SANTA CLARA UNIVERSITY
Department of Computer Engineering

Date: ______________

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Todd J.H. King

ENTITLED

Finding Refactorings Using Lightweight Code Analysis

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

OF

MASTER OF SCIENCE IN COMPUTER ENGINEERING

____
Thesis Advisor

____
Thesis Reader

____
Chairman of Department

FINDING REFACTORINGS USING LIGHTWEIGHT CODE ANALYSIS

BY

TODD J.H. KING

Submitted in Partial Fulfillment of the Requirements
for the degree of
Master of Science in Computer Engineering
School of Engineering
Santa Clara University

Santa Clara, California
December, 2005

FINDING REFACTORINGS USING LIGHTWEIGHT CODE ANALYSIS

Todd J.H. King

Department of Computer Engineering
Santa Clara University
2005

ABSTRACT

Poorly structured code is hard to maintain and read. Refactoring can improve the code structure and thus make it easier to preserve and to discern the underlying design. According to Martin Fowler's book *Refactoring*, 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." Refactoring is a difficult and time consuming process which makes it an unattractive activity for many developers. A tool that could automatically identify refactorable code sections and make fix suggestions would make refactoring much easier and faster. In my research I have developed the tool *Look#* for just this purpose.

The *Look#* tool uses lightweight detection algorithms based on code metrics to identify refactorable code sections. Each refactoring technique that *Look#* can detect has been distilled down into a lightweight algorithm based on thresholds and syntactic keywords. Some refactoring techniques are detected by calculating code metrics and then evaluating if those code metric counts exceed a certain threshold. Other refactoring techniques are detected by looking for certain syntactic indicators, like a `public` access modifier on a field.

Many of the guidelines Martin Fowler provides for determining when to apply certain refactoring methods are based on "human intuition" and abstract perceptions. This makes it extremely difficult to automatically identify where many of the refactoring methods should be applied. Therefore the goal of these refactoring algorithms is not to definitively find all the refactorings in the users code, but to use metrics that look for common tell-tale signs of refactorings to help point "informed human intuition" in the right direction.

The primary barrier to automatic refactoring detection is that it requires a good understanding of the code. In the absence of a good human understanding of the code, the best a detection algorithm can do is look for common tell-tale signs that most likely indicate the presence of refactorable code. This thesis explores using code metrics to look for these tell-tale signs and how successful this approach is. In most cases, the refactoring detection algorithms discussed in this thesis had a high success rate. Some of the less successful algorithms need to be reworked and made smarter.

Table of Contents

1 Introduction **1**
- 1.1 The Importance of Software . . . 1
- 1.2 The Software Maintenance Problem . . . 2
- 1.3 What is Refactoring? . . . 3
- 1.4 Refactoring Tools . . . 4
- 1.5 The Look# Tool . . . 5

2 Refactoring Background **7**
- 2.1 Object-Oriented Languages . . . 7
- 2.2 The C# Language . . . 8
- 2.3 Refactoring Techniques . . . 10
  - 2.3.1 Encapsulate Field . . . 10
  - 2.3.2 Remove Empty Method . . . 10
  - 2.3.3 Remove Unused Field . . . 11
  - 2.3.4 Remove Unused Method . . . 11
  - 2.3.5 Hide Method . . . 11
  - 2.3.6 Extract Class . . . 12
  - 2.3.7 Extract Method . . . 12
  - 2.3.8 Decompose Conditional . . . 13
  - 2.3.9 Replace Magic Number . . . 14
  - 2.3.10 Move Field . . . 14
  - 2.3.11 Move Method . . . 15
  - 2.3.12 Remove Parameter . . . 15
  - 2.3.13 Pull Up Field . . . 15
  - 2.3.14 Pull Up Method . . . 16
  - 2.3.15 Extract Superclass . . . 16
  - 2.3.16 Push Down Field . . . 16
  - 2.3.17 Push Down Method . . . 17
  - 2.3.18 Extract Subclass . . . 17
  - 2.3.19 Substitute Algorithm . . . 17
  - 2.3.20 Split Temporary Variable . . . 18

3 Look# Architecture **19**
- 3.1 Overview . . . 19
- 3.2 AST Construction . . . 21
- 3.3 Symbol Table Construction . . . 24
- 3.4 Refactoring Detection . . . 25

4 Refactoring Detection Algorithms **28**
4.1 Overview 28
4.2 Detecting refactorings 29
4.2.1 Encapsulate Field 29
4.2.2 Remove Empty Method 31
4.2.3 Remove Unused Field 31
4.2.4 Remove Unused Method 31
4.2.5 Hide Method 32
4.2.6 Extract Class 32
4.2.7 Extract Method 33
4.2.8 Decompose Conditional 33
4.2.9 Replace Magic Number 34
4.2.10 Move Field 35
4.2.11 Move Method 36
4.2.12 Remove Parameter 37
4.2.13 Pull Up Field 38
4.2.14 Pull Up Method 38
4.2.15 Extract Superclass 39
4.2.16 Push Down Field 39
4.2.17 Push Down Method 40
4.2.18 Extract Subclass 41

5 Experimental Results **42**
5.1 Overview 42
5.2 Refactoring Detection Results 43
5.2.1 Encapsulate Field 43
5.2.2 Remove Empty Method 47
5.2.3 Remove Unused Field 47
5.2.4 Remove Unused Method 47
5.2.5 Hide Method 48
5.2.6 Extract Class 48
5.2.7 Extract Method 48
5.2.8 Decompose Conditional 49
5.2.9 Replace Magic Number 49
5.2.10 Move Field 49
5.2.11 Move Method 50
5.2.12 Remove Parameter 50
5.2.13 Pull Up Field 51
5.2.14 Pull Up Method 51
5.2.15 Extract Superclass 51
5.2.16 Push Down Field 51
5.2.17 Push Down Method 52
5.2.18 Extract Subclass 52
5.3 Error Analysis 52
5.4 Reliability of the Results 54
5.5 False Negative Analysis 58
5.6 Conclusion 60

6 Related Work **62**

7 Conclusion **64**

List of Figures

2.1 (a) public field usage, (b) public property usage. 9
2.2 Part (a) is an example of how properties can check input values. Part (b) is an example of how properties can be read-only or write-only. 10
2.3 Before (a) and after (b) *Encapsulate Field* is applied. 11
2.4 Example of an empty method that should be removed. 11
2.5 Before (a) and after (b) *Extract Class* is applied. 12
2.6 Before *Extract Method* is applied. 13
2.7 After *Extract Method* is applied. 14
2.8 Before (a) and after (b) *Decompose Conditional* is applied. 14
2.9 Before (a) and after (b) *Replace Magic Number* is applied. 14
2.10 In this example the parameter `size` is no longer used so it should be removed. Before (a) and after (b) *Remove Parameter* is applied. 15
2.11 Before (a) and after (b) *Pull Up Field* is applied. 16
2.12 Before (a) and after (b) *Pull Up Method* is applied. 16
2.13 Before (a) and after (b) *Push Down Field* is applied. 17
2.14 Before (a) and after (b) *Substitute Algorithm* is applied. 18
2.15 Before (a) and after (b) *Split Temporary Variable* is applied. 18

3.1 Pipe & filter style overview of *Look#* architecture. 20
3.2 High level class view of the *Look#* architecture. 20
3.3 The AST node hierarchy. 22
3.4 Reference example. Line 2 declares myvar, and line 3 & 4 reference it. 23
3.5 Class diagram of the refactoring engine. 26

4.1 *Encapsulate Field* algorithm. 29
4.2 *Remove Empty Method* algorithm. 31
4.3 *Remove Unused Field* algorithm. 31
4.4 *Remove Unused Method* algorithm. 32
4.5 *Hide Method* algorithm. 32
4.6 *Extract Class* algorithm. 33
4.7 *Extract Method* algorithm. 33
4.8 *Decompose Conditional* algorithm. 34
4.9 *Replace Magic Number* algorithm. 35
4.10 *Move Field* algorithm. 36
4.11 *Move Method* algorithm. 37
4.12 *Remove Parameter* algorithm. 37
4.13 *Pull Up Field* algorithm. 38
4.14 *Pull Up Method* algorithm. 39
4.15 *Push Down Field* algorithm. 40
4.16 *Push Down Method* algorithm. 41

List of Tables

4.1 Pseudo code commands key. 30

5.1 Description of programs used in the experiments. 42
5.2 Summary of the time required to analyze the test case programs. 43
5.3 Number of refactorings detected for each program. 44
5.4 Success rates of refactorings detected for each program. 45
5.5 Summary of refactoring detections. 46
5.6 Error types table. 53
5.7 Confidence interval of the first ten programs compared to the initial results of last two programs. 55
5.8 The initial results for NDoc and NUnit. 56
5.9 Comparison of the confidence interval of the first ten programs and the final analysis with all twelve programs. 58
5.10 GmailerXP false negative analysis. 59

Chapter 1

Introduction

1.1 The Importance of Software

Software is becoming an increasingly important part of today's world. We rely on software to help fly our planes, connect our phones, control our traffic lights, and run our ATMs. Software has become so prolific that it affects even the the most basic everyday activities. When you take a shower, the water heater has software that ensures you have enough water at the right temperature. When you drive to work, software is constantly monitoring the fuel and air mixture in your engine to ensure smooth operation.

All of these are examples of embedded software, in other words software that does not run on your home PC but on a chip embedded in a device. This kind of software is typically very thoroughly tested before it is released because once it is embedded in the device, updating it is very difficult. If there is a bug in your water heater's software, you cannot simply download an update from the Internet as you would for a PC program. For that reason, it is especially important for embedded software to be thoroughly debugged and tested before being released. This is particularly important for health and safety software like pacemakers and anti-lock brake systems.

However, the software's life cycle does not end there. The same software will probably be used for future products, so the software needs to be maintained and updated. As this process of updating and maintaining the software continues, it becomes harder and more costly to do so each product cycle. In fact a recent study estimated that eighty percent of all software development cost is applied toward maintaining software [National Institute of Standards and Technology, 2002].

Every year software is becoming even more important in all aspects of our lives. Improved software

engineering techniques are needed to keep up with the software's increasing demand while still ensuring it is reliable. One such software engineering technique is refactoring. Refactoring is a form of preventative maintenance that helps reduce maintenance costs.

1.2 The Software Maintenance Problem

Why does software require maintenance? Hardware systems consist of parts that wear out and need to be fixed or replaced. However software has no parts that can wear out, "*whiledo* constructs do not wear out after 10,000 loops, and semicolons do not fall off the ends of statements" [Pfleeger, p. 464]. Software systems differ from hardware because they are designed to be updateable and flexible. 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 initial phases of a typical software development process are analysis and design. During these phases the developers work very hard to create a solid program design that fulfills all the requirements while remaining flexible. Unfortunately, no matter how good the design is, inevitably unexpected problems occur and workarounds are found. Over time the program's code is modified and rearranged. The code gradually deteriorates in this manner away from the originally conceived design structure. This makes code maintenance and further development even more difficult. By the time the program is finished and working, the developers could probably create a much better design in hindsight. "In most projects, the first system built is barely usable. It may be too slow, too big, awkward to use, or all three" [Brooks, 2003, p. 116].

A program is shipped as soon as it works. "Working" typically means it fulfills the requirements and has no serious bugs. However "it's possible to write ugly, obfuscated, horribly convoluted, and undocumented code that still manages to work" [Ganssle, 2003]. To make matters worse, often the developers assigned to maintain a program are not the same ones who originally created it. There is always a large learning curve for a developer who is new to a software system. How large that learning curve is depends on the size of the system and the clarity of the code. It is not uncommon for a new developer in this situation who does not completely understand the system to introduce many new bugs for each bug he or she fixes. There is a very expensive long-term cost for sloppy programming. "As systems gain in complexity, the old hack-and-ship approaches fail utterly and completely and, occasionally, spectacularly" [Douglass, 2002, p. xxiii].

To avoid this downward spiral of confusing code and higher maintenance costs, preventative maintenance is used. Preventative maintenance is the practice of improving a program's code to prevent future bugs from appearing and to make future debugging easier. Essentially, preventative maintenance is not fixing bugs or adding features; it is maintaining the structure of the code itself. Time spent performing preventative maintenance will often save even more time during debugging and future maintenance.

1.3 What is Refactoring?

Refactoring is one form of preventative maintenance. Poorly structured code is hard to maintain and read. Refactoring can improve the code structure and thus make it easier to preserve and to discern the underlying design. 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" [Fowler, 1999, p. 53].

According to the old adage "If it ain't broken, don't fix it," poorly structured code should not be restructured as long as it works. However there are many benefits to restructuring code. Refactoring improves the readability of code and thus improves the ease of code maintenance. Refactoring forces one to carefully examine code and helps the developer achieve a new understanding of the code, and this better understanding of the code makes identifying bugs much easier. Once the developer has improved the readability of the code and has a greater understanding of exactly what it is doing, it is easier for the developer to debug the code and add features. Ideally, refactoring could decrease the development and maintenance time for programming projects. If developers took the time to refactor their code as they implement, they could save themselves much trouble in the long run.

Refactoring is essentially rearranging a program's code so it makes more sense. Refactoring is best explained with a few examples. Say you are working on an employee information and billing program. Every employee is represented by the same class; however, due to how the billing works each type of employee has a different billing scheme. The first thing one might do to refactor this code is extract a class hierarchy out of this one large, monolithic employee class. Then you can move the specialized billing code into the subclasses. Other specialized attributes can be moved down into the subclasses as well. What we have just done has not changed the functionality of our code at all. The new code and old code should function in the exact same manner. The difference is that now the code is much easier to understand. Instead of one large, ungainly billing function filled with special cases, it is now broken up into many specialized billing functions. Now attributes that only pertain to certain types of employees are only present on that employee type class, instead

of being common to all employee classes. This new code structure makes adding and removing features much easier, enables testers to better isolate bugs, and improves readability.

So why do many developers choose not to refactor their code? Unfortunately, refactoring is also a difficult and time consuming process which makes it an unattractive option for many developers. While the above scenario was a simple example, it still would have taken a bit of time to create the class hierarchy, move all the code and comments around, and test it to make sure it still works exactly as it used to. Unfortunately, real world problems are never as simple as the examples, and they often take much longer to solve. At the end of this process, no new features have been added and no new requirements have been met. You have essentially just spent all your time making the program do something it was already doing correctly, but through a different means. Therefore, it can be difficult to justify the short term cost of refactoring.

1.4 Refactoring Tools

To encourage refactoring, tools that help identify refactorings and implement them are needed. These tools would reduce the time it takes to refactor code, making refactoring a much more doable option for developers on tight schedules.

Many excellent refactoring tools already exist such as *IntelliJ IDEA*, *JRefactory*, and *RefactorIT*. However, all of these tools are for the Java programming language, which seems to be the language of choice for refactoring. Microsoft has recently published a new programming language as part of its .NET technology called C#. C# is "a simple, modern, object oriented, and type-safe programming language derived from C and C++ "[Archer and Whitechapel, 2002, p. xix]. C# bears a very close resemblance to Java, so existing Java refactoring methods can easily be applied to C#. Nevertheless, because C# is a new language, very few refactoring tools exist for it. Two popular refactoring tools for C# are *C# Refactory* and *C# Refactoring Tool*. These tools allow the user to select a portion of code and perform an automated refactoring technique. Unfortunately, these tools do not tell the user where these refactoring methods should be applied.

As of this writing, no tool exists that does a very good job of identifying where refactoring techniques should be applied in C# code. The *dotEasy* code metric tool comes close to providing this functionality, however it is more focused on computing code metrics than applying refactoring techniques. Simon et al. have made an interesting tool that displays the coupling between class elements in a 3D space using a distance metric (2001). However their tool still leaves it up to the users to draw their own conclusions about what needs refactoring from the 3D view rather than explicitly identifying locations where refactorings should be applied.

Their tool also does not scale well for large projects, since it is only designed to compare two or three classes at a time. Tourwé and Mens made a tool called *SOUL* that uses logic meta programming to find refactorings in code (2003). Their tool appears to work quite well, however it can currently only identify two refactoring techniques. It is also difficult to tell how much testing they have done on their tool. As you can see in Chapter 5 different programs can yield quite different results.

1.5 The Look# Tool

A tool that can identify refactorable code sections and make fix suggestions would be a very valuable aid to the refactoring process. I believe such a tool could greatly improve one's code if used properly during development. This tool would allow developers to identify design structure problems early on and fix them quickly before they grow into much larger problems. In my research I have developed the tool *Look#* for just this purpose.

Tools that analyze source code often use static analysis, also know as data-flow analysis, to get more detailed information about the code. Data-flow analysis is "a process of collecting information about the way variables are used in a program"[Aho et al., 1986, p. 586]. Such tools analyze the program source, computing information about the program such as data and control dependences. 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.

