Improving Reuse in Software Development for the Life Sciences

Nicholas Vincent Iannotti
Purdue University

Follow this and additional works at: https://docs.lib.purdue.edu/open_access_dissertations
Part of the Databases and Information Systems Commons

Recommended Citation

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
This is to certify that the thesis/dissertation prepared

By Nicholas Vincent Iannotti

Entitled
Improving Reuse in Software Development for the Life Sciences

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Michael D. Kane  
Chair

John A. Springer

Thomas J. Hacker

Roy D. Dejoie

To the best of my knowledge and as understood by the student in the Research Integrity and Copyright Disclaimer (Graduate School Form 20), this thesis/dissertation adheres to the provisions of Purdue University’s “Policy on Integrity in Research” and the use of copyrighted material.

Approved by Major Professor(s): Michael D. Kane

Approved by: James L. Mohler  11/11/2013

Head of the Graduate Program  Date
IMPROVING REUSE IN SOFTWARE DEVELOPMENT
FOR THE LIFE SCIENCES

A Dissertation
Submitted to the Faculty
of
Purdue University
by
Nicholas V. Iannotti

In Partial Fulfillment of the
Requirements for the Degree
of
Doctor of Philosophy

December 2013
Purdue University
West Lafayette, Indiana
Dedicated to my family.
ACKNOWLEDGMENTS

I wish to gratefully thank Professors Michael Kane, John Springer, Thomas Hacker, and Roy Dejoie for their support, guidance, and patience.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>ABBREVIATIONS</td>
<td>ix</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>x</td>
</tr>
<tr>
<td>CHAPTER 1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Statement of the Problem</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Significance of the Problem</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Purpose of the Study</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Hypothesis</td>
<td>6</td>
</tr>
<tr>
<td>1.5 Delimitations</td>
<td>7</td>
</tr>
<tr>
<td>1.6 Limitations</td>
<td>8</td>
</tr>
<tr>
<td>1.7 Organization of the Study</td>
<td>8</td>
</tr>
<tr>
<td>CHAPTER 2. REVIEW OF THE LITERATURE</td>
<td>10</td>
</tr>
<tr>
<td>2.1 Design Patterns</td>
<td>10</td>
</tr>
<tr>
<td>2.1.1 Changing the Timing of Reuse</td>
<td>11</td>
</tr>
<tr>
<td>2.1.2 Pattern Categories</td>
<td>11</td>
</tr>
<tr>
<td>2.1.3 Patterns Are Abstractions</td>
<td>12</td>
</tr>
<tr>
<td>2.1.4 Pattern Implementations</td>
<td>12</td>
</tr>
<tr>
<td>2.1.4.1 Generics</td>
<td>12</td>
</tr>
<tr>
<td>2.1.4.2 Class Libraries</td>
<td>13</td>
</tr>
<tr>
<td>2.1.4.3 Frameworks</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Fundamental Patterns and Related Concepts</td>
<td>14</td>
</tr>
<tr>
<td>2.2.1 Object Interactions</td>
<td>14</td>
</tr>
<tr>
<td>2.2.1.1 Event Pattern</td>
<td>14</td>
</tr>
<tr>
<td>2.2.1.2 Command Pattern</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1.3 Mediator Pattern</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1.4 Event Aggregator Pattern</td>
<td>17</td>
</tr>
<tr>
<td>2.2.2 Interactions in Object Collections</td>
<td>17</td>
</tr>
<tr>
<td>2.2.2.1 The Iterator and Observer Patterns</td>
<td>17</td>
</tr>
<tr>
<td>2.2.2.2 Push vs. Pull Model</td>
<td>18</td>
</tr>
<tr>
<td>2.2.3 Presentation Model Pattern</td>
<td>18</td>
</tr>
<tr>
<td>2.2.4 Plugin Pattern</td>
<td>19</td>
</tr>
<tr>
<td>2.2.5 Method Chaining Pattern</td>
<td>20</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Descriptions of Tools and Software Used</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Implementations of the Rx Library in Other Languages and Platforms</td>
<td>22</td>
</tr>
<tr>
<td>3.1 Programs Included in the “Entrez Applications” Case Study</td>
<td>27</td>
</tr>
<tr>
<td>3.2 Pattern and Concept Map of Implementations</td>
<td>29</td>
</tr>
<tr>
<td>3.3 Quantitative Metrics Used in Measuring Code Reusability</td>
<td>31</td>
</tr>
<tr>
<td>3.4 Procedure to Extract Code Metric Measurements Using NDepend</td>
<td>32</td>
</tr>
<tr>
<td>3.5 Hypothesis Evaluation Criteria</td>
<td>34</td>
</tr>
<tr>
<td>4.1 Usage of Legwork Code in Case Studies</td>
<td>36</td>
</tr>
<tr>
<td>4.2 Effect on Mean Metric Measurements</td>
<td>40</td>
</tr>
<tr>
<td>4.3 Workflow Used by Service Components to Call a Web Service</td>
<td>41</td>
</tr>
<tr>
<td>4.4 Workflow Used for “Text Linked DataDocs” Data Pipeline</td>
<td>44</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 The Number of Resources, Tools, and Databases Registered at Bioinformatics.ca Per Year</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Prevalence of .NET Framework Version Support</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Legwork: A Class Library Focusing on Object Interactions Designed to Improve Code Reuse in Software Development for the Life Sciences</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Change Notification Methods in the MVC and MVVM Design Patterns</td>
<td>19</td>
</tr>
<tr>
<td>2.2 Using LINQ for Filtering, Sorting, and Projection</td>
<td>21</td>
</tr>
<tr>
<td>2.3 Converting a Property to Support Change Notifications Using the Event Pattern and INPC</td>
<td>24</td>
</tr>
<tr>
<td>4.1 Metric Measurement Summaries for the Entrez Applications Case Study</td>
<td>38</td>
</tr>
<tr>
<td>4.2 Metric Measurement Summaries for the .NET Bio Case Study</td>
<td>39</td>
</tr>
<tr>
<td>4.3 Code-Control Flow Diagram for Calling a Web Service Using Legwork</td>
<td>42</td>
</tr>
<tr>
<td>4.4 Code-Control Flow Diagram for Calling a Web Service Using Best Practices</td>
<td>43</td>
</tr>
<tr>
<td>4.5 Usage of Prism's Event Aggregator Pattern Requires Creating an Additional Prism-Derived Class for Each Event</td>
<td>43</td>
</tr>
<tr>
<td>4.6 Activity Diagram for “Text Linked DataDocs” Data Pipeline</td>
<td>45</td>
</tr>
<tr>
<td>4.7 Legwork Code and Sequence Diagram From the “Text Linked DataDocs” Data Pipeline from the Entrez Applications Case Study</td>
<td>47</td>
</tr>
<tr>
<td>4.8 Best Practices Code and Sequence Diagram from the “Text Linked DataDocs” Data Pipeline from the Entrez Applications Case Study</td>
<td>48</td>
</tr>
<tr>
<td>5.1 Dependency Diagrams Generated from Legwork and Prism Source Code</td>
<td>53</td>
</tr>
<tr>
<td>5.2 An Example of Design Pattern Encapsulation and Composition Using Legwork</td>
<td>56</td>
</tr>
<tr>
<td>5.3 Development Time and Cost for the Entrez Applications and .NET Bio Case Studies</td>
<td>58</td>
</tr>
<tr>
<td>A.1 Difference in Cyclomatic Complexity Between Legwork and Best Practices in the Entrez Applications Case Study</td>
<td>67</td>
</tr>
<tr>
<td>Figure</td>
<td>Difference in Efferent Coupling Between Legwork and Best Practices in the Entrez Applications Case Study</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>A.2</td>
<td></td>
</tr>
<tr>
<td>A.3</td>
<td>Difference in Lines of Code Between Legwork and Best Practices in the Entrez Applications Case Study</td>
</tr>
<tr>
<td>A.4</td>
<td>Difference in Lack of Cohesion of Methods Between Legwork and Best Practices in the Entrez Applications Case Study</td>
</tr>
<tr>
<td>A.5</td>
<td>Difference in Cyclomatic Complexity Between Legwork and Best Practices in the .NET Bio Case Study</td>
</tr>
<tr>
<td>A.6</td>
<td>Difference in Efferent Coupling Between Legwork and Best Practices in the .NET Bio Case Study</td>
</tr>
<tr>
<td>A.7</td>
<td>Difference in Lines of Code Between Legwork and Best Practices in the .NET Bio Case Study</td>
</tr>
<tr>
<td>A.8</td>
<td>Difference in Lack of Cohesion of Methods Between Legwork and Best Practices in the .NET Bio Case Study</td>
</tr>
</tbody>
</table>
ABBREVIATIONS

