well in most cases. For *TVGuide*, the erroneous refactorings were due to complex array indexers that *Look#* does not yet handle. The high success rate for *unused method* indicates that most of the methods that we thought were unused, are in fact unused. Some incorrectly reported refactorings are due to the necessity of implementing the method in order to implement an interface. Other errors stem from the use of a complex expression (e.g., nested array references) to compute the object being referenced or from variable argument lists, which *Look#* cannot yet handle.

Extract class/method: The *extract class* refactoring was detected infrequently, which may indicate that our thresholds are too high. However, since it had one of the lower average success rates, increasing the thresholds would probably result in more erroneous refactorings. More likely, a new detection algorithm is needed. In contrast, *extract method* works well and has a much higher number of occurrences. The difference in results between the two refactorings is surprising since they use similar algorithms: *extract class* counts the number of lines, methods, and fields while *extract method* counts the number of statements and lines. Although *extract method* experiences more failures on the larger programs, we determined that size was not the cause. Most failures are due to methods that have large switch statements that are difficult to extract. The few other failures were true failures in the sense that the methods themselves were large, highly cohesive methods that would be very difficult to extract. Rather, extracting part of the method could instead reduce understandability.

Decompose conditional: This detection algorithm has almost a perfect success rate, with the only errors occurring in *ZedGraph*. One of these had an object creation as part of the conditional expression and therefore the expression could not be moved. This is certainly a case where using data-flow analysis would have eliminated the error. However, this particular case accounts for less than 2% of its reported refactorings, further confirming our hypothesis that refactorings can be correctly detected using purely syntactic techniques.

Replace magic number: Like *extract method*, the *replace magic number* refactoring experiences more failures on the larger programs. However, we again determined that size was not the inherent reason for the failures. The larger programs such as *NAnt* and *ZedGraph* provide scaffolding to facilitate testing, and this code contains many instances of literals that are themselves unrelated. For example, *NAnt* provides unit tests and the expected results of several tests happen to be the same integer; however, the values themselves are not related, so replacing them with a symbolic constant would be inappropriate, as shown below:

```
AssertExpression("1 + 2", 3);
AssertExpression("1 + 2 + 3", 6);
AssertExpression("1 + 2 * 3", 7);
AssertExpression("2 * 1 * 3", 6);
AssertExpression("1 / 2 + 3", 3);
```

Move field: While we are pleased with the success rate for most of the refactorings, the success rate for *move field* is definitely disappointing. We found that in most cases, *Look#* suggested moving a field from a data class that had multiple instances to a data processing class that had only a single instance. For example, a student data class may have a field name that is seldom referenced within the class, but is referenced frequently by other classes that perform data processing tasks such as printing rosters or calculating payments. It is impossible to move such fields to another class even if that class is the only one referencing the fields because of the number of instances. To overcome this problem, we are considering restricting the set of target fields to static fields. Of course, this change will result in far fewer refactorings being detected. The much higher success rate for *3DProS* is interesting. This program uses a single struc-

	FCKeditor	MyACDSee	AscGen	TVGuide	MFXStream	GmailerXP	3DProS	HeroStats	ZedGraph	NAnt	Total
encapsulate field	0	13	6	102	13	39	128	177	226	29	733
remove empty method	1	6	1	2	3	2	0	6	0	22	43
remove unused field	0	4	2	51	1	3	2	77	5	16	161
remove unused method	2	6	7	40	11	19	14	142	98	864	1203
hide method	3	4	4	12	10	13	16	59	60	74	255
extract class	0	0	0	0	1	0	2	3	1	0	7
extract method	0	1	1	1	0	5	10	5	9	28	60
decompose conditional	0	1	3	0	0	1	3	7	32	10	57
replace magic number	1	13	15	5	21	34	40	18	78	28	253
move field	0	0	1	4	7	9	34	24	11	39	129
move method	0	0	1	11	8	13	33	15	32	48	161
total	7	48	41	228	75	138	282	533	552	1158	3062

Table 2. Number of refactorings detected for each program.

	FCKeditor	MyACDSee	AscGen	TVGuide	MFXStream	GmailerXP	3DProS	HeroStats	ZedGraph	NAnt
encapsulate field	—	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
remove empty method	100.0	100.0	100.0	100.0	100.0	100.0	—	100.0	—	54.5
remove unused field	—	100.0	100.0	100.0	100.0	100.0	100.0	94.8	100.0	87.5
remove unused method	0.0	83.3	42.9	67.5	90.9	94.7	78.6	97.2	58.2	94.9
hide method	100.0	100.0	100.0	100.0	80.0	100.0	75.0	100.0	95.0	82.4
extract class	—	—	—	—	0.0	—	100.0	33.3	100.0	—
extract method	—	100.0	100.0	100.0	—	100.0	70.0	80.0	88.9	92.9
decompose conditional	—	100.0	100.0	—	—	100.0	100.0	100.0	93.8	100.0
replace magic number	100.0	100.0	100.0	100.0	85.7	100.0	100.0	66.7	75.6	75.0
move field	—	—	0.0	0.0	0.0	0.0	61.8	0.0	9.1	0.0
move method	—	—	100.0	72.7	100.0	100.0	45.5	46.7	46.9	35.4

Table 3. Success rates of refactorings detected for each program.

ture with static fields that are defaults for the entire program or serve as global variables. However, these fields are often only used in a single class, and thus perhaps should be moved to that class.

In summary, *Look#* detected over three thousand refactorings of which approximately 88% were identified as correct. We are pleased with the success rates of all the individual refactorings with the exceptions of *extract class* and *move field*. We could improve the success rates of many of the other refactorings both by improving the tool's ability to correctly determine types in complex expressions (e.g., nested array references) and also by analyzing precompiled libraries to determine the properties of external types such as interfaces and delegates.