The *Look#* tool can detect refactorings without performing data-flow analysis. It simply parses a C# project's source code and constructs an abstract syntax tree (AST) and symbol table. From there it is able to scan the AST and symbol table for the information it needs to compute the metrics it needs to determine where refactoring techniques should be applied. To detect refactorings, I have distilled many refactoring techniques down to concrete algorithms based on thresholds using code metrics. I found that the principle barrier to increasing the algorithms' accuracy was the increasingly complex input the improved algorithms required. Conceptually, many of the refactoring methods can be translated into a simple high-level algorithm, however that simple algorithm almost always requires very complex inputs and has many unforeseen exceptions. The key issue is refactoring requires a good understanding of the code. So to make accurate refactoring

identification algorithms, inputs that reflect that good understanding are required. However that kind of data is very difficult to automatically calculate.

Due to the abstract nature of the original refactoring techniques, the accuracy of my algorithms is very subjective. In my own tests, I have judged my tool to be accurate the majority of the time. I have also found that some refactorings techniques are so simple they can be detected with one hundred percent accuracy. In other cases I have found that there are some refactoring techniques that a computer will never be able to detect with one hundred percent accuracy, but many have tell-tale indicators the *Look#* tool can look for. As *Look#* is designed to aid developers in finding sections of code that can be refactored, *Look#* has been designed to minimize false positives, so ideally all the refactorings reported are accurate. The consequence of this decision is there may be many valid refactorings that are ignored by the tool. In order to maximize the tool's effectiveness for the developer, I decided that presenting the developer with only valid refactorings would increase the tool's effectiveness since the developer will not have to waste time sifting through false positives to find a reported refactoring that is valid.

Chapter 2

Refactoring Background

2.1 Object-Oriented Languages

Refactoring is targeted at object-oriented languages such as Java, C++, C#, and others. Object-oriented languages have three main principles: data abstraction, inheritance, and dynamic method binding [Sebesta, 2002, p. 22].

Data abstraction or data hiding, is the principle that objects should hide their data from other objects. Objects can grant access to their data through accessor methods. In this way, objects can control how their data is manipulated, if it is read only, write only, or read-write. Objects can also make sure their data is never put in an invalid state, by verifying the data is within bounds.

Inheritance is a mechanism that allows types to form type hierarchies. In such hierarchies, inheritable attributes in the parent class are inherited by the child class. If the parent class `Array` declares the method `Count()`, then the child class `SortedArray` will inherit `Count()`. Most object-oriented languages have mechanisms known as "access modifiers" that determine whether a type attribute is inheritable. Using access modifiers, a type can specify which attributes its child classes will inherit. Child classes also can override (re-define) methods they inherit from their parents, if they need to change the method's functionality. So even though both parent and child classes may define the same method, the method's implementation can be different.

Object-oriented languages allow instances of a child class to be used anywhere an instance of the parent class type is expected. This is known as polymorphism. Even though a child class instance can be used where a parent class type is expected, only the parent class's interface is visible, so only attributes defined by the

parent class can be used. However if any parent class attributes are overridden by the child class, then the child class's overridden attribute is used instead. This is called dynamic binding.

Most refactoring methods are designed to take advantage of these aspects of object-oriented languages. There is a whole family of refactoring techniques dedicated to inheritance hierarchies. Many other refactorings are based on the principle of data abstraction, that certain data belongs to certain classes and only those classes should manipulate that data. Some refactorings are independent of object-oriented languages however, as they focus on algorithms and code fragment problems that are common among most imperative languages. So refactoring is not exclusively for object-oriented languages, but it is certainly tailored for them.

2.2 The C# Language

In June 2000, Microsoft released a new strongly-typed object-oriented language called C# along with their new .NET platform. Essentially C# seems to be a hybrid language based on C++ and Java. The C# language bears an especially close resemblance to Java. It is no surprise that Java and C# are remarkably similar, they were both created for the same reason, to provide a simpler and more reliable alternative to the C++ language. C# is focused on increasing programming efficiency and reliability by freeing the developer from the responsibility of performing memory management, type checking, index range checking, etc.

There are several ways in which C# is more developer friendly than previous object-oriented languages. For one, C# has a rooted class hierarchy where all classes are subclasses of the object class. This means that any type instance can be used where an Object type is expected since all types inherit from Object. Even the primitive types in C# inherit from the Object type.

Multiple inheritance in C++ can be the source of much confusion and frustration if used improperly. Therefore C# follows Java's lead and only supports single inheritance of classes but allows a type to inherit from multiple interfaces. An interface is a type that allows you to specify attribute declarations, like methods, but contains no implementation. This avoids the problem of multiple inheritance where two different implementations can be inherited.

C# methods are non-virtual by default. To override a method in C# the "virtual" keyword must be used in the original declaration, and the "override" keyword must be used in the overriding declaration. Also, C# does not allow global methods, all methods must be declared on a type such as a class or struct. By default C# passes arguments by value, however C# allows an argument to also be passed by reference through the use of the "ref" and "out" keywords. While passing by value is much safer, passing by reference allows a

```
public class Person
{
  public string name;
}

...
  Person person = new Person();
  person.name = "Todd";
  string temp = person.name;
...
```

(a)

```
public class Person
{
  private string name;
  public string Name
  {
    get{return name;}
    set{name = value;}
  }
}
...
  Person person = new Person();
  person.Name = "Todd";
  string temp = person.Name;
...
```

(b)

Figure 2.1: (a) public field usage, (b) public property usage.

method call the return multiple objects by changing the values of its arguments.

One of the goals of C# was to provide a safer and more reliable alternative to C++. C# did this by removing all of the unsafe features of C++ by default, such as pass by reference, pointers, and unchecked bounds. For example, C# demands that a variable be initialized before it is used. However C# allows programmers to still use some of these unsafe features but only through the use of special keywords that confirm the developer knows what he or she is doing.

C# has a specialized mechanism for accessing data called "properties". A property, also known as a "smart field", allows accessors to be used the same way you would use a public field [Archer and Whitechapel, 2002, p. 143]. Syntactically there is no difference between accessing a public field and a public property. This is extremely convenient because it allows a developer to use the property exactly as one would use a public field as you can see in figure 2.1. This has the added benefit of allowing a developer to switch from a public field to a public property without changing everywhere the field was accessed. The reason for providing data accessors in the first place is so data checking can be performed to make sure the data is never given an invalid value, or accessed while it is in an inconsistent state. Also properties do not have to be read and write. A property can also be read only or write only by only defining a `get` or `set`. Now that C# provides this property mechanism, there is no reason for developers not to practice good data abstraction by providing indirect access to fields through properties.

```
public class Data {
  private double oxygenRatio;
  public double OxygenRatio
  {
    get{return oxygenRatio;}
    set{
      if(value>=0.00 && value<=1.00)
        oxygenRatio = value;
    }
  }
  ...
}
```

(a)

```
public class Expression
{
  private string input;
  public string Input
  {
    set{input = value;}
  }
  private double result;
  public double Result;
  {
    get{return result;}
  }
  ...
}
```

(b)

Figure 2.2: Part (a) is an example of how properties can check input values. Part (b) is an example of how properties can be read-only or write-only.

2.3 Refactoring Techniques

2.3.1 Encapsulate Field

The object-oriented programming style dictates that fields should not be public. “One of the principal tenets of object orientation is encapsulation, or data hiding” [Fowler, 1999, p. 206]. When an object’s data is exposed foreign objects can modify the data in illegal ways. Therefore, public fields should always be made non-public. Objects can provide accessor and modifier methods so foreign objects can still access the data indirectly. Of all the languages where this principle should be followed, none are better suited than C#. C#’s property attribute makes applying *encapsulate field* extremely painless. Since a C# property can be treated just like a public field, to apply *encapsulate field* you simply make a public property with the same name as the public field, rename the field slightly (so the object does not have two attributes with the same name), and make the field non-public.

2.3.2 Remove Empty Method

If a method has nothing in its body, then it most likely serves no purpose. Something like this might happen if a developer declares a method and intends to implement it, but ends up not needing to and forgetting about it. Useless methods such as these clutter the object’s interface and source code. Empty methods should be removed unless they do somehow serve a purpose.

```
public class Name
{
  public string FirstName;
  public string LastName;
}
```

(a)

```
public class Name
{
  private string firstName;
  private string lastName;
  public string FirstName
  {
    get{return firstName;}
    set{firstName = value;}
  }
  public string LastName
  {
    get{return lastName;}
    set{lastName = value;}
  }
}
```

(b)

Figure 2.3: Before (a) and after (b) *Encapsulate Field* is applied.

```
public void DoNothing()
{}
```

Figure 2.4: Example of an empty method that should be removed.

2.3.3 Remove Unused Field

If a field is never used, then it is probably dead code. This often happens when a field is replaced with a different one, but the developer forgets to remove the original field. If the field is indeed never used, and it serves no purpose or will not be used in the future, then it should be removed.

2.3.4 Remove Unused Method

If a method is never called, then it is probably dead code. This often happens when a method is replaced with a different one, but the developer forgets to remove the original method. If the method is indeed never used, and it serves no purpose or will not be used in the future, then it should be removed.

2.3.5 Hide Method

If a public method is only called within its own class, then that method should be made non-public. An object's public interface should only contain methods that foreign classes actually want to use. Public methods that are only used by the declaring object needlessly clutter its public interface.

2.3.6 Extract Class

Extract class is used to break up a large class that is trying to do too many different tasks. Reducing a large class down into multiple smaller, more modular classes improves readability and reusability. Classes that should be broken up typically have many data fields, methods, and responsibilities. To split such a class, identify and extract subsets of data and methods that logically go together into a new class.

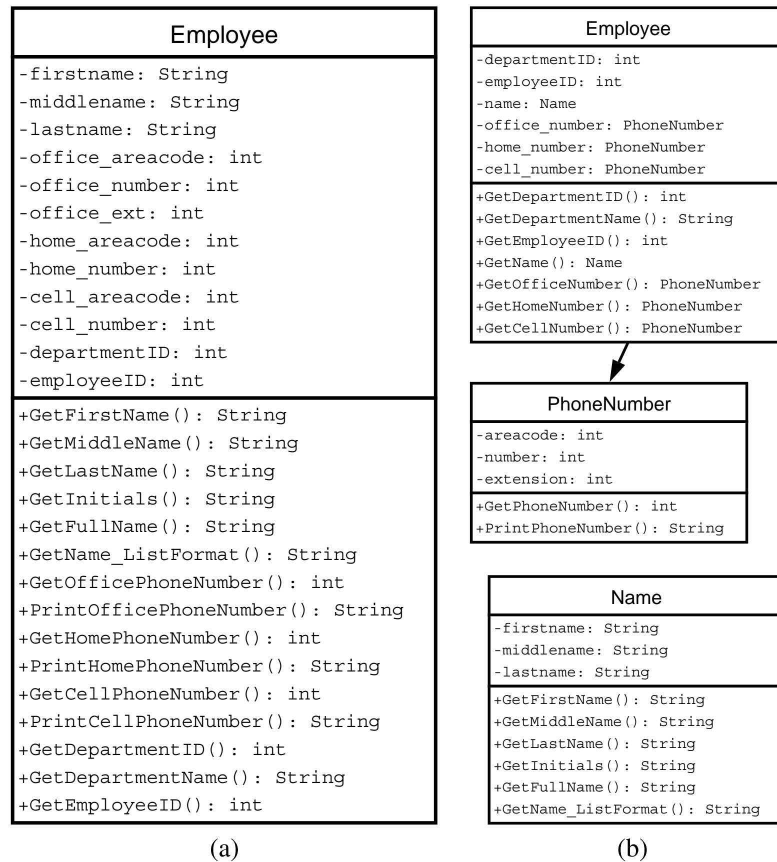

Figure 2.5: Before (a) and after (b) *Extract Class* is applied.

2.3.7 Extract Method

Extract method is used to break up a large and or confusing methods into smaller methods to increase readability and reusability. The idea behind *extract method* is that objects with many small, well-named methods are much clearer and easier to understand than objects with a small number of very large methods. Often, large methods will have reoccurring code fragments or very cohesive and independent code fragments that could be moved outside of the large method. That is what *extract method* is all about, moving those reoc-

```
private bool CompareObjects(object obj1, object obj2, CompareOp compOp){
  switch (compOp){
    case CompareOp.Equal:
      if (obj1 is string && obj2 is string)
        return obj1.Equals(obj2);
      else if (obj1 is bool && obj2 is bool)
        return obj1.Equals(obj2);
      else if (obj1 is int && obj2 is int)
        return obj1.Equals(obj2);
      else if (obj1 is int && obj2 is double)
        return (Convert.ToDouble(obj1)).Equals(obj2);
      else if (obj1 is double && obj2 is double)
        return obj1.Equals(obj2);
      else if (obj1 is double && obj2 is int)
        return obj1.Equals(Convert.ToDouble(obj2));
      else
        throw new ApplicationException("objects not comparable!");

    case CompareOp.NotEqual:
      if (obj1 is string && obj2 is string)
        return !obj1.Equals(obj2);
      else if (obj1 is bool && obj2 is bool)
        return !obj1.Equals(obj2);
      else if (obj1 is int && obj2 is int)
        return !obj1.Equals(obj2);
      else if (obj1 is int && obj2 is double)
        return !(Convert.ToDouble(obj1)).Equals(obj2);
      else if (obj1 is double && obj2 is double)
        return !obj1.Equals(obj2);
      else if (obj1 is double && obj2 is int)
        return !obj1.Equals(Convert.ToDouble(obj2));
      else
        throw new ApplicationException("objects not comparable!");
    }
    return false;
}
```

Figure 2.6: Before *Extract Method* is applied.

curring and or independent code fragments into their own methods. Doing this makes the original method clearer since some of its code fragments have been replaced with well named method calls. See figure 2.6 and 2.7.

2.3.8 Decompose Conditional

Large complicated conditional statements are often times hard to read and understand. Sometimes these confusing conditionals are accompanied by comments explaining them, however making the conditional simple and readable is an even better solution. To perform *decompose conditional*, extract the condition in the conditional statement into its own well-named method, and simply evaluate the result of that well-named method in the conditional statement.

```
public bool CompareObjects(object obj1, object obj2, CompareOp compOp){
  switch (compOp){
    case CompareOp.Equal:
      return AreEqual(obj1, obj2);

    case CompareOp.NotEqual:
      return !AreEqual(obj1, obj2);
    }
    return false;
}
```

Figure 2.7: After *Extract Method* is applied.

```
if( angle > 6.4 && angle < 15.7 &&
    velocity < 200.2 &&
    velocity > 176.34 &&
    heading > 160 && heading < 200)
{ ... }
```

(a)

```
if( IsReEntrySafe() )
{ ... }
```

(b)

Figure 2.8: Before (a) and after (b) *Decompose Conditional* is applied.

2.3.9 Replace Magic Number

A magic number is a number with a special value that is usually not obvious [Fowler, 1999, p. 204]. A good example of a magic number is the gravity constant (9.81) or π (3.14). Universal mathematical constants are not the only magic numbers, any number with a special significance is a magic number. Other common examples are numeric thresholds as seen in figure 2.8. In that figure the angle, velocity, and heading all must fall within certain thresholds. Those thresholds are magic numbers with special significance that should be made into symbolic constants. Array size values are another common source of magic numbers.

2.3.10 Move Field

A field should be moved to a foreign class when the foreign class uses that field more than the class that declares it. This situation is particularly common with data classes. A class that simply stores data for other classes and does not make use of its own data is a data class. *Move field* is also frequently used when applying *extract class* to help move fields around to the class they belong in.

```
if( size >= 5 && size <= 54)
{ ... }

int[] scale = new int[43];
for(int i=0; i<43; i++)
  scale[i] = 0;
```

(a)

```
if( size >= MINSIZE && size <= MAXSIZE)
{ ... }

int[] scale = new int[SCALESIZE];
for(int i=0; i<SCALESIZE; i++)
  scale[i] = 0;
```

(b)

Figure 2.9: Before (a) and after (b) *Replace Magic Number* is applied.

```
public void initNumArray(int[] nums,
                          int size)
{
  for(int i=0; i<nums.Length; i++)
    nums[i]=0;
}
```

(a)

```
public void initNumArray(int[] nums)
{
  for(int i=0; i<nums.Length; i++)
    nums[i]=0;
}
```

(b)

Figure 2.10: In this example the parameter `size` is no longer used so it should be removed. Before (a) and after (b) *Remove Parameter* is applied.