API Application Programming Interface
APM Asynchronous Programming Model
BLAST Basic Local Alignment Search Tool
CC Cyclomatic Complexity
EA Entrez Applications
EC Efferent Coupling
IDE Integrated Development Environment
IQ Interquartile
INPC INotifyPropertyChanged
LCOM Lack of Cohesion of Methods
LINQ Language Integrated Query
LOC Lines of Code
MEF Managed Extensibility Framework
MVC Model View Controller
MVVM Model View ViewModel
NCBI National Center for Biotechnology Information
OO Object-Oriented
Rx Reactive Extensions
WPF Windows Presentation Foundation
ABSTRACT

Iannotti, Nicholas V. Ph.D., Purdue University, December 2013. Improving Reuse in Software Development for the Life Sciences. Major Professor: Michael D. Kane.

The last several years have seen unprecedented advancements in the application of technology to the life sciences, particularly in the area of data generation. Novel scientific insights are now often driven primarily by software development supporting new multidisciplinary and increasingly multifaceted data analysis. However, despite the availability of tools such as best practice frameworks, the current rate of software development is not able to keep up with the needs of scientists. This bottleneck in software development is largely due to code reuse generally not being applied in practice.

This dissertation presents Legwork, a class library of reuse-optimized design pattern implementations for desktop applications written in the C# programming language using Microsoft’s .NET Framework. Two case studies were used to evaluate the effect of Legwork on improving code reusability as compared to Microsoft’s “best practices” Prism framework. First, a collection of six established web service-based workflows leveraging the National Center for Biotechnology’s Entrez database retrieval system. Second, a modular genomics data analysis and visualization application based on the open source “.NET Bio” bioinformatics toolkit.