5. Related Work

Mens and Tourwé [8] provide an excellent survey of software refactoring, including refactoring at both the design and source code level. In work closest to ours, Simon et al. [13] use metrics to detect refactorings. They employ a distance metric to measure the cohesion of a class, which works better for some refactorings than our approach of using reference counts. Their work is aimed at software visualization. They use the distance metric to determine the positions of objects in a three-dimensional view of the program. The tool user can then look for fields and methods that are isolated or do not belong with other objects in a cluster. Because of the visualization aspect, their approach

	Correct	Reported	Rate(%)
encapsulate field	733	733	100.0
remove empty method	33	43	76.7
remove unused field	155	161	96.3
remove unused method	1089	1203	90.5
hide method	233	255	91.4
extract class	4	7	57.1
extract method	53	60	88.3
decompose conditional	55	57	96.5
replace magic number	218	253	86.2
move field	22	129	17.1
move method	84	161	52.2
total	2679	3062	87.5

Table 4. Summary of refactoring detections.

works well only when used on a few classes. Finding refactorings in a large visualization space can be difficult, and the time required by their tool to layout the space can be prohibitive. In contrast, our approach is inexpensive and scales well. Interesting, their approach to detecting *move field* suffers from the same problem as ours: moving a field from a class with multiple instances to a class that has only a single instance.

Tourwé and Mens [15] use a logic programming approach to identify refactorings. Program facts are computed and stored in a database that can then be queried using Prolog. The detection algorithms are themselves coded in Prolog. One advantage of this separation is that should the program facts change (e.g., become more accurate if a different code analysis is used), the queries themselves do not change. However, the authors admit that program facts are not enough to detect some refactorings and that metrics will probably need to be added. Although they report a perfect success rate, they only analyzed the tool itself and tried to identify only two refactorings. Our experiments cover a much larger set of programs and refactorings.

Kataoka et al. [7] detect refactorings using program invariants. Likely program invariants are computed from test runs and then used to locate refactorings. For example, the *remove parameter* refactoring can be applied when the parameter is not used by the method body or when its value is always a constant. The latter is a program invariant that their tool can detect. The authors present a small case study of a single Java application and report a 35% success rate among detected refactorings that are definitely correct and a 65% success rate among those that are possibly correct.

Finally, refactorings can be detected in other software artifacts such as design documents. For example, a widely recommended way to detect refactorings is to observe design shortcomings manifested during development and maintenance [5].

6. Conclusion

Program refactoring is the process of applying meaning-preserving changes to a program to improve its structure in order to aid understanding and maintenance. Although some earlier work has focused on detecting refactorings in code, much of the work on refactorings has focused on automating the changes to the source code. We have presented a low-cost, syntactic approach for automatically discovering refactorings in source code. Our approach uses symbol table and reference information together with simple code metrics such as line and statement counts.

To validate our approach, we implement a tool called *Look#* to analyze C# programs. We examined ten C# programs, attempting to find a variety of refactorings. Over three thousand refactorings were discovered across ten programs. We then inspected each suggested refactoring by hand to determine its correctness. We found that over 87% of the refactorings suggested were correct. Most notably, we found that some refactorings have a near-perfect success rate and overall most individual success rates were high. In the validation process, we discovered some interesting facts about the programs itself, in particular that programs often provide a rich set of interface methods for classes, many of which are not used. Additionally, the property mechanism of C# that was designed to transparently encapsulate public fields was not used extensively, and that many programmers do not consider public fields as a violation of encapsulation.

As future work, we plan to analyze more programs and provide a statistical analysis of the refactorings detected. In particular, we have been keeping historical data as additional programs have been added to the test suite to determine the likelihood of success on a new program.

Also, we define success rate as the number of detected refactorings that are correct. Our motivation for focusing on eliminating false positives is that a tool that reports many refactorings only a few of which are correct will simply not be used. However, we should also consider the number of refactorings that the tool may have missed (i.e., false negatives). Obtaining such a result would require a detailed inspection by hand of a number of programs to locate refactorings that the tool missed. We expect that examining the history of a program's changes through source code repositories or change logs will be beneficial in this undertaking.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] R. S. Arnold, editor. *Tutorial on Software Restructuring*. Washington, D.C., 1986.

[3] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, Jan. 1990.

[4] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, 1999.

[5] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[6] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Methodol.*, 2 (3):228–269, July 1993.

[7] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *Proc. 2001 Int. Conf. on Softw. Maint.*, pages 736–743, Florence, Italy, Nov. 2001.

[8] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, Feb. 2004.

[9] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving program slicing using dynamic points-to data. In *Proc. 10th ACM Symp. on Found. Softw. Eng.*, pages 71–80, Charleston, SC, Nov. 2002.

[10] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, Strategic Planning and Economic Analysis Group, Gaithersburg, MD, May 2002.

[11] W. F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1992.

[12] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Softw. – Pract. Exp.*, 25(7):789–810, July 1995.

[13] F. Simon, F. Steinbrücker, and C. Lewerentz. Metrics based refactoring. In *Proc. 5th Eur. Conf. on Softw. Maint. Reeng.*, pages 30–38, Lisbon, Portugal, Mar. 2001.

[14] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. *Autom. Softw. Eng.*, 8(1):89–120, 2001.

[15] T. Tourwé and T. Mens. Identifying refactoring opportunities using logic metc programming. In *Proc. 7th Eur. Conf. on Softw. Maint. Reeng.*, pages 91–100, Benevento, Italy, Mar. 2003.

[16] M. Vittek. Refactoring browser with preprocessor. In *Proc. 7th Eur. Conf. on Softw. Maint. Reeng.*, pages 101–110, Benevento, Italy, Mar. 2003.

[17] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, SE-10 (4):352–357, July 1984.