2.3.11 Move Method

A method should be moved to a foreign class when the foreign class uses that method more than the class that declares it, or the method uses more resources from the foreign class than the method's defining class. *Extract method* is often used with *move method* if a method is using many foreign resources but the whole method can not be moved. In that case, it is often possible to extract the code fragments making use of those foreign resources into a separate method and then move that method to the foreign class.

2.3.12 Remove Parameter

If a method is no longer using a parameter, then that parameter should be removed. Developers are sometimes hesitant to remove parameters because they fear they may need them again. This is a bad idea because everyone who calls that method must make the extra effort to provide the data for that extra parameter [Fowler, 1999, p. 277]. It is also in a developer's best interest to remove useless parameters quickly so they have fewer method calls to correct. There are some exceptions to this rule, such as when the method is implementing an interface and therefore can not change the method signature, or if the method is overridden in a subclass that actually does use the parameter.

2.3.13 Pull Up Field

If all or most of a class's subclasses have the same field, then it should be pulled up into the parent class. This way it is only defined once in a central location instead of being defined multiple times. If many of the subclasses have the same field but it should not be pulled up into the parent class, then *extract superclass* can be used. See figure 2.11.

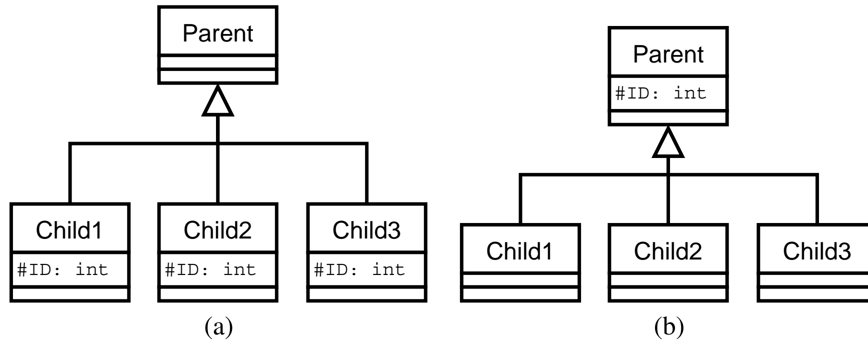

Figure 2.11: Before (a) and after (b) *Pull Up Field* is applied.

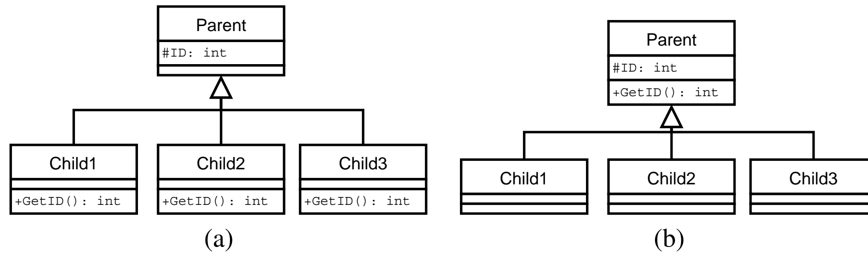

Figure 2.12: Before (a) and after (b) *Pull Up Method* is applied.

2.3.14 Pull Up Method

If all or most of a class's subclasses declare the same method, then it should be pulled up into the parent class. This way it is only defined once in a central location instead of being defined multiple times. If many of the subclasses have the same method but it should not be pulled up into the parent class, then *extract superclass* can be used. See figure 2.12.

2.3.15 Extract Superclass

If multiple classes share some similar attributes, such as fields and methods, then those common attributes should be pulled up into a common superclass. This refactoring can be applied to classes that do or do not share a common class hierarchy. If all the classes share a common superclass already, then *pull up field* or *pull up method* should probably be used.

2.3.16 Push Down Field

When an inheritable field is used by very few of the subclasses that inherit it, and the class that defines the field does not use it, that field should be pushed down into the subclasses that use it. If very few subclasses

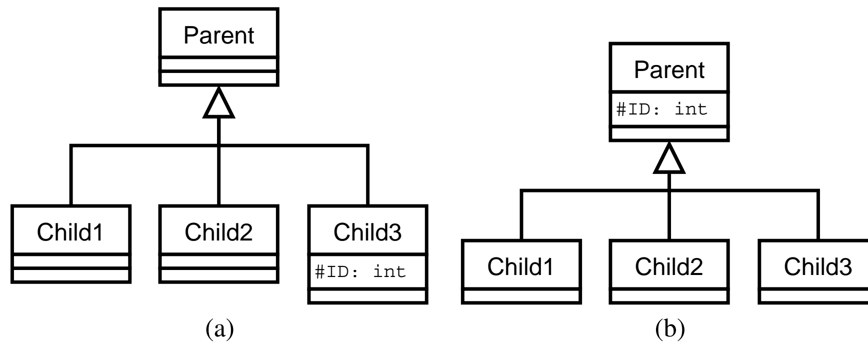

Figure 2.13: Before (a) and after (b) *Push Down Field* is applied.

need a particular field, then there is no sense in forcing all the other subclasses to be larger than they need to be by inheriting this field. The field should be moved out of the superclass, and down into the subclasses that need it. See figure 2.13.

2.3.17 Push Down Method

If an inheritable method is used by very few of the subclasses that inherit it, and the class that defines the method does not use it, that method should be pushed down into the subclasses that use it. Rarely used methods like this clutter the interface of the entire class hierarchy. The method should be moved out of the superclass, and down into the subclasses that actually make use of it.

2.3.18 Extract Subclass

When a subset of a superclass's inheritable attributes are only used by a minority of the superclass's child classes, a subclass should be extracted with that subset of attributes and made the parent of all the child classes that use that subset of attributes. *Extract subclass* can also be applied when a subset of a class's features are only used by some of its instances [Fowler, 1999, p. 330]. If only one subclass uses an inherited parent class attribute, then *push down field* or *push down method* should probably be used.

2.3.19 Substitute Algorithm

Whenever possible, large and confusing algorithms should be replaced with clearer ones. There is always more than one way to get from point A to point B. If at some point you find an easier more direct path, then you should take it. Even if the old confusing algorithm already works properly, simpler is always better. You

```
if(showalias && name.Equals("Mikey"))
  return "Master " + name;
else if(showalias &&
        name.Equals("Andre"))
  return "Big " + name;
else if(showalias &&
        name.Equals("Jimmy"))
  return "Quick " + name;
else
        return name;
```
(a)

```
Hashtable nicknames;
nicknames = MakeNicknamesTable();
if(showalias &&
   nicknames.Contains(name))
{
  return nicknames[name] + name;
}
else
  return name;
```
(b)

Figure 2.14: Before (a) and after (b) *Substitute Algorithm* is applied.

(a)
```
string result;
double temp = 2 * PI * radius * height;
result = temp.ToString() + ", ";
temp = PI * radius^2 * height;
result += temp.ToString();
return result;
```

(b)
```
string result;
double surface_area = 2 * PI * radius * height;
result = surface_area.ToString() + ", ";
double volume = PI * radius^2 * height;
result += volume.ToString();
return result;
```

Figure 2.15: Before (a) and after (b) *Split Temporary Variable* is applied.

will save everyone else who has to work with that code the time and effort it takes to comprehend what that old algorithm was doing. See figure 2.14.

2.3.20 Split Temporary Variable

If you have a temporary variable that is not a loop variable, is assigned multiple values, and each of those values are unrelated, then that temporary variable should be split into multiple variables with names that more accurately reflect what they are being used for. Modern compiler optimization techniques have made it so that using many well-named temporary variables has almost no affect on performance versus code that attempts to reuse the same temporary variables over and over again for different purposes. Therefore, it makes much more sense to improve the readability of your code by using as many well named temporary variables as you need instead of trying to reuse your temporary variables for different tasks. See figure 2.15.

Chapter 3

Look# Architecture

3.1 Overview

Look# has a fairly simple high level control flow. First *Look#* parses a C# project and builds an AST. Then it scans the AST and generates a symbol table. Then the refactoring algorithms use the data in the AST and symbol table to find refactorings. Finally, the list of discovered refactorings is displayed for the user with additional information on how to apply the suggested refactorings. Figure 3.1 shows the *Look#* architecture from a pipe and filter view [Clements et al., 2004, p. 126].

In figure 3.2 you can see a very high-level class view of the *Look#* architecture. There are two major packages, the *LookCS* package contains the backend of the tool where all the work is done. The *LookCSGUI* package contains the Windows GUI for the tool. The GUI and backend were separated to ease adapting *Look#* to a different GUI, for example as a plug-in for Microsoft Visual Studio instead of a stand-alone application. Coupling between the GUI and the backend is minimized because a GUI only needs to know about `AnalysisController` to operate the *Look#* backend. `AnalysisController` provides the methods `StartAnalysis()` and `StopAnalysis()` to start and stop the analysis and the property `RefactoringItems` that returns the refactorings found when the analysis is done. Those three members are all a GUI needs to use the *Look#* backend. `AnalysisController` manages all of the analysis. It first passes `ParserController` the pathname of the file to be parsed, `ParserController` then returns the AST. Then `SymbolTableController` is passed the AST and it returns the symbol table. Lastly, `RefactorController` is passed the AST and symbol table and produces a list of refactorings. `MainForm` can then retrieve that list of refactorings from `AnalysisController` and display them for the user.

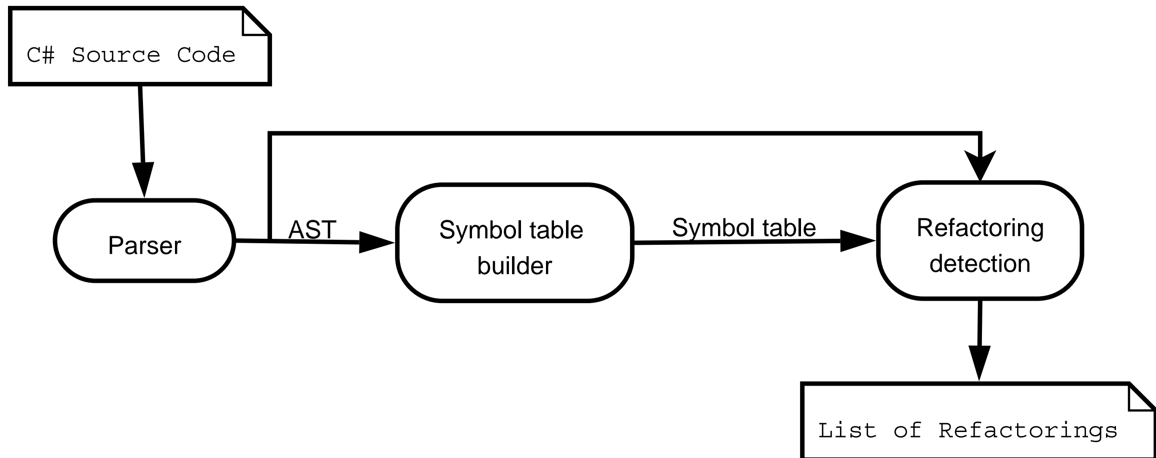

Figure 3.1: Pipe & filter style overview of *Look#* architecture.

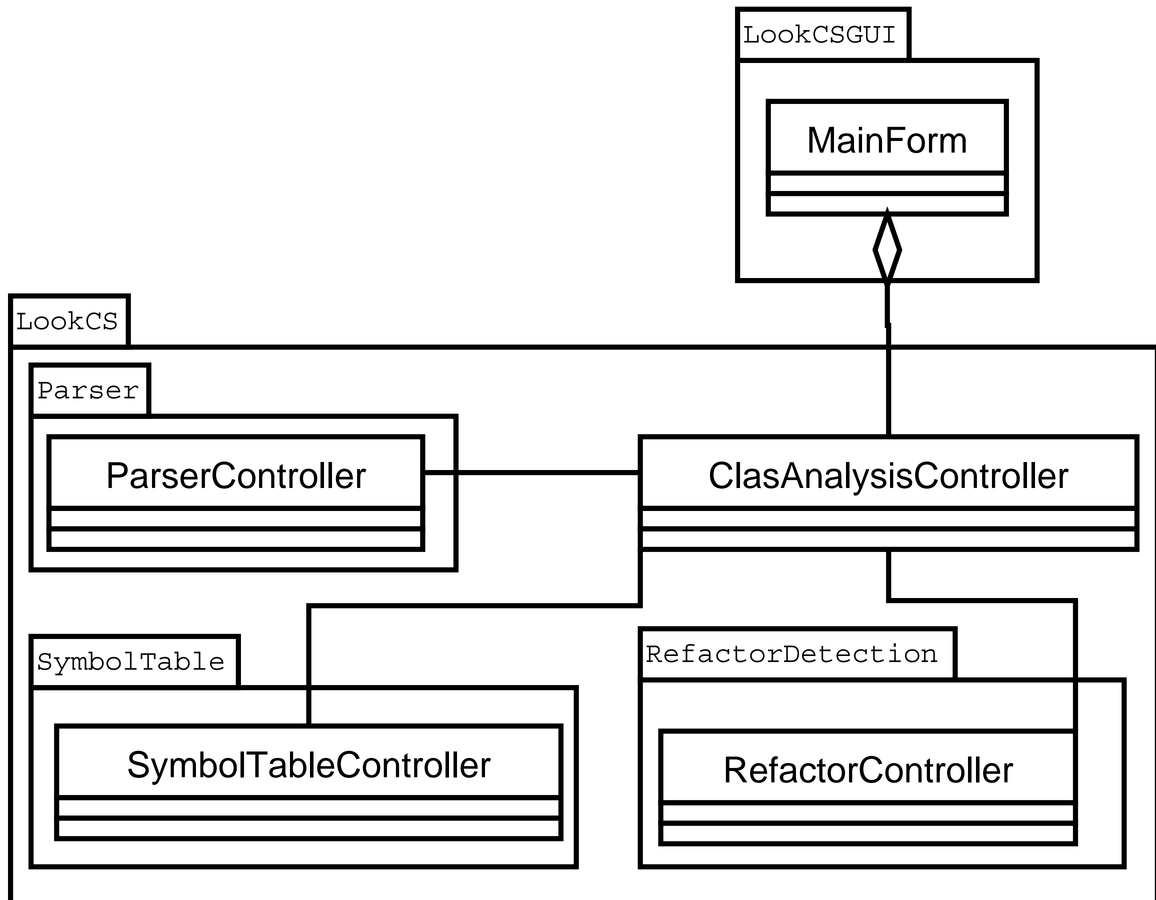

Figure 3.2: High level class view of the *Look#* architecture.

3.2 AST Construction

Look# uses ANTLR (ANother Tool for Language Recognition), formerly known as PCCTS, to generate the parser. ANTLR was chosen as the parser for *Look#* because it automatically generates an AST. `CommonAST` is the default node class ANTLR uses in its AST. Each `CommonAST` node knows its right sibling and the node directly under it, In addition, the `CommonAST` node also stores the name and type of each node.

This setup only allows downward tree traversal. Some refactorings require that more information be recorded, so *Look#*'s AST nodes are a subtype of the `CommonAST`, called `BaseNode`, which includes a `parent` field that references the node's parent node. This allows the AST to be traversed upwards as well.

Many refactorings also make use of code metrics such as the lines of code (LOC) metric. Therefore `BaseNode` also includes a `line` field that records the line number of the token that produced the AST node. Not all AST nodes are associated with a lexical token and therefore do not have line number values. Such nodes are always high level nodes such as class or expression. In those cases, the node performs a left-to-right depth-first search of its child nodes until it finds a node that has a line number and uses that line number as its own. In a similar fashion the last line contained by a node can also be found using a right-to-left depth-first search instead. This allows us to calculate the line number range of a class or method. In order to preserve this line number data, *Look#*'s AST preserves more tokens than would be normally found in an AST. For example, all opening and closing braces are given AST nodes so the line number range calculations for classes and methods are more accurate.

Since the AST is kept in memory the entire time, the AST nodes also store symbol information instead of creating separate symbol objects. Most of the nodes in the AST do not represent symbols, so it would be inefficient to make each node in the AST a symbol node. A symbol node is any node that inherits from the `ISymbolNode` interface. Therefore the AST is a heterogeneous tree with many different classes of nodes, all of them inheriting from `BaseNode`. Figure 3.3 shows the AST node class hierarchy.

The `DeclNode` is a child of `BaseNode` that is used to represent variables. This node is used for all AST nodes that define or declare a variable. It has a `symType` field that stores the variable's type, such as int or string. `DeclNode` also has a `references` field that is an array of all the other `BaseNodes` that reference this variable. As you can see in figure 3.4, line one is where `myvar` is declared, so that node is a `DeclNode`. Line two and three are the places where `myvar` is used so they are referencing the `myvar` declaration on line one. The `myvars` on lines two and three are just represented by `BaseNodes` because they are just references; they do not declare anything.