Employing quantitative and qualitative methods, code reusability was evaluated at the class, subsystem, and system levels of software design through comparing established class metrics for code reuse, code control flow, and code composition, respectively. The results from both case studies demonstrate that using Legwork provides a consistent improvement in code reusability over Microsoft’s Prism framework across all three levels of program design evaluated.
CHAPTER 1. INTRODUCTION

Code reuse is one of the oldest concepts in programming. Parametrized procedures enable a set of instructions to be named and customized for reuse within different contexts (Biddle, Martin, & Noble, 2003). Object-oriented (OO) programming allows data structures to contain procedures that use them, enabling the reuse of both data and procedures together. One of the reasons that OO programming is credited with becoming the prevalent programming paradigm is due to code reuse becoming increasingly important (Johnson & Foote, 1988). More recently, reuse has taken the form of the "separation of concerns" or grouping code into components by function, with the goal of composing these components into a functional application.

While object-orientation and component-based design represent significant advancements, code reuse continues to be referred to as the "holy grail" of software development; always sought and yet never found (Bosch, 2005; Llorens, Fuentes, Prieto-Diaz, & Astudillo, 2006). In contrast, computer hardware has seen significant breakthroughs in many areas such as processing speed, hard drive capacity, and in reducing the cost of sending information over networks (Moore, 1965; Tehrani, 2000; Walter, 2005).

Before Moore's Law, no other technology had ever observed the same magnitude of performance gains (Brooks, 1987). However, the last few years have seen unprecedented improvements in the application of technology to science. In many cases, such as next-generation sequencing platforms for example, these advancements are far outpacing the improvement in computational power under Moore's law (Zerbino, Paten, & Haussler, 2012). In a very short amount of time, the amount of data available for processing has gone from being considered a "Data Deluge" (Hey & Trefethen, 2003) to a "Data Tsunami" (Stein, 2010) with public
data repositories already containing hundreds of thousands of experimental results (see Figure 1.1) (Huttenhower, 2009). To put things into perspective, in 2001 it cost approximately one hundred million dollars to sequence one human genome. Today, the cost is only a few thousand dollars, and in the next five to ten years sequencing an entire human genome is projected to cost less than routine lab tests (Zax, 2012).

![Figure 1.1. The Number of Resources, Tools, and Databases Registered at Bioinformatics.ca Per Year (Brazas, Yamada, & Ouellette, 2010).](image)

As technological advancements have made scientists increasingly able to detect, measure, and quantify experimental results and microscopic activities, scientific research has turned to software to process and understand this data (Huttenhower, 2009). In genomics, for example, the increasing availability of data is leading to the need for multidimensional analysis tools incorporating DNA sequence alignments, sequence variant lists, phylogenetic trees, functional and epigenomic assays, and phenotypic deviants. Obtaining the maximum information from these experiments now requires the integration of many different areas of research linked together through mathematical models (Zerbino et al., 2012).
At present, the current rate of software development in the life sciences is not able to keep up with this need. This can largely be traced to the fact that code reuse is simply not being applied (Llorens et al., 2006). However, recent progress in software development and design environments have now made significant improvements possible. Toward increasing support for reuse, object-oriented programming languages and integrated development environments (IDE) have increasingly raised the level of abstraction with the intention of allowing developers to spend more time delivering functionality and less time writing low-level infrastructure (Miller, 2009). As the most used IDE, Microsoft Visual Studio users represent the largest software development audience, with Microsoft technology as a whole combining to form the largest and most diverse network of application development.

Currently, the most active framework is the .NET Framework version 4.0 (see Figure 1.2), which is fully supported by Windows XP Service Pack 3 or greater and is natively supported by Visual Studio 2010.

---

![Figure 1.2. Prevalence of .NET Framework Version Support. Determined from an analysis of 28 million unique visitors in January 2012 using the Internet Explorer browser (Statowl, 2012).](image)

Building upon previous versions, .NET Framework 4.0 contains three features that have received a particularly large amount of attention. First, in addition to the traditional pull-based model, version 4.0 also contains a definition for push-based object interactions using event patterns through the IObserverable
interface (Microsoft, 2011a). Second, the Managed Extensibility Framework (MEF) has been officially integrated after years of open source development. MEF uniquely supports run-time code discovery and composition through a plug-in-like architecture based on code contracts and attributed tagging of composable parts. Third, the .NET Framework now contains a new type, “dynamic.” These dynamic objects bypass static compile-time checking and enable run-time value resolution. Though each of these three features affects very different aspects of software development, their collective inclusion in the .NET Framework version 4.0 provides the groundwork to greatly advance the current state of code reuse in software development for the life sciences.

1.1 Statement of the Problem

Reuse continues to be a rare occurrence in software development for the life sciences (Llorens et al., 2006). Even in cases where documented libraries specifically designed for reuse are available, reuse is still not widely incorporated into software development practice (Ko et al., 2011). Instead, scientists often create their own incarnation, resulting in the duplication of work and proliferation of “software silos” (Dubois, 2002). This practice often results in “quick and dirty” coding practices and wastes time and effort by scientists (Kilkenny & Boyle, 2009).

Many of the roadblocks facing code reuse can be traced to the absence of platform support for several important code characteristics that impact reusability: encapsulation, modularity, minimizing complexity and component coupling, and the improvement of code expressivity. While “best practice frameworks” are available to assist in application development, they require large initial investments resulting in up to 300% increase in development costs (Poulin, 2006) and often greatly increase code complexity (Schmidt, 1999).
1.2 Significance of the Problem

Though particularly affecting software development for the life sciences, the issues yet to be solved with regard to code reuse are pervasive problems in software development (Deelman, Gannon, Shields, & Taylor, 2009). For example, the application infrastructure responsible for managing code reuse in Adobe’s desktop applications accounts for half of all of the bugs reported during a product development cycle (Maier, Rompf, & Odersky, 2010). Surveys of scientists indicate similar trends: writing and debugging code interactions and software construction often occupies the majority of their time instead of focusing on the underlying science being investigated (Wilson, 2009). Improving the integration of code reuse would lead to more rapid application development and potentially accelerate the current rate of scientific discovery (Huttenhower, 2009). Better support for code reuse can also have a considerable impact on resource allocation; reusing common components instead of developing them from scratch has been shown to reduce development costs by more than 90% (Goulão & Abreu, 2011). Sustaining an increasing rate of advancement in life sciences research will require the research community leverage code reuse (U.S. DOE, 2010).

1.3 Purpose of the Study

The focus of this dissertation is on improving the reuse of code in software development for the life sciences. To address current code reuse roadblocks, we have created the “Legwork” class library. Legwork improves code reuse by providing reuse-optimized design patterns and infrastructure support to better manage object interactions in code (see Figure 1.3).

This study differs from previous work as well as current best practices in two fundamental aspects. First, the emphasis on improving reuse is through the design and development of a code class library, as opposed to a framework. This is an important distinction that represents a key difference in design and a departure
Figure 1.3. Legwork: a class library focusing on object interactions, designed to improve code reuse in software development for the life sciences.

from the “all or nothing” requirement associated with using a framework (Killcoyne & Boyle, 2009). Second, we approach code reuse from three separate vantage points: minimizing code dependencies, establishing new implementations design patterns, and providing infrastructure support for design pattern encapsulation, composition, and code control flow. Through quantitative and qualitative code evaluation in two case studies, we demonstrate using Legwork improves code reuse over current best practices.

1.4 Hypothesis

The objective of this work is to present evidence supporting the rejection of the Null Hypothesis and the acceptance of the Alternative Hypothesis:

H₀: Using Legwork offers no quantitative or qualitative improvement in reuse in software design and development over current best practices in
desktop applications using the C# programming language and targeting the .NET Framework version 4.0.

H₀: Using Legwork offers quantitative and qualitative improvement in reuse in software design and development over current best practices in desktop applications using the C# programming language and targeting the .NET Framework version 4.0.

1.5 Delimitations

Using the Legwork class library requires a few years of experience with the C# programming language. Though open-source alternatives may be used, the work presented was developed and tested using the Visual Studio 2010 IDE. The code associated with the Legwork class library itself has two dependencies: .NET Framework 4.0, and the Reactive Extensions (Rx) class library. Visual Studio 2010, .NET Framework 4.0, and Rx are all actively supported by Microsoft and are freely available for download. Seen in Table 1.1 below and described in detail in Section 3.2.1.2 the “NDepend” code analysis tool was also used to conduct static analysis of source code files and provide code reuse metric measurements (Smacchia, 2012).

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual Studio 2010</td>
<td>IDE used for programming and diagramming</td>
</tr>
<tr>
<td>Reactive Extensions 1.1.1</td>
<td>Class library for event processing</td>
</tr>
<tr>
<td>NDepend 4.0.1</td>
<td>Used to provide code metric measurements</td>
</tr>
</tbody>
</table>
From a high-level perspective, the focus of this work is on the application of programming concepts and techniques to facilitate reuse in software development for the life sciences. Many of concepts described are applicable to other areas of science and other programming languages based on the object-oriented programming paradigm. However, as stated in the Hypotheses, the scope of this dissertation only extends to using the C# programming language for .NET Framework-based software development in the life sciences targeting desktop application development. Future work toward adapting the concepts and techniques presented and further considerations for improving reuse outside of this scope are addressed in the final chapter. Finally, it is also important to note that this work concentrates solely on the improvement of the source code; future work may (and should) include considerations within the context of more human-centric aspects of software development.

1.6 Limitations

Due to the scope of the project, currently known limitations are at present by design and described in “Delimitations”. As limitations were discovered during the design and development process they were captured and recorded.

1.7 Organization of the Study

This dissertation contains five chapters. This chapter provided the purpose, context, scope, and significance for this work. Chapter 2 reviews the literature, providing the necessary background in software development design patterns and concepts, and the state of current best practices. In Chapter 3, the methodology used for quantitative and qualitative evaluation in this study are presented. Chapter 4 presents the data collected from each case study alongside detailed analyses and evaluations. The last chapter, Chapter 5, functions as a recapitulation of the preceding chapters and provides the conclusions and contributions of the research
presented. Finally, recommendations are made for improvements and additions for future work.
CHAPTER 2. REVIEW OF THE LITERATURE

The improvement and integration of code reuse into software development practice has been a topic of study since the 1960s Brechner (2011). Up until the early 1990s, this work generally focused largely on code-based improvements, with the goal of providing higher levels of abstraction (Brooks, 1987). These efforts have been generally regarded as successful in reducing the semantic distance between software design concepts and their representation in code, particularly in object-oriented programming. However, benefits of this work did not had the impact expected as software projects greatly increased in size and complexity during this time (Cox, 1990).