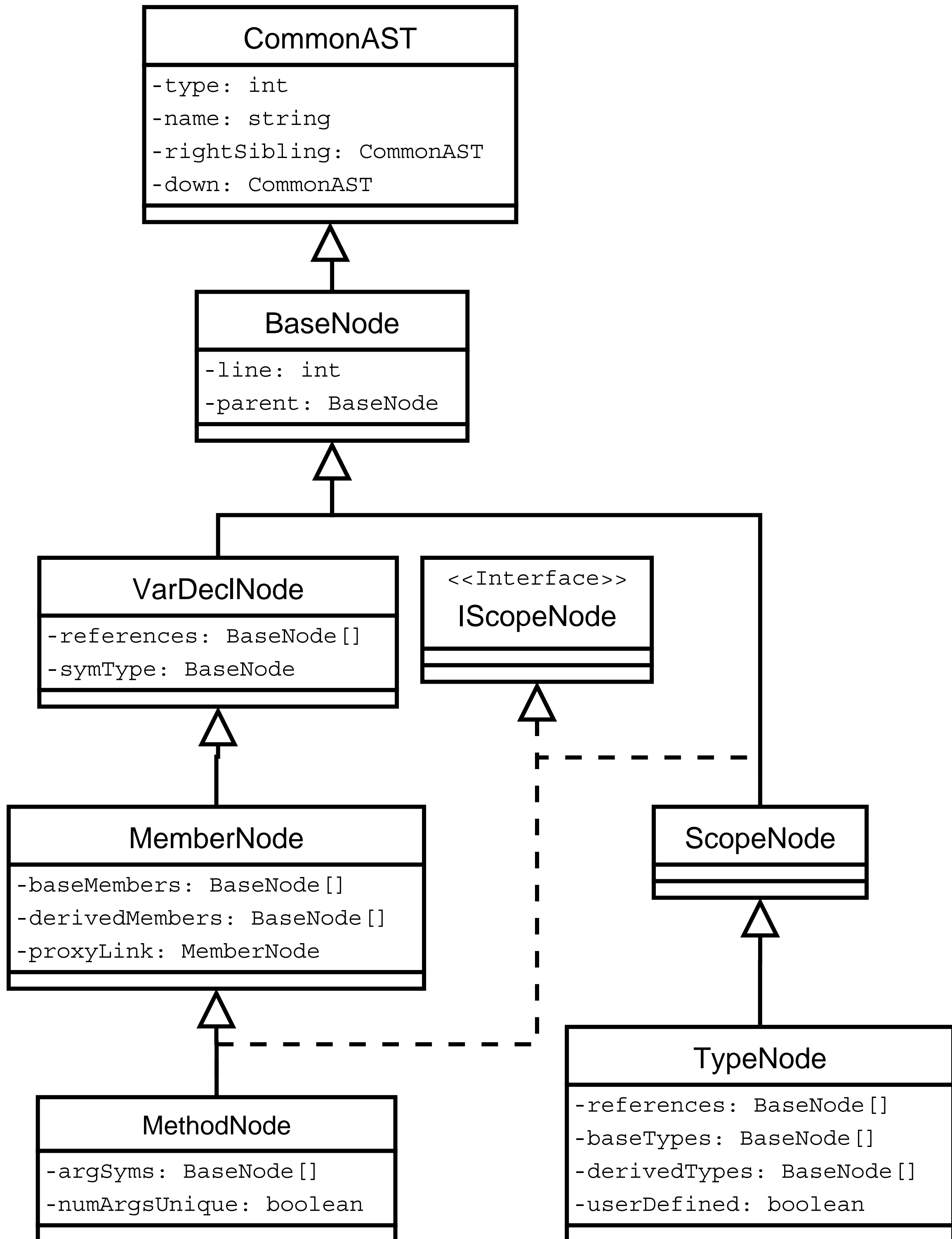

Figure 3.3: The AST node hierarchy.

```
1  public int MyMethod(){
2    int myvar;
3    myvar = 4;
4    return myvar;
5  }
```

Figure 3.4: Reference example. Line 2 declares myvar, and line 3 & 4 reference it.

The `MemberNode` class is a subclass of the `DeclNode` class. `MemberNode` is used for all member attributes of a type, such as a field, method. Many member attributes like methods and properties can be overridden by subclasses, so each `MemberNode` has an array of immediate `derivedMembers` and `baseMembers`. `MemberNode` also has a `proxyLink` field that points to another `MemberNode` that uses this one as a proxy. Right now this is only used for properties that act as data accessors for private fields, so any reference to that property also adds an indirect reference to that field through the proxy link.

`MethodNode` is a subclass of the `MemberNode` class. It is used for methods, constructors, indexers, and operators. Because methods can be overloaded in C#, identifying which method is being called can be difficult. `argSyms` is an array of `BaseNodes` that contains an in-order list of argument types the method takes to help identify it. Even though methods can be overloaded, sometimes the number of arguments a method takes is still unique. To indicate when an overloaded method's number of arguments is unique, the `MethodNode` class has the `numArgsUnique` boolean flag. `MethodNode` also inherits from the `IScopeNode` interface. The `IScopeNode` interface is simply an empty interface that serves as a flag to indicate that the node creates its own new scope.

The `ScopeNode` is an empty class that inherits from the `BaseNode` class. It implements the `IScopeNode` interface so it is recognized as a scope delimiter by the symbol table generator. This node type is just used for indicating when a non symbol node creates a new scope, such as an if statement or while statement.

`TypeNode` is a subclass of the `ScopeNode` class. The `TypeNode` class is used for all the AST nodes that represent a type such as a class, struct, interface, delegate, or enum. Most types can inherit from parent types, so each `TypeNode` has an array of `derivedTypes` and `baseTypes`. Types can also be referenced so they also have an array of `references`.

To fully understand a user's code, the external libraries and user references must also be parsed. However those external libraries should not be considered for refactorings. For that reason, each `TypeNode` has a boolean flag `userDefined`, so the refactoring engine can differentiate between the user's code and external code. Ideally I would not need this flag, sadly it is necessary due to my symbol table's design. This flag was placed on the `TypeNode` class because it is the least frequent node. Any other node that needs to know if it is

user defined can simply walk up the tree until it finds a type node to check the value of the user defined flag.

3.3 Symbol Table Construction

The symbol table uses a scoped hierarchy approach where each scope gets its own symbol table. Each scope node has a name, unique id number, and a hash table of symbols. Since the symbol table takes the form of a scope tree, each scope node also knows its parent, child, and right sibling scope nodes. To save memory, the nodes in the AST also serve as symbols in the symbol table. Once the AST is constructed, four passes are then made over the tree, with each pass requiring linear time in the size of the program.

The first pass constructs the scope tree and inserts all the symbols. This is a very straightforward process since all the work has already been done by the parser when it classified the different AST node types. This process traverses the tree from the root down, each time it encounters a node that implements the `IScopeNode` interface it creates a new scope and inserts it in the scope tree. Each time a `DeclNode`, `MemberNode`, `MethodNode`, or `TypeNode` is encountered, that node is inserted into the current scope's hash table. By the end of this AST traversal, all the scope tree has been built and all the symbols have been inserted. Now all those symbols need their data fields filled in.

The second pass focuses on filling in each `DeclNode`'s `symType` field. This operation requires looking up that symbol's type, thus it must be done on the second pass once all the symbols are in the symbol table. Each time this pass encounters a `DeclNode` during its AST traversal, it looks up that `DeclNode`'s `symType` in the symbol table. By the end of this pass each `symType` node points to its type through the `symType` field.

The second pass also sets up each `TypeNode`'s `baseTypes` array. During the second pass's traversal of the AST, it looks up all the immediate inherited types and implemented interfaces for each `TypeNode` and adds them to its `baseTypes` array. The `derivedTypes` array is also populated at the same time. Each time a `TypeNode` *Child* adds another `TypeNode` *Parent*, to its `baseTypes` array, *Child* is automatically added to *Parent's* `derivedTypes` array. Once the second pass is complete, each type node knows who its parents and children are.

The third pass sets up each `MemberNode`'s base and derived types, and also sets up each `MethodNode`'s argument symbols. This third pass is different from the rest because instead of traversing the AST, this pass traverses the scope tree. As it does so, it searches each scope for type symbols. For each `TypeNode` found, all of its derived and base types are checked to see if they contain definitions of any of that type node's overridable members (properties, methods, events, and indexers). If so those derived and base members

are added to the `MemberNode`'s `derivedMembers` and `baseMembers` arrays respectively. The third pass also looks for `MethodNodes` in each scope's symbol table. For every `methodNode` found, each of its argument types are looked up and added to its `argSyms` array.

The fourth pass finds the references to each symbol. This is by far the most complicated pass in terms of implementation because it requires parsing every expression to discover which symbol each identifier is referencing. This pass traverses the AST looking for expression nodes. For each expression found, that expression is parsed from left to right and each identifier referring to a symbol is added to that symbol's `references` array. If during expression parsing there is an identifier whose symbol can not be found, then the expression parsing is aborted at that point since there is no way to continue without knowing what that identifier's symbol type is. Typically, symbol table lookup fails because the code is referencing a non-user defined symbol located in an external library that *Look#* could not find or load for some reason. Currently *Look#* still has difficulty locating some external libraries that projects use. The project file that *Look#* reads sometimes has out-of-date filepath information about the external libraries it uses. This importing process is the reason for the `userDefined` flag mentioned in the previous section.

3.4 Refactoring Detection

The refactoring engine was designed to be flexible and user configurable. Each refactoring checking class is completely independent of all the other refactoring checking classes. That means any number of refactoring checks can be performed in any order. This is accomplished by having all the refactoring checking classes implement a common interface defined in `RefactorChecker`. All refactoring checking classes inherit from the `RefactorChecker` class. This class defines the abstract method `createRefactorList()` and the abstract property `RefactorName`, that all of child classes of `RefactorChecker` must implement. The `createRefactorList()` method returns an array of `RefactorItem` objects. The `RefactorItem` class describes where a refactoring technique should be applied. It includes the name of the refactoring that should be used, a description of how and why it should be applied, and the code location where it should be applied. The `RefactorController` class simply takes a list of refactoring checking classes and sequentially calls `createRefactorList()` on each appending the results into an overall `RefactorItem` list that is eventually presented to the user. Any combination of refactoring checking classes in any order can be included in the list that `RefactorController` executes.

Many refactoring detection algorithms require that additional information be computed. Often this ad-

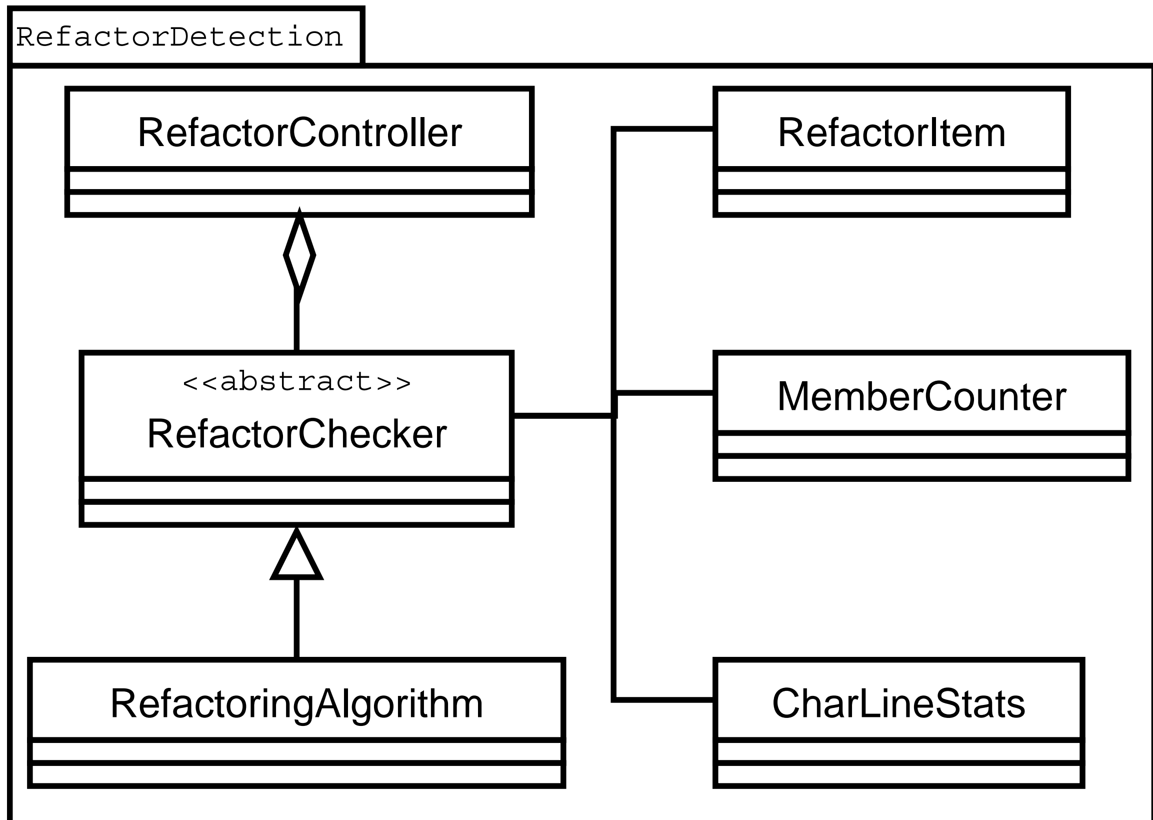

Figure 3.5: Class diagram of the refactoring engine. Note: each refactoring technique has its own class, but for simplicity sake, they are all represented by `RefactoringAlgorithm`.

ditional info is not unique to just one refactoring detection algorithm, so the refactoring engine includes a collection of common helper classes and methods. All of these helper methods and classes are inherited from the *RefactorChecker* class that all the refactoring checker classes inherit from.

The `CharLineStats` class takes a `BaseNode` and scans the source code that node represents and calculates the lines/characters of code, lines/characters of comments, and the ratio of comments to code. `CharLineStats` determines the first and last line of a node tree to find a line range, then opens the file containing that node and scans that line range counting the lines and characters of code and comments it finds while scanning that section of code. For example, if a method node is passed, then `CharLineStats` will calculate the lines/characters of code/comments for the entire method. These calculations are useful to refactoring techniques that need to know the size of methods and classes such as *Extract Class* or *Extract Method*.

The `MemberCounter` class takes a `TypeNode` node and counts the number of member attributes the type has defined. Individual counts are provided for each kind of member attribute (constants, fields, properties, methods, indexers, operators, events, and constructors). This information is useful for refactoring techniques that need to know the size or complexity of a type, such as *Extract Class*.

Most refactoring detection algorithms operate on a certain attribute or type, so the refactoring detection classes must examine every instance of that kind of attribute or type in the user's code. For that reason `RefactorChecker` provides the methods `GetFieldNodes()`, `GetPropertyNodes()`, `GetMethodNodes()`, and `GetClassNodes()`. Each of those methods returns a list of `BaseNode` objects that represent every attribute or type of that kind in the user's code. Since these are frequently used methods, once their results are calculated once, they are cached to improve the performance of future queries. Most refactoring techniques detection classes make use of these helper methods. For example, *Encapsulate Field* uses `GetFieldNodes()` to get the list of every field in the user's code, so it can check each field to evaluate if it needs to be encapsulated.

Some refactoring detection algorithms analyze the class hierarchy of a user's code. Most often such a refactoring algorithm needs to examine every class hierarchy in a user's code. `RefactorChecker` also provides the method `GetInheritanceList()` that returns an array of `BaseNode` objects with the root classes of every class hierarchy in the user's code. In other words it returns the inheritance forest. Since *Look#* is only analysing the user's code, there is no common inheritance root because `Object` is defined by the *System* library. For example, *Pull Up Method* uses `GetInheritanceList()` to examine every class hierarchy to evaluate if any methods should be pulled up.

Chapter 4

Refactoring Detection Algorithms

4.1 Overview

The primary challenge in making this tool is being able to correctly and accurately identify the situations when a certain refactoring technique should be applied. Many of the guidelines Fowler provides for determining when to apply certain refactoring methods are based on "human intuition" and abstract perceptions like "smells" [Fowler, 1999, p. 75-76]. This makes it extremely difficult to automatically identify where many of the refactoring methods should be applied. Fowler even claims that "no set of metrics rivals informed human intuition" [Fowler, 1999, p. 75]. Unfortunately it takes "informed human intuition" much longer to find refactorings in a large scale program than a computer program using metrics. Therefore the goal of *Look#* is not to definitively find all the refactorings in the users code, but to use metrics that look for common tell-tale signs of refactorings to help point "informed human intuition" in the right direction.