In the last several years, efforts to improve code reuse have generally shifted from a purely code-based focus to higher-level concepts such as design patterns (Frakes & Kang, 2005). As the Legwork class library is based on improving reuse using this concept in particular, the rest of this chapter provides a discussion of design patterns and related programming concepts, and background information on current best practices for comparison.

2.1 Design Patterns

As stated by Johnson and Foote (1988) “one of the most important kinds of reuse is reuse of designs.” Design patterns (or just “patterns”) can be described as high-level solution templates with an established terminology built around promoting the use of good object-oriented software design (Millett, 2010). The ultimate value of patterns lie in the fact that they represent reusable, tested solutions, providing the developer with confidence in their effectiveness (Millett, 2010). Patterns have become such an important fixture in programming that many
experts determine code quality by how much of the code can be represented and understood as implementations of specific patterns (Riehle, 2009).

2.1.1 Changing the Timing of Reuse

Reusable assets are often created retroactively, for example, at the end of a project. The goal is to then use those assets on future projects. This practice has led to criticism that the benefits of reuse are only seen on subsequent projects, with the first project “paying the price” of the reuse. However, if we are able to create and use patterns within the same project the productivity benefits will be immediately seen in the same project in which the pattern was created (Ackerman, 2010).

2.1.2 Pattern Categories

Patterns can be grouped in many different ways; pattern categories reflect the different levels of scope, impact, and abstraction that patterns support (Ackerman, 2010). As more patterns have been formally established, patterns are now often grouped into the context of use (Fowler, 2002). Programmers can leverage pattern categories systematically, for example, so that larger-scale patterns are considered and selected in advance of those of smaller scope. Such an approach improves productivity as the selection of larger-scale patterns will influence and constrain our choice of smaller-scale patterns (Ackerman, 2010). Many patterns can also be combined to produce a synergistic effect. A “compound pattern” describes such a concept; the encapsulation of a number of patterns that work together within a larger pattern. In this context, pattern users are able to more easily and quickly use the combined set of patterns (Ackerman, 2010).
2.1.3 Patterns Are Abstractions

The goal of abstractions in programming is to allow the programmer to express design logic “without having to express large amounts of syntactic material that add no information content” (Liskov & Zilles, 1974). As stated by Johnson and Foote (1988), the most important attitude for a software developer is “the importance given to the creation of reusable abstractions”. Patterns can be seen as a reusable abstraction of software design as they can summarize a complex set of design and architecture elements and interactions (Ackerman, 2010). In this sense, patterns allow a higher-order expression of design and reduce complexity by providing abstract constructs of high level solutions in code.

2.1.4 Pattern Implementations

A pattern implementation provides value through effectively and consistently automating the reuse of a pattern in a particular environment. These pattern implementations can be seen as concrete tools for immediate use within the development environment (Ackerman, 2010). Using automation in the form of pattern implementations can significantly improve productivity and minimize complexity by reducing the depth of knowledge needed to successfully apply the pattern. There are three established forms of reusable pattern implementations; type-independent “templates” also called “generics,” class libraries, and frameworks.

2.1.4.1. Generics

First-order parametric polymorphism is a highly used feature included in modern programming languages such as C# and Java. These “generics” maximize code reuse and performance by allowing differing data types to use the same implementations without incurring the performance cost of casting or boxing operations (Microsoft, 2012). For example, to use a generic List with a collection of integers, we provide the int type as a parameter when we declare the variable.
We can reuse the same List class with strings simply by similarly providing a string type as a parameter. Behind the scenes, the language compiler creates a customized version of the list class for each type of element (Gamma, Helm, Johnson, & Vlissides, 1995). In the programming languages supported by the .NET Framework, generics are the most commonly used form of algorithmic standardization and reuse (Hazzard & Bock, 2011).

2.1.4.2. Class Libraries

Class libraries correspond to the types of tools commonly found in a programming platform’s support library. Each tool is designed to serve as a stand-alone application-independent component, though when used together they often collectively contribute synergistic functionality to help alleviate a large range of different programming problems (Johnson & Foote, 1988).

2.1.4.3. Frameworks

A framework can be defined as a collection of code components that provide solutions to a set of related problems specific to a particular kind of application (Johnson & Foote, 1988). In object-oriented programming, a framework typically makes extensive use of inheritance, and contains specialized classes and components designed specifically for inclusion in the framework. As such, using a framework commonly requires that the software developer know how a particular framework component is implemented in order to reuse it (Johnson & Foote, 1988). In other words, in contrast to class libraries and generics, “the framework itself sets the overall shape and tone of the software” (Biddle et al., 2003).
2.2 Fundamental Patterns and Related Concepts

While there are a multitude of patterns used in everyday object-oriented software design, this section provides an overview of common and reusable patterns and concepts for the expression of object interaction, presentation, and communication. Most of these design patterns and concepts are defined in the .NET Framework solely through interfaces, delegating a sizable responsibility to the developer to come up with concrete implementations (Fowler, 2006).

2.2.1 Object Interactions

Beginning with the highly influential work of Gamma et al. (1995), patterns have traditionally been described in classic object-oriented terminology, focusing on configuring specific instances of classes to achieve a specific purpose (Riehle, 2009). However, as applications have grown in size and complexity the focus and usage of design patterns has shifted to describe patterns instead terms of object interactions (Riehle, 2009).

2.2.1.1. Event Pattern

With the goal of simplifying object interactions and supporting loose coupling between classes, one of the most used abstractions in object-oriented software development is the Event Pattern (Purdy & Richter, 2002). This pattern views an object’s interaction with its world as transmissions of events in response to state changes (Fowler, 2006). This abstraction helps keep each component simple from a design and development perspective. All components need to know of their environment is the events they’re involved in, and need to “listen” for and respond to. Whenever a state change or anything else of interest happens components simply emit an event - they don’t need to know if anyone other components are listening. This enables programmers to concentrate on one component at a time with
well-defined inputs and outputs. As such, the great strength of the Event Pattern is that it affords a very loose coupling between its components (Fowler, 2006).

In an application that uses the Event Pattern, most of the code executes in reaction to events. For example, when the user clicks on a button, a change notification (a "message") is generated in the form of raising an event. The user interface thread then picks up the message and executes the button’s Click event handler. Additionally, the program may also invoke asynchronous calls that instruct the operating system to perform operations on a background thread and notify the program of the results of the operation. For example, using web services to request data in response to a button click and once it completes needs to display the result to the user interface. In all of these situations, program execution is controlled by notifications in the form of an event, and the application is concerned with reacting to them (Petricek, 2010).

While many user interface components contain pre-wired change notifications for common events such as the Click event on a button as given above, most often the developer must explicitly set up change notification events by hand. In the .NET Framework this is accomplished through implementing the INotifyPropertyChanged (INPC) interface within a class, thereby enabling properties within such classes to manually raise an event to publish change notifications (Microsoft, 2011c). With regard to asynchronous event-based programming, Microsoft provides detailed directions and implementations of the "Asynchronous Event Pattern" (Microsoft, 2011d; Richter, 2007) using a traditional "pull-based" model. Following best practices as described in detail in Microsoft (2011d) requires explicitly registering event handlers for any event that we want to pull data from before any execution takes place. We then call the asynchronous method and wait for the event to be raised and data to be returned (Microsoft, 2011d). For example, if a program makes an asynchronous call to a web service the Completed event will be raised when data is returned to the client from the web service.
2.2.1.2. Command Pattern

The Command Pattern is used to create objects that represent events in an application (Holpe & Woolf, 2003). A Command Object encapsulates an event and optimally contains enough contextual information to comprehend exactly what event has occurred. In the .NET Framework, the Command Pattern is defined by the well-known ICommand interface. Internally, the ICommand interface represents functionality very similar to an implementation of the Event Pattern, and user interface elements with commonly used events such as button’s Click event contain a Command property. If the Command property is set, the button will call the property’s Execute method when clicked.

The Command Pattern provides an abstraction layer for the handling of events by decreasing coupling (Brumfield et al., 2011). However, on their own Command Pattern objects do not have any knowledge of state, limiting usefulness in all but the simplest applications. For example, if we want to execute a method based on an interaction such as a button being pressed three times, we would need to keep track of number of times the button has been pressed in a separate ‘storage variable elsewhere in the class, complete with procedures to reset the “count” variable to zero after the button has been pressed three times. The inherent complexity involved escalates quickly when combinations of conditions must be met.

2.2.1.3. Mediator Pattern

The Mediator Pattern defines an object that provides central authority over a group of objects by encapsulating how these objects interact (Gamma et al., 1995). The Mediator Pattern promotes loose coupling by managing object interactions independently through a “publish-subscribe” pattern similar to the Observer Pattern (described later). This concept is often implemented in the form of a simple message-passing system, often referred to separately as the “Messenger
Pattern” or as the “Aggregator Pattern,” in both cases describing a system for processing many messages together as a whole (Holpe & Woolf, 2003).

2.2.1.4. Event Aggregator Pattern

Many applications developed for the life sciences require the combination of synchronous and asynchronous messages. Collecting information across these multiple messages and contexts can be very challenging from a programming perspective (Holpe & Woolf, 2003). The Event Aggregator Pattern is a specialized implementation of the Mediator Pattern, and acts by channeling events from multiple sources through a single object to simplify the management of object interactions (Brumfield et al., 2011).

2.2.2 Interactions in Object Collections

The two most used patterns for interacting with object collections are the Iterator and Observer patterns. The Iterator and Observer patterns are mathematical duals of each other. In other words, they represent a pairing between converse operators; the Iterator and the Observer.

2.2.2.1. The Iterator and Observer Patterns

The Iterator Pattern provides a mechanism for sequential access to the items of an object collection without revealing its underlying representation (Gamma et al., 1995). In other words, the Iterator design pattern abstracts access to a collection of objects by separating the interaction with objects in a collection from the traversal of these objects. In contrast, the Observer Pattern seeks to decouple those interested in an object’s state changes from the changing object completely (Brumfield et al., 2011) by defining a one-to-many relationship link
between parent and child objects. State changes within the parent object trigger automatic notifications and updates to child objects (Gamma et al., 1995).

2.2.2.2. Push vs. Pull Model

The push and pull models describe the two modalities of object interaction. In the scenario of an update based on a state change, the push model sends contextual information as part of the notification. In contrast, the pull model effectively waits for notification updates, at times requiring another request and an extra round trip to retrieve contextual information. As a result, the pull model generally requires more channels, more messages, and more code to manage (Hohpe & Woolf, 2003). From a pattern perspective, the Iterator Pattern follows a pull model, while the Observer Pattern follows a pull model.