I classified the feasibility of automatically detecting the approximately seventy refactorings discussed in [Fowler, 1999]. About one third appear to be too high-level for any automated tool to detect (or perform). For example, *Substitute Algorithm*(See 2.3.19) suggests replacing an existing algorithm with another that is easier to understand. Computers can reduce simple mathematical equations. Well known compiler optimizations exist that can reduce mathematical equations involving one variable and multiple constants [Cooper and Torczon, 2004, p. 401]. Typically if a mathematical equation is not reduced as much as it can be, the developer left it unreduced for a reason. For example, the equation $2 * \pi * radius = circumference$ is actually clearer than the more reduced version $6.2832 * radius = circumference$. This is a case there the computer can detect an algorithm that can be reduced, but should not reduce it because the unreduced version is actually clearer.

Many algorithms that should be simplified are not strictly mathematical. For example, obviously there is no way a computer could automatically detect that the algorithm in figure 2.14(a) could be replaced with the better algorithm in figure 2.14(b). Refactoring techniques like *Substitute Algorithm* are best left for humans to detect and implement.

I also estimated that data-flow analysis [Aho et al., 1986, p. 586] is required to effectively detect about one sixth of the refactoring techniques Fowler describes. For example, to detect the *Split Temporary Variable* (See 2.3.20) refactoring technique, more detailed information is needed about where each value a temporary variable is given is used. Such information can be derived from the reaching definitions data-flow analysis [Aho et al., 1986, p. 610]. Without the reaching definitions information, accurately detecting this refactoring would be very difficult and error prone.

The remaining 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, I decided to build a tool that uses purely syntactic data such as symbol table information and simple code metrics.

4.2 Detecting refactorings

Of the refactorings that were classified as detectable through syntactic means, the following are the ones I chose to implement. Some of the refactoring techniques I implemented are not explicitly mentioned in [Fowler, 1999]. To automatically detect where a refactoring technique should be applied, each refactoring technique needs to be distilled down into a conditional that checks a set of counts, code metrics, and other attributes against a set of thresholds and expected values. That is exactly what I have done with the following refactoring techniques. All the algorithms that use counts or other metrics have variable thresholds that can be adjusted by the tool user.

4.2.1 Encapsulate Field

Construct a list of all field members in the code. Examine each field member and flag any field member that is declared public. This is easily the simplest refactoring to detect and also the most foolproof.

```
for c ∈ classes[program] do
    for f ∈ fields[c] do
        if PUBLIC ∈ modifiers[f] ∧ CONST ∉ modifiers[f] ∧ READ-ONLY ∉ modifiers[f] then
            report("encapsulate field", f)
```

Figure 4.1: *Encapsulate Field* algorithm.

Command	Description
fields	Takes a class and returns a set of all the fields defined on that class.
methods	Takes a class and returns a set of all the methods defined on that class.
properties	Takes a class and returns a set of all the properties defined on that class.
child-classes	Takes a class and returns a set of that class's child classes.
class-hierarchy-roots	Takes a set of classes and returns the subset classes that are at the top of a user-defined inheritance hierarchy.
implemented-interfaces	Takes a class and returns the set of interfaces that class implements.
parent-method	Takes an overridden method and returns the parent class method it is overriding.
modifiers	Takes a field, method, property, or class and returns a set of access modifiers for that item.
references	Takes a field or method and returns a set of all the identifier nodes that are a reference to that field or property.
parameters	Takes a method, and returns the set of parameters that method takes.
containing-class	Takes any non-type element and returns the type that contains it.
containing-method	Takes any statement, expression, or identifier and returns the method that contains it.
containing-property	Takes any statement, expression, or identifier and returns the property that contains it.
lines-of-code	Takes a class or method and returns the lines of code in the class or method body.
statements	Takes a method and returns the number of statements in the method body.
return-statements	Takes a set of statements and returns the subset of return statements.
identifiers	Takes a set of statements, and returns a set of identifiers found in those statements.

Table 4.1: Pseudo code commands key.

4.2.2 Remove Empty Method

Construct a list of all the methods in the code. Count the number of code statements in each method body. Any method body with a zero statement count most likely serves no purpose. The only time when this method should not be removed is if it is implementing an interface member or abstract method that must be implemented. Therefore check the parents of the method's class for interfaces that define this method or a parent class that defines an abstract method with the same signature. If the method is not implementing an interface method or an abstract method, and it has a statement count of zero, it should be flagged as an empty method. It is important that a statement count is used instead of a line count to detect this refactoring technique. If a simple line count is used, many situations where the developer commented out all the statements in the method body or left a blank line in the body would be missed.

for $c \in$ *classes*$[$*program*$]$ **do**
    for $m \in$ *methods*$[c]$ **do**
        if *statements[m]* $= 0$ **then**
            if $m \notin$ *implemented-interfaces*$[c] \vee ($OVERRIDE $\in$ *modifiers*$[m] \wedge$ ABSTRACT $\notin$ *modifiers*$[$*parent-method*$[m]])$ **then**
                report(*"Remove Empty Method"*, m)

Figure 4.2: *Remove Empty Method* algorithm.

4.2.3 Remove Unused Field

Construct a list of all field members in the code. Examine each field member and flag any field member that has no references in the user's code, meaning that field is not used anywhere in the code. The algorithm itself is simple, however finding every reference of a field can be difficult.

for $c \in$ *classes*$[$*program*$]$ **do**
    for $f \in$ *fields*$[c]$ **do**
        if $|$*references*$[f]| = 0$ **then**
            report(*"Remove Unused Field"*, f)

Figure 4.3: *Remove Unused Field* algorithm.

4.2.4 Remove Unused Method

Construct a list of all the methods in the code. Examine each method and flag any method that has no references in the user's code, meaning that method is not used anywhere in the code. This detection mechanism

is not totally accurate because the method may be defined as part of an external API only intended for other users to use. It is impossible to know if that is the case without the developer examining the code.

```
for c ∈ classes[program] do
    for m ∈ methods[c] do
        if |references[m]| = 0 then
            report("Remove Unused Method", m)
```

Figure 4.4: *Remove Unused Method* algorithm.

4.2.5 Hide Method

Construct a list of all the methods in the code. Examine each method's references. If all of a method's references are from within the class the method is declared in, then recommend hiding that method since no external classes are using it. Typically if a method is only used by the class that defines it, then there is no need to share it with the other classes since they do not seem to need it. Depending on the maturity of the system, this may not be true, the method may be used by an external class at some point in the future. In that case it is very easy to make the method public again once it is needed. Like *remove unused method*, this detection mechanism is not totally accurate because the method may be defined as part of an external API only intended for other users to use. It is impossible to know if that is the case without the developer examining the code.

```
for c ∈ classes[program] do
    for m ∈ methods[c] do
        if PUBLIC ∈ modifiers[m] then
            flag := TRUE
            for r ∈ references[m] do
                if containing-class[r] ≠ c then
                    flag := FALSE
            if flag = TRUE then
                report("Hide Method", m)
```

Figure 4.5: *Hide Method* algorithm.

4.2.6 Extract Class

Construct a list of all the classes in the code. For each class, count the lines of code, the number of fields, the number of methods, and the number of properties. If all of those counts exceed their corresponding thresholds, then the class should probably be extracted into multiple smaller classes. This algorithm relies on the assumption that most classes with a large number of attributes and span many lines are rather complex

and not very cohesive. Of course, it is possible to have a very large class with many attributes that is still very cohesive and crystal clear, but this algorithm assumes that is not the case typically.

$\lambda := 1000, \phi := 10, \mu := 10, \zeta := 5$
for $c \in$ *classes*$[$*program*$]$ **do**
  if *lines-of-code*$[c] \geq \lambda \wedge |$*fields*$[c]| \geq \phi \wedge |$*methods*$[c]| \geq \mu \wedge |$*properties*$[c]| \geq \zeta$ **then**
    report(*"Extract Class"*, c)

Figure 4.6: *Extract Class* algorithm.

4.2.7 Extract Method

Make a list of all the methods in the code. For each method, count the lines of code and the number of statements in the method's body. If both of those counts exceed their corresponding thresholds, then the method should probably be extracted into multiple smaller methods. The statement count is important because it prevents false positives from methods with statements that span many lines. For example, a method may have just a few statements, but one of those statements can contain a conditional expression that spans one or more screens. The line count is less important; however, it is useful for users that want to enforce strict policies on how many lines a method may span.

$\lambda := 70, \beta := 50$
for $c \in$ *classes*$[$*program*$]$ **do**
  for $m \in$ *methods*$[c]$ **do**
    if *lines-of-code*$[m] \geq \lambda \wedge |$*statements*$[m]| \geq \beta$ **then**
      report(*"Extract Method"*, m)

Figure 4.7: *Extract Method* algorithm.

The *Extract Method* algorithm is based on the assumption that a method with a large body can most likely be split up into smaller methods. This is done by extracting chunks of functionality from the large method into separate small methods. This algorithm also makes the assumption that the large the method body is, the more likely it is to have multiple chunks of code that are mostly independent of each other. Those independent chunks of code are easily extracted. This is not always the case, sometimes a method is so interconnected and cohesive that there is no functionality that can easily be extracted into separate methods.

4.2.8 Decompose Conditional

Traverse the AST searching for conditional (i.e., logical) expressions. Count the number of boolean OR and AND operators in each conditional expression found. Also count the number of statements in the method

body containing the conditional 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.

This algorithm assumes that counting the number of conditional logic junctions (boolean OR and AND operators) is an effective way of measuring the complexity of a conditional expression. In other words, the more conditions a conditional expression evaluates, the more complex it is. Obviously this is not always true, well named methods and variables can make large conditional expressions very clear. Conversely, poorly named methods and variables can make small conditional expressions very confusing. There is no good way for a computer to determine if the methods and variables used in a conditional expression are well named. Therefore this algorithm assumes that any conditional expression with a certain number of conditional logic junctions is most likely confusing or too complex.

```
α := 3, β := 12
for c ∈ classes[program] do
    for m ∈ methods[c] do
        DFS(m)
function DFS(n) do
    if type[n] = AND ∨ type[n] = OR then
        count := 1
        for d ∈ descendants[n] do
            if type[d] = AND ∨ type[d] = OR then
                count := count + 1
        if count ≥ α ∧ |statements[method-containing(n)]| ≥ β then
            report("decompose conditional", n)
    else
        for c ∈ children[n] do
            DFS(c)
```

Figure 4.8: *Decompose Conditional* algorithm.

4.2.9 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.) 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 automatically 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.

```
α := 2, β := 3
for c ∈ classes[program] do
    for m ∈ methods[c] do
        for n ∈ body[m] do
            if type[n] = LITERAL ∧ type[parent[n]] ≠ INITIALIZER then
                method-count[value[n]] := method-count[value[n]] + 1
        for v ∈ method-count do
            if method-count[v] > α then
                report("replace magic number", v, m)
            else
                class-count[v] := class-count[v] + method-count[v]
        delete method-count
    for v ∈ class-count do
        if class-count[v] > β then
            report("replace magic number", v, c)
    delete class-count
```

Figure 4.9: *Replace Magic Number* algorithm.

4.2.10 Move Field

Make a list of all the fields in the user's code. Examine each field, if a field has no references or is not static, then it should be ignored. Count how many of those references occur in each class in the user's code. If the class with the most field references is not the class that declares the field, then that field should be moved. This algorithm makes the assumption that if a field is used more by an external class than the class it is defined on, then it should be moved to that class.

The static check severely limits the number of fields considered by this refactoring algorithm. The static check was added after many of the fields flagged by this algorithm could not be moved due to problems with

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

a discrepancy in the number of instances of the class that defines the field and the class the field should be moved to. For example, many instances of a class `Student` with the field `name` may be passed to a single instance of a class `RosterGenerator`. Even though `RosterGenerator` may use `name` more than the `Student` class, because only one instance of `RosterGenerator` uses `name` but it is defined on many instances of the `Student` class, obviously `name` can not be moved to `RosterGenerator`. There is a fundamental problem with moving a field defined on a class with multiple instances to a class with a single instance. The simplest way to resolve with problem is to only focus on static fields which do not have this problem.

```
for c ∈ classes[program] do
    for f ∈ fields[c] do
        if |references[f]| > 0 ∧ STATIC ∉ modifiers[f] then
            for r ∈ references[f] do
                refcount[containing-class[r]] := refcount[containing-class[r]] + 1
                if refcount[highestcountclass] < refcount[containing-class[r]] then
                    refcount[highestcountclass] := refcount[containing-class[r]]
                    highestcountclass := containing-class[r]
            if highestcountclass ≠ containing-class[f] then
                report("move field", highestcountclass, f)
```

Figure 4.10: *Move Field* algorithm.

4.2.11 Move Method

This detection algorithm is based on the data class "smell" [Fowler, 1999]. 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.

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.

$\alpha := 0.40, \beta := 4, \gamma := 2$
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**
        $m :=$ *containing-method*(r)
        fields-referenced$[m] :=$ *fields-referenced*$[m] \cup \{f\}$
        total-references$[m] :=$ *total-references*$[m] + 1$
    for $m \in$ *fields-referenced* **do**
      if *total-references*$[m] \geq \beta \wedge |$*fields-referenced*$[m]| \geq \gamma$ **then**
        report(*"move method"*$, m, c)$
    delete *fields-referenced*, *total-references*

Figure 4.11: *Move Method* algorithm.

4.2.12 Remove Parameter

This refactoring algorithm is very similar to the *remove unused field* refactoring algorithm, except with method parameters instead of fields. Make a list of all the methods in the user's code. Ignore all methods with no parameters, overridden methods, and extern methods. Examine each parameter for every method, any parameter that has zero references should be flagged for removal. If the method is abstract or virtual, then all of its child instances must also be checked to make sure none of them use the parameter either. The method must also be checked to make sure it is not an implementation of an interface, in which case the method's signature can not be changed. Overridden and extern methods are not considered because their method signatures can not be changed. Thats also the same reason methods that implement interface methods are also not considered.

This algorithm makes the assumption that if a method does not make use of a parameter, then it does not need that parameter. There are many exceptions to this rule. If the method is implementing an interface method, then the method signature can not be changed so the parameter can not be removed. The same applies for methods that override virtual and abstract methods inherited from ancestors.

for $c \in$ *classes*[*program*] **do**
  for $m \in$ *methods*$[c]$ **do**
    if $|$*parameters*$[m]| > 0 \wedge$ OVERRIDE $\notin$ *modifiers*$[m]$ **then**
      if EXTERN $\notin$ *modifiers*$[m] \wedge m \notin$ *implemented-interfaces*$[c]$ **then**
        for $p \in$ *parameters*$[m]$ **do**
          if $|$*references*$[p]| = 0 \wedge (($VIRTUAL $\notin$ *modifiers*$[m] \vee$ ABSTRACT $\notin$ *modifiers*$[m]) \wedge$ *no-child-references*$[m, p])$ **then**
            report(*"remove parameter"*$, m, p)$

Figure 4.12: *Remove Parameter* algorithm.

4.2.13 Pull Up Field

Make a list of all the inheritance hierarchy roots in the user's code. Walk down each inheritance hierarchy and examine each class's subclasses. Compare all the subclass's fields and flag any fields that are common to all the subclasses. Any field with the same type and name that appear on all the subclasses, then that field should be pulled up to the parent class. This algorithm makes the assumption that any field with the same name and type will also be used in similar ways in a class hierarchy. It is possible for many subclasses to have the same field but use it in completely different ways, in which case moving the field up may not be appropriate.

```
α := 0.501
for c ∈ class-hierarchy-roots[classes[program]] do
    ExamineTree(c)
function ExamineTree(root) do
    if |child-classes[root]| > 1 then
        for c ∈ child-classes[root] do
            for f ∈ fields[c] do
                common-fields-count[f] := common-fields-count[f] + 1
        for f ∈ common-fields-count do
            if common-fields-count[f] = |child-classes[root]| then
                report("pull up field", f)
            else if common-fields-count[f]/|child-classes[root]| ≥ α then
                report("extract superclass", f)
    for c ∈ child-classes[root] do
        ExamineTree(c)
```

Figure 4.13: *Pull Up Field* algorithm.