2.2.3 Presentation Model Pattern

The Presentation Model Pattern is one of several user interface patterns that focus on keeping the logic for the presentation separate from the visual representation in the user interface. This is done to separate the concerns of visual presentation from that of visual logic, which helps improve maintainability, testability, and reuse (Brumfield et al., 2011). The two most used presentation patterns in software development at present are based on the Presentation Model; Model View Controller (MVC) used in Java and ASP.NET web applications (Lloyd, Rimov, & Hamel, 2004), and Model View ViewModel (MVVM) used in the .NET Framework (Smith, 2009) as well as JavaScript programming (Sanderson, 2012).

In the .NET Framework, both MVVM and MVC depend on the Event Pattern to provide loose coupling (that is, separation) of presentational aspects (the View) from the underlying data classes (the Model) and business logic (the Controller or ViewModel). This loose coupling between objects allows for changes in one component to have less of an unintentional impact on the entire system, an
important characteristic in object-oriented software design and in enabling code reuse. As seen in Figure 2.1 above, both MVC and MVVM depend on user interface and INPC implementations of the Event Pattern and the Command Pattern for communication (Petzold, 2009).

2.2.4 Plugin Pattern

The Plugin Pattern is a common pattern often used when an application requires different implementations of a particular behavior. The Plugin Pattern facilitates this requirement by providing centralized run-time configuration (Fowler, 2002). From a programming perspective, the Plugin Pattern creates a concrete object instance from an interface at run-time. This differs from class inheritance, where behavior is altered or overwritten, or configuration, where behavior modification is limited to the capabilities of the predefined configuration options (Brumfield et al., 2011).
2.2.5 Method Chaining Pattern

The Method Chaining Pattern requires modifier methods for an object to return the host object so that multiple modifiers can be invoked in a single expression resulting in syntax that seems to flow like its own language (Fowler & Parsons, 2011). The Method Chaining Pattern is often used when designing configuration or computation libraries, and can greatly improve the readability and expressiveness of commonly used blocks of code (Fowler & Parsons, 2011).

2.2.6 Monads

Monads, also called computation expressions, are a programming concept that has been useful in functional programming languages (Petricek, 2010). While not officially recognized as such, monads are essentially a design pattern describing relationships between functions. From a high-level perspective, a monad precisely expresses an abstraction of sequential computations, allowing the encapsulation of computational side effects (Moggi, 1991; Wadler, 1990, 1995). Recent work has extended the concept of a monad to practical applications in programming, such as in creating composable continuations (Atkey, 2009). However, while functional programming languages have seen great success with the use of monads, the type system of object-oriented platforms such as Java and .NET are not expressive enough to support monads directly.

2.2.7 Language Integrated Query

Language Integrated Query (LINQ) is an extensive software library built into the .NET Framework based on the concept of monads described previously. LINQ provides a uniform way to query collections of objects and relational data (Meijer & Beckman, 2006). The major advantage of LINQ is its use of generic extension methods, providing the ability to add monadic methods to any collection of object without needing to define these methods in the original code (Meijer, 2007).
One of the most distinctive characteristics of LINQ is its similarity to an internal domain-specific language, also called a “Fluent Interface” (Fowler & Parsons, 2011). This is achieved through the Method Chaining Pattern described previously. For example, if we require filtering, sorting, and projection on an array of strings, we may use the LINQ-provided `Where`, `OrderBy`, and `Select` extension methods, respectively, in succession.

```
1: string[] names = { "John", "Bob", "Harry", "Mary", "Jay" };
2: var query = names.Where (name => name.Contains("a"))
   .OrderBy (name => name.Length)
   .Select (name => name.ToUpper());
3: // Final value of 'query' is { "JAY", "MARY", "HARRY" }
```

Figure 2.2. Using LINQ for Filtering, Sorting, and Projection

Figure 2.2 above specifies a query that sequentially calls the `Where`, `OrderBy`, and `Select` extension methods using lambda expressions to specify the parameters. This standardized syntax enables composability that makes queries much easier to author and understand (Meijer, 2007).

2.2.8 Reactive Extensions

New to version 4.0, the .NET Framework contains a definition for push-based event patterns using the Observer Pattern through the `IObservable` interface (Microsoft, 2011a). This interface is the counterpart to the Iterator Pattern, defined through the `IEnumerable` interface, and supports the same underlying functionality programmers have previously used in traditional pull-based programming. Supporting this new programming model in the .NET Framework is Microsoft’s Reactive Extensions (Rx), a software library of sequence operators and
combinators for composing asynchronous and event-based programs using observable sequences (Microsoft, 2011b). The Rx library has seen rapid and widespread adoption within the .NET community, and has also been ported to many other languages and platforms (see Table 2.1).

Table 2.1
Implementations of the Rx Library in Other Languages and Platforms

<table>
<thead>
<tr>
<th>Platform or Language</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript</td>
<td>RxJS</td>
</tr>
<tr>
<td>Clojure</td>
<td>cljque</td>
</tr>
<tr>
<td>Objective C</td>
<td>ReactiveCocoa</td>
</tr>
<tr>
<td>Mono.NET</td>
<td>MonoReactive</td>
</tr>
<tr>
<td>Dart</td>
<td>Reactive Dart</td>
</tr>
<tr>
<td>Java</td>
<td>Reactive 4 Java</td>
</tr>
</tbody>
</table>

2.3 Current Best Practices

For the last decade Microsoft’s Patterns and Practices team has been tasked with providing best practices on how to design and develop applications using Microsoft’s software development platforms. For desktop development, the Patterns and Practices team provides “Prism”, a framework designed to assist developers in building applications (Brumfield et al., 2011). Prism contains over 8,900 lines of code and provides “best practice” implementations of 14 different design patterns.