4.2.14 Pull Up Method

This algorithm is identical to the *pull up field* refactoring algorithm with the exception that each method's arguments must be matched in addition to its name and type. Another difference with *pull up method* is that the similarities between methods can better be judged by comparing if the line counts of each method are similar. If all the methods are performing similar actions then they should all also have similar line counts. A better strategy would be to use some kind of duplicate detection. Future versions of *Look#* may use duplicate detection to improve this algorithm. This algorithm makes the assumption that multiple instances of any method in a class hierarchy with the same method signature will be used in very similar ways. It also assumes that methods with the same method signature and roughly the same number of lines are more likely to have a similar purpose.

```
α := 0.501, β := 20
for c ∈ class-hierarchy-roots[classes[program]] do
    ExamineTree(c)
function ExamineTree(root) do
    if |child-classes[root]| > 1 then
        for c ∈ child-classes[root] do
            for m ∈ methods[c] do
                common-methods-count[m] := common-methods-count[m] + 1
                if common-methods-max-lines[m] < lines-of-code[m] then
                    common-methods-max-lines[m] := lines-of-code[m]
                if common-methods-min-lines[m] > lines-of-code[m] then
                    common-methods-min-lines[m] := lines-of-code[m]
        for m ∈ common-methods-count do
            if common-methods-max-lines[m] − common-methods-min-lines[m] ≤ β then
                if common-methods-count[m] = |child-classes[root]| then
                    report("pull up method", m)
                else if common-methods-count[m]/|child-classes[root]| ≥ α then
                    report("extract superclass", m)
    for c ∈ child-classes[root] do
        ExamineTree(c)
```

Figure 4.14: *Pull Up Method* algorithm.

4.2.15 Extract Superclass

Extract superclass uses the exact same algorithms as *pull up field* and *pull up method.* The only difference is, instead of requiring that all of the subclasses have the same field or method, it only requires that a majority do. To avoid *extract superclass* and one of the pull up refactorings being suggested for the same attribute, the *extract superclass* algorithm does not trigger when all of the subclasses share the same attribute. Also, to avoid situations where extracting a superclass makes no sense, there must be at least three subclasses. That ensures that the *extract superclass* algorithm is never triggered on a single subclass. For example, if the threshold we set to fifty percent, and there were only two subclasses, then the algorithm would suggest extracting a superclass for every attribute not shared by the two classes.

4.2.16 Push Down Field

Make a list of all the inheritance hierarchy roots in the user's code. Walk down each inheritance hierarchy and examine each class's fields. If all of the references to a field occur in a subclass of the current class, then that field should be pushed down into that subclass. If references to that field appear in external classes outside of that class's inheritance hierarchy, then the field can not be moved since it is most likely being referenced though accessors in its defining class. Also, if references to that field appear in the class the field is defined in then that class is using that field, so the field can not be pushed down into a subclass. An exception to that last rule is when the references to a field only occur in properties that return that field. In those cases

the class is probably setting up a public accessor but not actually using the field. This algorithm is based on the assumption that if a subclass is the only place a parent class field is referenced, then that field is not used by the parent class, only the subclass. Therefore if the parent class is not using the field, it should be pushed down into the subclass that is using it.

```
α := 0.40
for c ∈ class-hierarchy-roots[classes[program]] do
    EXAMINETREE(c)
function EXAMINETREE(root) do
    if |child-classes[root]| > 1 then
        for f ∈ fields[root] do
            if |references[f]| > 0 then
                for r ∈ references[f] do
                    if containing-class[r] ∈ child-classes[root] then
                        reference-containers := reference-containers + containing-class[r]
                    else if containing-class[r] ≠
                    root ∨ ¬is-in-property[r] ∨ identifiers[return-statements[containing-property[r]]] ∩ references[f] = {} then
                        reference-containers := {}
                        break
                if |reference-containers| = 1 then
                    report("push down field", f, root)
                else if |reference-containers|/|child-classes[root]| ≤ α ∧ |reference-containers|/|child-classes[root]| > 0.0 then
                    report("extract subclass", f, root)
                reference-containers := {}
    for c ∈ child-classes[root] do
        EXAMINETREE(c)
```

Figure 4.15: *Push Down Field* algorithm.

4.2.17 Push Down Method

Make a list of all the inheritance hierarchy roots in the user's code. Walk down each inheritance hierarchy and examine each class's methods. If all of the references to a method occur in a subclass of the current class, then that method should be pushed down into that subclass. If references to that method appear in external classes outside of that class's inheritance hierarchy, then the method can not be moved. Also, if references to that method appear in the class the method is defined in then that class is using that method, so the method can not be pushed down into a subclass. This algorithm is based on the assumption that if a subclass is the only place a parent class method is referenced, then that method is not used by the parent class, only the subclass. Therefore if the parent class is not using the method, it should be pushed down into the subclass that is using it.

```
α := 0.40
for c ∈ class-hierarchy-roots[classes[program]] do
    EXAMINETREE(c)
function EXAMINETREE(root) do
    if |child-classes[root]| > 1 then
        for m ∈ methods[root] do
            if |references[m]| > 0 then
                for r ∈ references[m] do
                    if containing-class[r] ∈ child-classes[root] then
                        reference-containers := reference-containers + containing-class[r]
                    else
                        reference-containers := {}
                        break
                if |reference-containers| = 1 then
                    report("push down method", m, root)
                else if |reference-containers|/|child-classes[root]| ≤ α ∧ |reference-containers|/|child-classes[root]| > 0.0 then
                    report("extract subclass", m, root)
                reference-containers := {}
    for c ∈ child-classes[root] do
        EXAMINETREE(c)
```

Figure 4.16: *Push Down Method* algorithm.

4.2.18 Extract Subclass

Extract subclass is actually part of the *push down field* and *push down method* algorithms. If a field or method is referenced by more than one subclass, then those algorithms check if the field or method is referenced by a minority of subclasses. If a field or method is referenced by a minority of subclasses, then a subclass with that field or method can be extracted and placed between the parent class and the subclasses using the field or method. The push down algorithms also need to check if the ratio of subclasses that use a field or method is greater than zero. The way these algorithms are setup, there must be at least four subclasses for *extract subclass* to be suggested. The minority threshold is user adjustable, so the ratio does not necessarily need to be a minority ratio if the user wishes.

Chapter 5

Experimental Results

5.1 Overview

To validate the hypothesis that lightweight techniques can be used to accurately detect program refactorings, twelve public domain C# programs were analyzed with *Look#*. General information about each program is given in table 5.1. The programs are listed in this and other tables in order of increasing size. All the 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 in table 5.2. While the total time it takes

Program	**Version**	**Lines**	**Methods**	**Classes**	**Description**
FCKeditor	2.0	809	34	6	online text editor
MyACDSee	1.3	4118	137	14	image viewer
AscGen	2.0.2.4	4762	168	17	image conversion
TVGuide	0.4.3.1.3	6231	296	55	TV listings
MFXStream	1.0.0	7077	323	24	media streamer
GmailerXP	0.7	9347	288	29	mail client
3DProS	1.0	13629	381	22	3D animation
HeroStats	2.2.1	28840	1317	115	gaming statistics
ZedGraph	1.0	33539	1193	107	graphing package
NDoc	1.3.1	38118	1119	269	Documentation tool
NUnit	2.2.2	39910	1943	402	Unit testing tool
NAnt	0.85	80986	3416	408	Ant-like build tool

Table 5.1: Description of programs used in the experiments.

the *Look#* tool to completely analyze a program is a bit long, this is mostly due to a very inefficient symbol table generator. The refactoring detection engine runs very fast.

For each program in our test suite, table 5.3 gives the number of refactorings detected. Not surprisingly,

Program	AST Gen	Symboltable Gen	Refactoring Detection	Total
FCKeditor	2.08	75.25	0.02	77.34
MyACDSee	1.81	81.20	0.12	83.14
AscGen	2.28	79.31	0.19	82.20
TVGuide	3.00	110.20	0.17	113.38
MFXStream	4.08	88.11	0.29	92.48
GmailerXP	3.00	109.64	0.29	122.94
3DProS	5.65	139.59	0.55	145.80
HeroStats	5.42	127.95	1.28	134.65
ZedGraph	4.14	132.22	0.95	137.31
NDoc	6.69	147.75	1.64	156.07
NUnit	7.25	151.23	1.38	159.86
NAnt	7.61	178.02	1.96	188.44

Table 5.2: Summary of the time required to analyze the test case programs.

more refactorings were generally detected for the larger programs. Of course, all refactorings detected by *Look#* are not necessarily correct. Each reported refactoring was verified by hand to determine its correctness. Each refactoring reported is scored on a correctness scale from one to four, where one is definitely incorrect, two is probably incorrect, three is probably correct, and four is definitely correct. Unless otherwise stated, a score of three or four is considered correct, and a score of one or two is considered incorrect.

The correctness values reported in this paper are also not infallible. The correctness of each refactoring reported was only evaluated by one developer. Due to a combination of human error and the developer's personal opinions about refactoring application, a certain degree of error is inherent in the correctness scores.

Table 5.4 lists the percentage of refactorings that were correct for each program. Table 5.5 provides a summary of the correctness across all programs. Table 5.5 has three columns that show the success rate and success count of each refactoring if a success is only a four, if a success is a four or three, and if a success is a four, three, or two. Table 5.5 also shows the accuracy of each refactoring algorithm if correct is a four, maybe is a three or two, and incorrect is a one. It is interesting to see that the more subjective or gray area refactorings are the ones with higher maybe percentages.

5.2 Refactoring Detection Results

5.2.1 Encapsulate Field

Not surprisingly, the *encapsulate field* detection algorithm has a perfect score. However, the high count was slightly surprising. The high count shows that many developers do not acknowledge that public fields

	FCKeditorV2	GmailerXP	MyACDSee	3DProS	TVGuide	ascgen2	MFXStream	HeroStats	NAnt	ZedGraph	NDoc	NUnit	Total
Decompose Conditional	0	1	1	3	0	3	0	7	10	32	15	1	73
Encapsulate Field	0	39	13	128	102	6	13	177	29	226	69	90	892
Extract Class	0	0	0	2	0	0	0	1	0	1	1	0	5
Extract Method	0	5	1	10	1	1	0	4	28	9	10	1	70
Extract Subclass	0	0	0	0	0	0	0	0	4	0	0	0	4
Extract Superclass	0	0	0	0	0	0	0	0	8	0	1	1	10
Hide Method	3	13	4	16	12	4	8	59	64	57	38	49	327
Move Field	0	0	0	33	4	1	0	0	0	1	0	1	40
Move Method	0	23	0	32	11	1	8	15	48	32	32	19	221
Pull Up Field	0	0	0	0	0	0	0	0	7	0	0	5	12
Pull Up Method	0	0	0	0	0	2	0	0	2	3	0	1	8
Push Down Method	0	0	0	0	0	0	0	0	2	0	0	2	4
Remove Method	0	2	6	0	2	1	3	3	15	0	5	3	40
Remove Parameter	0	1	0	3	0	0	2	14	7	14	21	18	80
Replace Magic Number	1	34	13	40	5	15	21	18	28	78	16	9	278
Unused Field	0	3	4	2	51	2	1	77	16	5	7	5	173
Unused Method	0	18	5	12	29	4	10	142	409	67	101	60	857
Total	4	141	47	284	217	40	66	519	677	530	316	267	3108

Table 5.3: Number of refactorings detected for each program.

	FCKeditorV2	GmailerXP	MyACDSee	3DProS	TVGuide	ascgen2	MFXStream	HeroStats	NAnt	ZedGraph	NDoc	NUnit
Decompose Conditional	–	1.00	1.00	1.00	–	1.00	–	1.00	1.00	0.94	0.93	1.00
Encapsulate Field	–	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Extract Class	–	–	–	1.00	–	–	–	1.00	–	1.00	1.00	–
Extract Method	–	1.00	1.00	0.70	1.00	1.00	–	1.00	0.93	0.89	0.80	0.00
Extract Subclass	–	–	–	–	–	–	–	–	0.25	–	–	–
Extract Superclass	–	–	–	–	–	–	–	–	1.00	–	1.00	1.00
Hide Method	1.00	1.00	1.00	0.75	1.00	1.00	1.00	1.00	0.95	1.00	0.95	0.76
Move Field	–	–	–	0.64	0.00	0.00	–	–	–	1.00	–	0.00
Move Method	–	0.70	–	0.50	0.55	1.00	0.75	0.47	0.35	0.47	0.78	0.21
Pull Up Field	–	–	–	–	–	–	–	–	1.00	–	–	1.00
Pull Up Method	–	–	–	–	–	0.00	–	–	1.00	1.00	–	0.00
Push Down Method	–	–	–	–	–	–	–	–	0.50	–	–	1.00
Remove Method	–	1.00	1.00	–	1.00	1.00	1.00	1.00	0.80	–	0.20	0.67
Remove Parameter	–	1.00	–	1.00	–	–	1.00	0.93	0.86	0.93	1.00	0.78
Replace Magic Number	1.00	1.00	1.00	1.00	1.00	1.00	0.86	0.67	0.75	0.76	0.81	0.78
Unused Field	–	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Unused Method	–	1.00	1.00	0.92	0.93	0.75	1.00	1.00	0.99	0.85	0.94	0.57

Table 5.4: Success rates of refactorings detected for each program.

	Reported	**Success ≥ 4**		**Success ≥ 3**		**Success ≥ 2**		**2 ≤ Maybe ≤ 3**		
	Reported	**Correct**	**Rate**	**Correct**	**Rate**	**Correct**	**Rate**	**Correct**	**Maybe**	**Incorrect**
Decompose Conditional	73	51	0.70	70	0.96	73	1.00	0.70	0.30	0.00
Encapsulate Field	892	890	1.00	892	1.00	892	1.00	1.00	0.00	0.00
Extract Class	5	2	0.40	5	1.00	5	1.00	0.40	0.60	0.00
Extract Method	70	41	0.59	61	0.87	70	1.00	0.59	0.41	0.00
Extract Subclass	4	1	0.25	1	0.25	3	0.75	0.25	0.50	0.25
Extract Superclass	10	5	0.50	10	1.00	10	1.00	0.50	0.50	0.00
Hide Method	327	276	0.84	306	0.94	311	0.95	0.84	0.11	0.05
Move Field	40	14	0.35	22	0.55	35	0.88	0.35	0.53	0.13
Move Method	221	63	0.29	113	0.51	168	0.76	0.29	0.48	0.24
Pull Up Field	12	12	1.00	12	1.00	12	1.00	1.00	0.00	0.00
Pull Up Method	8	3	0.38	5	0.63	7	0.88	0.38	0.50	0.13
Push Down Method	4	2	0.50	3	0.75	3	0.75	0.50	0.25	0.25
Remove Method	40	30	0.75	32	0.80	33	0.83	0.75	0.08	0.18
Remove Parameter	80	68	0.85	73	0.91	73	0.91	0.85	0.06	0.09
Replace Magic Number	278	198	0.71	238	0.86	260	0.94	0.71	0.22	0.06
Unused Field	173	173	1.00	173	1.00	173	1.00	1.00	0.00	0.00
Unused Method	857	805	0.94	806	0.94	806	0.94	0.94	0.00	0.06
Total	3108	2637	0.85	2827	0.91	2943	0.95	0.85	0.10	0.05

Table 5.5: Summary of refactoring detections.

are a violation of good object oriented programming practice. The object oriented programming style calls for all fields to be encapsulated. Furthermore, C# provides the properties mechanism 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.

5.2.2 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 or is overridden by child classes. In the case of NDoc and NUnit, the incorrect *remove empty method* refactorings that are detected are incorrect because the developers purposely overrode parent class methods with empty methods to disable the functionality. In NAnt the incorrect *remove empty method* refactorings are methods with overridden child class implementations so they can not be removed either. This was not caught due to a bug in the *Look#* symbol table that causes it to occasionally fail to find child class implementations.

5.2.3 Remove Unused Field

The *remove unused field* algorithm has a perfect success rate. It is a fairly straight forward refactoring and is also easy to detect. Most of the detected unused fields are either legacy fields that are no longer used, or are part of a new section of code that is not yet complete.

5.2.4 Remove Unused Method

The high detection count of *unused method* refactorings is surprising, since this refactoring indicates a method that is never called. 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. Most of the incorrectly detected unused methods are due to *Look#*'s imperfect implementation of method overloading. Some of the key problems *Look#* has are resolving methods with an arbitrary number of arguments, and handling implicit down-casting of types. Those among other bugs contribute *Look#* missing method references and thus mistakenly labeling some methods as unused.

5.2.5 Hide Method

The *hide method* detection algorithm has a respectable success rate. *Hide method* has a lower success rate in 3DProS because some of the methods are only referenced outside the their local class through user-defined indexers, which *Look#* does not properly handle yet. *Hide method* also has a lower success rate in NUnit because many of the detected methods are part of a public API NUnit provides but does not use itself. The errors *Look#* has collecting references can be fixed, but there is no way to tell if a public method that is not used outside its local class is part of a published API or not.

5.2.6 Extract Class

The *extract class* refactoring was detected infrequently, which may indicate that its thresholds are too high. However further research revealed that lower the thresholds did indeed increase the number of detections, but it also lowered the success rate. The *Look#* project values the quality of the detected refactorings over the quantity, so the thresholds have been left at their high values. This is the kind of refactoring that typically occurs more frequently in larger projects. This is reflected in my data as *extract class* refactorings were only detected in the larger projects.

5.2.7 Extract Method

The *extract method* algorithm works fairly well and has a decent number of occurrences. The difference in results between *extract method* and *extract class* is surprising since they use similar algorithms: the *extract class* algorithm counts the number of lines, methods, and fields while the *extract method* algorithm counts the number of statements and lines. Many of the failures occur on large programs, however there does not appear to be any relation between the size of the program and the success rate of the *extract method* algorithm.

Most of the the *extract method* algorithm's failures are due to methods that have large switch statements that are difficult to break up and extract. Many of the other *extract method* failures are due to large methods that are very cohesive despite how large they are, and thus are difficult to break up and extract. In some of those cases the method is so cohesive that extracting part of the method could actually reduce understandability.

5.2.8 Decompose Conditional

This detection algorithm has a very good success rate. One its few errors was due to 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. The only other source of errors was a few cases where the conditional was sufficiently clear that it did not need to be decomposed and extracted. This was mostly due to well named variables and simple conditional expressions. In most cases the *decompose conditional* algorithm works very well, further confirming our hypothesis that refactorings can be correctly detected using purely syntactic techniques.

5.2.9 Replace Magic Number

Like *extract method*, the *replace magic number* refactoring algorithm experiences more failures on the larger programs. However, once again size was not the inherent reason for the failures. The larger programs such as NAnt, ZedGraph, NDoc, and NUnit 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);
```

There are also some cases where the same number is used multiple times but each use is totally unrelated. In those cases making a symbolic constant for those numbers does not make much sense.

5.2.10 Move Field

The success rate of the *move field* algorithm is definitely disappointing. The testing 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, the set of target fields was restricting to only static fields. This solves the multiple instance to single instance move problem since a static field has only one instance for all the instances of its defining class. Of course, this change caused far fewer refactorings to be detected. However, as stated before, the priority of the *Look#* project is the quality of detected refactorings, not the quantity.

The much higher detection count and success rate for 3DProS is interesting. This program uses a single structure 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.

5.2.11 Move Method

The *move method* refactoring algorithm does not have a very good detection rate. The current algorithm simply uses reference counts and data class detection to determine if a method should be moved into a data class. Obviously this algorithm is insufficient to accurately detect methods that should be moved. Some errors were caused by large copy methods that simply copy data from one class to another. Since all this data is being copied there is going to be a large amount of coupling between the two classes regardless of where the method resides so moving the method does not make much sense. Many other errors were caused by methods that referred to the same reference many times but did not use many different resources. In these cases, if temporary variables had been used to store the external data, then the methods probably would not have been reported as move method cases. Most of the detection failures were just simply because nothing in the method could be moved or extracted out. That represents a fundamental flaw in the detection algorithm. Compiling a list of all the references in a method and determining which class owns the most of those references might be a better strategy.

5.2.12 Remove Parameter

The *remove parameter* algorithm has a good successful detection rate. Many of the errors were caused by problems with the *Look#* symbol table either not detecting that a parameter was used, or not detecting that the method is derived from a parent class method or that a child class method overrides a method. In those cases the method's signature can not be changed so a parameter can not be removed. In NUnit, the errors were caused by empty methods that were purposely overriding the parent class methods with empty methods to disable that functionality. Since the methods were overloading parent class methods, their signatures could not be changed.

5.2.13 Pull Up Field

The *pull up field* algorithm has a very good successful detection rating but is detected infrequently. *Pull up field* refactorings were only detected in the larger projects with many classes. This makes sense because these larger projects also have larger inheritance hierarchies. Smaller projects tend to have very small inheritance hierarchies and so do not have any *pull up field* refactoring instances.

5.2.14 Pull Up Method

The *Pull up method* algorithm does not have a very good success rate. The reason for the discrepancy between *pull up field* and *pull up method* is, in order to pull up a method, all the methods must be doing roughly the same thing, to pull up a field, all the fields simply have to have the same type. The current *pull up method* algorithm simply tries to evaluate method similarity by comparing the size of the method body. Obviously this is a poor way to determine the similarity between methods. Most of the *pull up method* errors are caused by trying to pull up many different methods with very different bodies into a single method in a parent class. Using a better mechanism, like duplicate code detection, to measure the similarity between methods would increase the accuracy of this algorithm.

5.2.15 Extract Superclass

While the *extract superclass* algorithm appears to have a very good success rate, the maybe percentage is also very high. This is because while it correctly identifies fields and methods that could be extracted into a superclass. However, sometimes this algorithm is only triggered once on a class hierarchy, so it is simply not practical to create a whole new superclass just so a single method or field can be moved up into it. Ideally this algorithm would only trigger when multiple fields and or methods can be moved up into a superclass. Unfortunately, due to the current design of the refactoring engine, this is difficult to do.

5.2.16 Push Down Field

Unfortunately this algorithm did not detect any *push down field* refactoring instances in any of the test cases. I believe this is not because the algorithm is broken, but because it is quite rare for a class to declare a field and then not use it. In order to push a field down, the class that defines the field can not use it, only a single child class.

5.2.17 Push Down Method

The *push down method* algorithm also had very few detection instances, however most of those were correct. The only failure for this algorithm was when it recommended pushing down a method that accessed private class data not visible to the child class. This could be avoided by checking each reference in the method's body for private data. However, pushing down a method that accesses private data is not always wrong as sometimes that private data can also be pushed down with the method. Because very few instances were detected, not much can really be extrapolated from these results.

5.2.18 Extract Subclass

The *extract subclass* detection algorithm has a very poor success rate in contrast to *push down method* even though it mostly suggested pushing methods down into a new subclass. This most likely shows that the *push down method* algorithm does not work as well as the test case results seem to indicate. One error with the *extract subclass* algorithm occurred because it recommended pushing down a method that simply acted as a wrapper for another method. In that case it makes no sense to push down a wrapper method. Another error was from a method accessing private data again. Checking if a method accesses private data would have helped in this case.

5.3 Error Analysis

To help identify what caused the refactoring detection errors, each error was classified as either interpretive(I), algorithm(A), or tool(T). An interpretive error is an error such that even though the refactoring algorithm worked correctly, the refactoring it suggested was incorrect due to a subjective factor the algorithm can not detect accurately. For example, the problems the *unused method* algorithm has in NUnit with the public API methods are interpretive errors because there is no way a computer algorithm can accurately detect if a method is part of a public API or not. Another example of an interpretive error is when the *extract method* algorithm fails because the targeted method is classified as too cohesive and interconnected to breakup into multiple methods. In that case the *move method* algorithm itself worked properly in finding a large method, however the method it found was too cohesive to extract. It might be possible to redesign the algorithm so it measures cohesion in someway, but ultimately only a human will know for sure if a method should be extracted or not, thus those errors are interpretive errors. Algorithm errors are errors that are caused by failures in the algorithm. Typically these types of errors come in the form of special cases the are an

exception to the existing algorithm logic that could be patched up with special checks. An example of an algorithm error is when the *move method* algorithm suggests moving a method that is simply copying data from one class to another. In that case moving the method serves no purpose because the high degree of coupling between the two classes will remain the same because then the rolls would be switched and the other class would be directly accessing the original class's fields. The special case could probably be checked for by the algorithm with a good degree of accuracy. Tool errors are errors caused by an underlying problem in the *Look#* architecture. A good example is the *unused method* algorithm errors. The *unused method* algorithm simply checks if a method has zero references. If *Look#* fails to find the references to a method, it is the tool's fault for the failure, not the algorithm's.

	Errors	Count	I Type	I Count	A Type	A Count	T Type	T Count
Decompose Conditional	4.1	3	2.7	2	1.4	1	0.0	0
Encapsulate Field	0.0	0	0.0	0	0.0	0	0.0	0
Extract Class	0.0	0	0.0	0	0.0	0	0.0	0
Extract Method	12.9	9	8.6	6	4.3	3	0.0	0
Extract Subclass	75.0	3	50.0	2	25.0	1	0.0	0
Extract Superclass	0.0	0	0.0	0	0.0	0	0.0	0
Hide Method	6.4	21	3.1	10	0.9	3	2.4	8
Move Field	45.0	18	40.0	16	5.0	2	0.0	0
Move Method	48.9	108	38.9	86	10.0	22	0.0	0
Pull Up Field	0.0	0	0.0	0	0.0	0	0.0	0
Pull Up Method	37.5	3	25.0	2	12.5	1	0.0	0
Push Down Method	25.0	1	0.0	0	25.0	1	0.0	0
Remove Method	20.0	8	10.0	4	10.0	4	0.0	0
Remove Parameter	8.8	7	5.0	4	0.0	0	3.8	3
Replace Magic Number	14.4	40	12.9	36	1.4	4	0.0	0
Unused Field	0.0	0	0.0	0	0.0	0	0.0	0
Unused Method	6.0	51	2.1	18	0.1	1	3.7	32
Total	9.0	281	6.3	195	1.4	43	1.4	43

Table 5.6: Error types table.

Table 5.6 shows the break down of how many types of errors each refactoring algorithm had. Only three refactoring algorithms suffered from tool errors: *hide method*, *remove parameter*, and *unused method*. Most of these errors were caused because *Look#* failed to find method references, failed to properly link up the inheritance hierarchy, or failed in some other way. Many of the refactoring algorithms had a few algorithm errors. Most of these are special cases like methods that are called implicitly and should be blacklisted, access modifiers that should be checked for, and switch statements that should be avoided. The *move method* results in particular have many algorithm errors, most of these are attributed to large switch statements and

data copying methods. Most if not all of the algorithm errors could be fixed with more algorithm modification. Some algorithm errors have not been fixed due to time constraints, while other algorithm errors have not been fixed because there is a more fundamental problem with the algorithm and it should probably be rewritten anyway. In the case of *move method*, there are so many interpretive errors due to highly cohesive methods, the time fixing the algorithm errors would be better spent trying to develop a method cohesion metric. Interpretive errors were the most common errors. It is not surprising that many interpretive errors occur on the more complicated refactoring techniques like *replace magic number*, *move field*, and *move method*. It was surprising that some of the simpler refactoring algorithms like *unused method* and *hide method* had quite a few interpretive errors though. Most of those were caused by methods that were part of a program's public API. Most of the other interpretive errors were caused by some fundamental flaw in the refactoring algorithms. For example, the *replace magic number* algorithm can not tell which instances of a number are related, the *move field* algorithm can not determine the best class to move a field to, and the *move method* algorithm can not measure cohesiveness of a method.

From the results in table 5.6, it is plain to see that many of the refactoring algorithms could be improved. Ideally, there should be no tool errors at all. More work on the parser and symbol table could help eliminate those errors. Most of the algorithm errors could also probably be fixed especially on the *move method* algorithm. The interpretive errors are the hardest ones to fix. Some interpretive errors could be fixed with more complicated algorithms that calculate more metrics, while other errors could be fixed by using data flow analysis. Unfortunately some interpretive errors can not be avoided, like the public API errors.

5.4 Reliability of the Results

Overall the *Look#* tool appears to work pretty well on the analyzed programs. Of course those results were obtained after much fine tuning and bug fixes based on the initial results from those programs. Obviously my vision for the *Look#* tool does not include having to tweak the tool for each new program it analyzes. I theorized that the more programs the *Look#* tool is tuned on, the more consistent its results would be on different programs. Initially the tool was tuned on ten programs: FCKeditor, GmailerXP, MyACDSee, 3DProS, TVGuide, ascgen2, MFXStream, HeroStats, NAnt, and ZedGraph. Binomial confidence intervals were then computed for each refactoring algorithm based on its performance on those ten programs. Two more programs, NDoc and NUnit, were then analyzed without any additional tuning or bug fixes. Table 5.7 compares the refactoring algorithm confidence intervals for the first ten programs to the initial results of the

last two programs. Table 5.8 shows the initial results for NDoc and NUnit.

	First 10 Programs			**Initial Results**	
	Lower	**Rate**	**Upper**	**NDoc**	**NUnit**
Decompose Conditional	0.88	0.96	1.00	**0.82**	**0.50**
Encapsulate Field	1.00	1.00	1.00	1.00	1.00
Extract Class	0.18	0.57	0.90	1.00	–
Extract Method	0.77	0.88	0.95	0.80	**0.00**
Extract Subclass	0.01	0.25	0.81	–	–
Extract Superclass	0.69	1.00	1.00	1.00	1.00
Hide Method	0.87	0.91	0.95	0.95	**0.74**
Move Field	0.40	0.56	0.72	–	**0.00**
Move Method	0.42	0.49	0.57	0.78	**0.21**
Pull Up Field	0.65	1.00	1.00	–	1.00
Pull Up Method	0.55	1.00	1.00	–	**0.00**
Push Down Method	0.01	0.50	0.99	–	1.00
Remove Method	0.61	0.77	0.88	**0.20**	0.67
Remove Parameter	0.73	0.87	0.95	0.95	0.78
Replace Magic Number	0.81	0.86	0.90	0.81	**0.78**
Unused Field	0.94	0.98	0.99	1.00	1.00
Unused Method	0.86	0.88	0.90	0.86	**0.49**
Total	0.88	0.89	0.90	0.88	**0.73**

Table 5.7: Confidence interval of the first ten programs compared to the initial results of last two programs.

Unfortunately this analysis does not work very well with low numbers so some interpretation of the results is required. As you can see in 5.3, the following refactoring algorithms all have a very low detection count total: *extract class*, *extract subclass*, *extract superclass*, *pull up field*, *pull up method*, and *push down method*. As a result, the confidence intervals for these algorithms are quite broad and do not tell us much. Likewise, a low detection count for some refactoring algorithms on the two new programs analyzed also makes the results harder to read. If only one or two instances of a refactoring are found on a program, then chances are that algorithm's success rate is not going to fit nicely into an existing confidence interval because there just is not enough data. Much like it is difficult to draw a best fit curve when there are only 2 data points.

For example, *decompose conditional* appears to work very poorly on NUnit, however if you look at table 5.8, you'll notice that only two instances of *decompose conditional* were found and only one of them was a success. With only two instances of *decompose conditional* found, it is hard to draw any conclusions from that data. However seventeen instances of *decompose conditional* were reported on NDoc, and with a success rate of eighty two percent, it just barely falls outside the confidence interval. Even though the success rate was out of the confidence interval range, eighty two percent is still a good success rate so decompose conditional appears to work reasonably well without further modification.

The *encapsulate field* algorithm maintained its one hundred percent success rate in NDoc and NUnit. At

	Initial NDoc Results			Initial NUnit Results		
	Report	**Correct**	**Rate**	**Report**	**Correct**	**Rate**
Decompose Conditional	17	14	0.82	2	1	0.50
Encapsulate Field	69	69	1.00	90	90	1.00
Extract Class	1	1	1.00	0	0	–
Extract Method	10	8	0.80	1	0	0.00
Extract Subclass	0	0	–	0	0	–
Extract Superclass	1	1	1.00	1	1	1.00
Hide Method	38	36	0.95	50	37	0.74
Move Field	0	0	–	1	0	0.00
Move Method	32	25	0.78	19	4	0.21
Pull Up Field	0	0	–	5	5	1.00
Pull Up Method	0	0	–	1	0	0.00
Push Down Method	0	0	–	2	2	1.00
Remove Method	5	1	0.20	3	2	0.67
Remove Parameter	22	21	0.95	18	14	0.78
Replace Magic Number	16	13	0.81	9	7	0.78
Unused Field	7	7	1.00	5	5	1.00
Unused Method	111	95	0.86	70	34	0.49
Total	329	291	0.88	279	204	0.73

Table 5.8: The initial results for NDoc and NUnit.

this point it is safe to say that the *encapsulate field* algorithm works very well as long as you subscribe to the philosophy that public fields should not exist.

The *extract method* algorithm worked as expected on NDoc with a success rate of eighty percent landing it within the confidence interval. Since only one instance of the *extract method* refactoring was detected in NUnit, not much can be learned from its results. The ten reported instances of *extract method* on NDoc helps support the claim that the *extract method* algorithm can produce consistent results.

The *hide method* algorithm worked very well on NDoc and not so well on NUnit. The *hide method* score on NUnit was significantly lowered due to the public API it provides users using its framework because it does not use some of the framework itself. As discussed in section 5.2.5, there is no way to know if a method is part of a public API or not. Therefore, despite the poor performance on NUnit, I believe the *hide method* algorithm works well with no need for further tweaking.

The *replace magic number* algorithm performed decently on NDoc and a little worse than expected on NUnit. I believe the poor score derives from the fact that NUnit already makes good use of symbolic constants, so any remaining numbers are more likely to be unrelated. Perhaps the *replace magic number* algorithm could be made more intelligent but ultimately only a developer can tell if two numbers are related or not.

The *move method* algorithm exceeded expectations on NDoc and disappointed on NUnit. Both NDoc

and NUnit had decent detection counts as well so their results can not be chalked up as flukes. I believe this shows that the *move method* algorithm is still wildly inconsistent. In large part this is due to its inability to measure the cohesion of a method. The NDoc and NUnit results indicate that there is still much work to be done on the *move method* algorithm.

NDoc revealed a weakness in the *remove method* algorithm. I had not previously considered that a developer might wish to purposely override inherited methods with empty methods to disable some parent functionality. While the algorithm functioned as expected on NUnit, the NDoc results indicate that more tweaking of this algorithm is required before it will produce consistent results.

The *remove parameter* algorithm was one of the few that actually performed as expected on both NDoc and NUnit. Unfortunately the lower bound of the confidence interval is a little low for what should be a simple check. I believe the expected performance should be higher.

The *unused method* algorithm worked as expected on NDoc but worse than expected on NUnit. Like *hide method*, part of the reason behind this poor score was the public API NUnit provides. Another problem was the large number of non-user-defined types NUnit used that the *Look#* symbol table could not identify. This result lead to modifications to *Look#* the symbol table to allow it to handle non-user-defined types better. Despite the poor NUnit score, no changes or tweaks to the *unused method* algorithm are called for. Because the *unused method* algorithm is so simple, almost all its errors are either due to a bug in the underlying tool, or the result of an interpretive error that can not be fixed (such as a public API). The difference in scores between *unused field* and *unused method* is evidence of this. Despite having almost identical algorithms, the *unused method* algorithm's score is much lower because *Look#*'s symbol table has difficulty resolving some overloaded method references.

The confidence interval analysis produced some pretty mixed results. It showed that roughly one third of the refactoring algorithms produced consistent results. One third of the refactoring algorithms need further fine tuning to produce consistent and reliable results. The remaining refactoring algorithms did not have a high enough detection count to draw any meaningful conclusions from their results. Not surprisingly, the more complex refactoring techniques tended to be the ones that need more modification or did not produce many results.

After improving some of the refactoring algorithms based on the results of the initial confidence interval analysis, a second set of confidence intervals were calculated based on the results from all twelve analyzed programs. Table 5.9 compares the earlier confidence interval to the updated one. Mostly improvements were made to the underlying tool's data collection however some refactoring algorithms were further tuned as

	First 10 Programs			Final Analysis w/ All 12		
	Lower	**Rate**	**Upper**	**Lower**	**Rate**	**Upper**
Decompose Conditional	0.88	0.96	1.00	0.88	0.96	0.99
Encapsulate Field	1.00	1.00	1.00	1.00	1.00	1.00
Extract Class	0.18	0.57	0.90	0.55	1.00	1.00
Extract Method	0.77	0.88	0.95	0.77	0.87	0.94
Extract Subclass	0.01	0.25	0.81	0.01	0.25	0.81
Extract Superclass	0.69	1.00	1.00	0.74	1.00	1.00
Hide Method	0.87	0.91	0.95	0.90	0.94	0.96
Move Field	0.40	0.56	0.72	0.38	0.55	0.71
Move Method	0.42	0.49	0.57	0.44	0.51	0.58
Pull Up Field	0.65	1.00	1.00	0.78	1.00	1.00
Pull Up Method	0.55	1.00	1.00	0.24	0.63	0.91
Push Down Method	0.01	0.50	0.99	0.19	0.75	0.99
Remove Method	0.61	0.77	0.88	0.64	0.80	0.91
Remove Parameter	0.73	0.87	0.95	0.83	0.91	0.96
Replace Magic Number	0.81	0.86	0.90	0.81	0.86	0.90
Unused Field	0.94	0.98	0.99	0.98	1.00	1.00
Unused Method	0.86	0.88	0.90	0.92	0.94	0.96
Total	0.88	0.89	0.90	0.90	0.91	0.92

Table 5.9: Comparison of the confidence interval of the first ten programs and the final analysis with all twelve programs.

well. As you can see most of the confidence intervals and average success rates have improved in the final analysis. The few cases where the confidence interval and or average success rate actually decrease in the final analysis are mostly due to bad results in the last two programs, NDoc and NUnit, that were not fixed by the improvements.

5.5 False Negative Analysis

Up until this point, only false positive analyses have been discussed. Even though avoiding false positives is the primary concern of my refactoring algorithms, knowing the false negative rate is still very useful in evaluating the refactoring algorithms' success. After all, a refactoring algorithm is not much use if it successfully avoids returning any false positives, but misses over half of the valid refactorings as a consequence.

Performing a false negative analysis is much more time intensive than performing a false positive analysis. To perform a false negative analysis, every line of the target project's code must be examined by the evaluator and every possible refactoring identified. For example, to determine which methods were unused, every single method in the program was individually commented out one at a time and the program recompiled to check for compile errors. If the program compiled with no errors with a method commented out, then that

method was deemed unused. In contrast, a false positive analysis simply requires each reported refactoring be evaluated for accuracy.

Due to the time intensive nature of false negative analysis, it was only performed on one program: GmailerXP. GmailerXP was chosen because it was roughly the median of our test suite in terms of program size. It is also not prohibitively large so the false negative analysis could be done in an acceptable amount of time. Table 5.10 shows the original number of successful refactorings found by *Look#*, the number of refactorings found by hand, and the resulting false negative count. Unfortunately during the false negative analysis of GmailerXP, I realized it was not the best choice because it had absolutely no inheritance hierarchy, so obviously there were no results for any of the inheritance related refactoring algorithms such as *pull up method* and *push down field*.

	Orig Success	**By Hand**	**False Negatives**
Decompose Conditional	1	2	1
Encapsulate Field	39	39	0
Extract Method	5	22	17
Hide Method	13	13	0
Move Method	16	20	4
Remove Method	2	2	0
Remove Parameter	1	1	0
Replace Magic Number	34	37	3
Unused Field	3	6	3
Unused Method	16	16	0
Total	130	158	28

Table 5.10: GmailerXP false negative analysis.

The one *decompose conditional* refactoring missed was because even though the conditional had a small number of logic junctions, one of the conditions was having a string match a complex regular expression whose purpose was very clear.

There were many *extract method* refactorings missed. Most of the missed *extract method* refactorings were found during the false negative analysis by using a similarity detection tool called *Simian*. *Simian* revealed that many methods in GmailerXP contained duplicate code, which is one of the "smells" associated with the *extract method* refactoring [Fowler, 1999, p. 75]. The results of the false negative analysis indicate that the current size based metric for detecting *extract method* refactorings is insufficient. A duplicate detection mechanism needs to be added to catch all the methods with duplicate code being missed by the current algorithm.

A few *replace magic number* refactorings were missed because they involved blacklisted numbers. The

replace magic number algorithm normally ignores the numbers negative one though two because these are very commonly used numbers that oftentimes do not have a symbolic meaning.

The false negative analysis confirms that the *move method* refactoring detection algorithm does not work very well. Some of the missed refactorings did not have many references to an external class, but even so only manipulated the data from an external class. This algorithm needs to use a different metric.

A few *unused field* refactorings were missed because the fields in question did have references, but only in properties or methods that were subsequently unused. During an iterative refactoring process these unused fields would have been eventually detected by *Look#* once the user had removed the unused properties and methods using these fields. For that reason, this is not really considered a failure because these *unused field* refactorings would have eventually been detected after some maintenance had been performed.

With the exception of the *extract method* refactoring detection algorithm, all the algorithms had low false negative counts. This shows that most of my algorithms are catching most of the valid refactorings. Of course simply analyzing one program for false negatives does not prove much of anything. Especially since GmailerXP had no inheritance hierarchy. In the future I hope to conduct more false negative analyses to get a better idea of how well the refactoring algorithms work.

5.6 Conclusion

Overall the *Look#* tool worked well for accurately detecting refactoring techniques. As would be expected, the simpler refactoring techniques, like *encapsulate field* and *unused field*, were detected with a very high degree of accuracy. The more subjective and abstract refactoring techniques had lower success ratings, but some of the more advanced refactoring algorithms, like *replace magic number* and *decompose conditional*, still managed to have good success rates. I thought it was interesting to see that the more advanced and subjective refactoring techniques like extract class/method and the pull up and push down algorithms, were the ones with high maybe scores. That could mean that it is harder to definitively find the more advanced refactoring technique instances. I believe it illustrates that the correctness of the more advanced refactorings is a gray area as compared to the simple black and white conditions for the simpler refactoring techniques like *unused method.*

Some of the refactoring algorithms still need improvement. Most notably, *move field*, *move method*, *extract method*, *remove method*, *pull up method*, and *push down method* all have lower success rates that could be improved. The *extract method* and *move method* algorithms would probably benefit from some form

of duplicate code detection, as well as a metric for measuring method cohesion. The *move field* algorithm would benefit from an object instance count metric so it could operate on more than just static fields. The *remove method* algorithm could be improved if it ignored overriding methods. The *pull up method* algorithm needs a better way of evaluating method similarity than lines of code. The *push down method* algorithm needs to evaluate if any private data is accessed by the method.

Chapter 6

Related Work

The field of refactoring detection is still in its infancy. At this point it appears that more work has been done on automated refactoring tools that implement refactoring techniques for the user rather than detecting where refactoring techniques should be applied. [Mens and Tourwé, 2004] provide an excellent survey of the existing software refactoring research, including refactoring detection at both the design and source code level.

[Simon et al., 2001] have done similar research using metrics to detect refactoring techniques. Their tool uses a distance metric to measure the coupling between class elements, which appears to work better for some refactoring techniques than the reference count metric the *Look#* tool employs. Their tool is more oriented toward software visualization. It uses the Virtual Reality Modeling Language (VRML) to display each class element in 3D space. The distance between these class elements in the 3D model is determined by their distance metric. The tool user can then look for fields and methods that are isolated or do not belong with other objects in a cluster. However this tool still leaves it up to the user to draw their own conclusions about what needs refactoring based on the 3D view rather than explicitly identifying locations where refactorings should be applied. Due to the visualization aspect of the tool, their approach only works well when used on a small number of 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. While their approach may work better for comparing two or three classes, *Look#* is more efficient and user friendly at identifying refactorings on a large scale. Interestingly, their approach to detecting *move field* suffers from the same problem as ours once did: moving a field from a class with multiple instances to a class that has only a single instance.

[Tourwé and Mens, 2003] made a tool called *SOUL* that uses logic meta programming to identify in code. 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. This allows their detection algorithms to be mostly language independent. *Look#* , being AST based, is not as language independent. However, the authors admit that program facts alone 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 refactoring techniques: obsolete parameter and inappropriate interface. *Look#* has been tested on a larger set of programs and can detect many more refactorings.

[Kataoka et al., 2001] 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.

[Balazinska et al., 2000] and [Ducasse et al., 1999] both use duplicate code detection, also called clone detection, to find refactorings. [Fowler, 1999] discusses duplicate code as a major indicator of refactorings. Duplicate code detection is useful in detecting *extract method*, *extract class*, and *pull up method*. Incorporating similar duplicate code detection techniques into *Look#* could improve several of its refactoring algorithms. Because *Look#* is AST based, an AST based clone detection technique as described by Baxter et al. could be used. Future versions versions of *Look#* will probably utilize some form of duplicate code detection.

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 [Gamma et al., 1995].

Chapter 7

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 refactoring has focused on automating the changes to the source code. As of this writing, no tool exists (that the author has found) that does a very good job of identifying where refactoring techniques should be applied in C# code. There are some specialized tools that do a good job of detecting one or two specific refactoring techniques but none that work well for detecting a broad spectrum of refactoring techniques. The *Look#* tool that was developed during my research uses low-cost syntactic approach to finding refactorings in source code. My refactoring algorithms use a symbol table and AST together with simple code metrics such as line and statement counts.

The *Look#* tool was developed to evaluate the effectiveness of the refactoring algorithms. Twelve C# programs of varying size and type were analyzed with *Look#*. Over three thousand refactorings were discovered across those twelve C# programs. Each reported refactoring was then inspected by hand to determine its correctness and validity. As a result of this analysis, over 91% of the suggested refactorings were correct. Some of the refactoring algorithms even had a near-perfect success rate. Overall the majority of the refactoring algorithms had a success rate of 80% or more. During the this testing and analysis some interesting trends were noticed about the programs being analyzed. Many of the programs provide a rich set of interface methods that are never actually used. Additionally, the property mechanism of C# that was designed to transparently encapsulate public fields was not used as extensively as expected. Apparently many C# developers so not take the object-oriented principle of encapsulation very seriously as many public fields were discovered during our testing. I also noticed that the *Look#* tool tended to not work as well on programs that contain test

scaffolding for self tests. Some of the refactoring algorithms were confused by the tests.

Future work will focus on expanding the library of refactoring techniques *Look#* can detect, and improving the accuracy of the existing refactoring algorithms. While most refactoring algorithms had a good success rate, some of them were quite low. These algorithms might be improved by using different metrics or more advanced analysis techniques like data flow analysis.

For the purposes of this project, success rate is defined as the number of detected refactorings that are correct. False negatives are not considered in our success rate. The motivation for focusing on eliminating the false positives is that a tool that reports many refactorings, only a few of which are correct, will simply not be used because the user has to sort through too many bad reports. The whole purpose of automated refactoring detection is to save the user time. Reporting bad refactorings wastes the user's time. The real advantage of autonomously identifying refactoring techniques is how quickly a computer can process large amounts of data. A tool like this can rapidly find many possible refactoring techniques, thus dramatically reducing the amount of data the user needs to process during the refactoring process. However, false negatives are still a concern. An algorithm that correctly reports one refactoring but misses ten others has limited usefulness. *(Summarize results of false negative analysis)*

Bibliography

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

T. Archer and A. Whitechapel. *Inside C#*. Microsoft Press, One Microsoft Way, Redmond, Washington, 2nd edition, 2002.

M. Balazinska, E. Merlo, M. Dagenais, B. Lagiie, and K. Kontogiannis. Advanced clone analysis to support object-oriented system refactoring. In *Proceedings of the 7th Working Conference on Reverse Engineer*, pages 98–107. IEEE Computer Society Press, 2000.

I. Baxter, A. Yahin, L. Moura, M. SantAnna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pages 109–118, Bethesda, Mayland. IEEE Computer Society Press.

F. P. Brooks. *The Mythical Man-Month*. Addison-Wesley, New York, anniversary edition, July 2003.

P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Number 5. Addison-Wesley, Boston, MA, Nov. 2004.

K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, San Francisco, CA, 2004.

B. P. Douglass. *Real-Time UML: Developing Efficient Objects For Embedded Systems*. Object Technology Series. Addison-Wesley, Boston, MA, second edition, May 2002.

S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance*, pages 109–118, Washington. DC. USA, 1999. IEEE Computer Society Press.

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

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

J. Ganssle. Beyond functional firmware. *Embedded Systems Programming*, 12(4), Apr. 2003.

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

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

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.

S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, Upper Saddle River, New Jersey.

R. W. Sebesta. *Concepts Of Programming Languages*. Addison-Wesley, Boston, MA, 5th edition, 2002.

F. Simon, F. Steinbrücker, and C. Lewerentz. Metrics based refactoring. pages 30–38, Lisbon, Portugal, 2001.

T. Tourwé and T. Mens. Identifying refactoring opportunities using logic meta programming. pages 91–100, Benevento, Italy, Mar. 2003.

Index

Argument Passing, 8

AST, 5

BaseNode, 21

C#, 4, 8

CharLineStats, 27

CommonAST, 21

correct, 43

Data Abstraction, 7, 9

Data Class, 14

Data Hiding, 7

Data-flow Analysis, 5

DeclNode, 21

Decompose Conditional, 13

Dynamic Binding, 8

Encapsulate Field, 10

Extract

- Class, 12
- Method, 12
- Subclass, 17
- Superclass, 16

Hide Method, 11

incorrect, 43

Inheritance, 7

Interface, 8

IScopeNode, 23

LOC, 21

Magic Number, 14

MemberCounter, 27

MemberNode, 23

MethodNode, 23

Move

- Field, 14
- Method, 15

Object-oriented, 7

override, 8

Polymorphism, 7

Properties, 9

Pull Up

- Field, 15
- Method, 16

Push Down

- Field, 16
- Method, 17

References, 21

Remove

- Empty Method, 10

Parameter, 15
Unused Field, 11
Unused Method, 11
Replace Magic Number, 14

ScopeNode, 23
Software Life Cycle, 1
Split Temporary Variable, 18
Static Analysis, *see* Data-flow Analysis
Substitute Algorithm, 17

TypeNode, 23

userDefined, 23, 25

virtual, 8