While described as “best practices” using frameworks such as Prism have several potential issues. First, there is a significant investment of time required. In fact, the documentation for Prism explicitly states: “...the project deadline must accommodate an investment of time up front” before starting to use Prism.
(Brumfield et al., 2011). Second, frameworks such as Prism necessitate a “complete buy-in” requiring developers to use techniques and patterns that they may not need or even want to include in their software (Brumfield et al., 2011). And third, as mentioned previously, frameworks such as Prism are designed to make programming work easier on the developer, but do not directly address code reuse.

2.4 Reuse Integration

An important aspect of reuse is selecting the most applicable abstractions and concepts for problem domains (Ko et al., 2011). Scientific analyses commonly involve a series of computational tasks and processes, also called a workflow or pipeline. Scientific workflows focus on data-flow, and typically contain chained data transformations. Paraphrasing Brechner (2011), this is an example where “imagining parts as class instances fails as an analogy” as these transformations are much more easily expressed and reused using concepts from functional programming. For example, as described previously many functional programming languages use a specific type called a monad to support the composition of computation expressions. However, functional languages in turn lack important programming concepts such as mutable state required in object-oriented programming. Supporting reuse therefore requires a hybrid of both functional and object-oriented programming concepts and constructs.

In addition to abstraction, effective reuse also necessitates code “isolation” (or encapsulation) considerations (Brechner, 2011). For example, a very common scenario in scientific data processing is composing several web services together into a data processing pipeline (Sayers & Miller, 2010). However, using the Event Pattern to execute asynchronous methods sequentially requires nesting complex code structures in order to chain the output of one method to another. This results in very tightly-coupled code and potentially causes unwanted side effects (Brumfield et al., 2011). Though there have been various attempts to simplify asynchronous
programming, best practices do not provide a complete solution as they require significant code re-writing, ad-hoc data types, lack customizability, and are generally difficult to use (Richter, 2008; Skeet, 2010).

Another important aspect toward facilitating reuse is to reduce the distance and friction between the developer’s intent for the software and the realization of that intent in code (Miller, 2009). This again is often addressed through abstraction. However, in many cases the complexity and boilerplate code required to implement the abstraction actually contributes to the problem instead (Kiselyov & Shan, 2012). A disproportionate amount of time is then spent wrestling with purely technical concerns instead of addressing the core problems the software is meant to solve (Miller, 2009). For example, implementing the Event Pattern using INPC is not technically difficult, but doing so adds “syntactic noise” and in large or more complex classes often obscures the original intent of the code (see Figure 2.3 below) (Perry, 2009).

![INPC Implementation](image)

```csharp
public class MyClass
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set
        {
            _name = value;
            OnPropertyChanged("Name");
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Figure 2.3. Converting a Property to Support Change Notifications Using the Event Pattern and INPC

Writing implementations such as INPC are monotonous and error-prone as they depend on a string that must match the name of the property (see bold code in
Figure 2.3 above) and are not included in any automatic code refactoring (McCarter, 2010). A simple typing error, forgetting to change the value of the string when changing the name of the property, and unrealized mistakes made during “copy and paste programming” are so common in software development that abstractions using this problematic pattern are often said to suffer from the “Magic Strings Problem” (Hazzard & Bock, 2011) and is even found in current best practice offerings such as Prism. This problem is particularly dreaded by developers as misspelling this string does not cause an error during compile-time or run-time, making debugging such code particularly difficult. With applications commonly containing tens or hundreds of Model classes alone implementations of INPC and similar event-based notification mechanisms especially play a critical role in enabling reuse in software design and development (McCarter, 2010).

2.5 Summary

In this chapter we covered many different and important concepts that affect reuse; design patterns, programming concepts from object-oriented programming and functional programming, several foundational frameworks and software libraries, as well as the current state of best practices and motivating examples of areas that need work. In the next section, we discuss the methodology used to evaluate the comparative impact of using Best Practices and Legwork to on code reusability.
CHAPTER 3. METHODOLOGY

This chapter reviews the methodology used for evaluation. The goal of the evaluation process was to assess the effectiveness of using Legwork code on improving reuse as compared to current best practices. From a high-level perspective, we employed a case study approach with quantitative and qualitative “pre-post comparisons” as described by Unterkalmsteiner et al. (2011) of code produced using Best Practices against code produced using Legwork.

3.1 Experimental Framework

Two case studies were used in evaluation. First, “Entrez Applications” (EA) consisting of six data pipeline applications leveraging the Entrez database web services provided by the National Center for Biotechnology Information (NCBI). Second, the “.NET Bio” case study, consisting of the “Bio Sequence Visualizer” demonstration application leveraging the .NET Bio toolkit.

3.1.1 Entrez Applications Case Study

“Entrez” is a database retrieval system providing public access to 35 different databases provided by NCBI (Sayers et al., 2012). The Entrez system has an Application Programming Interface (API) called the “Entrez Programming Utilities”, or just “E-Utilities”. The E-Utilities in turn include eight web services that can be used to search, link and download data from any of the Entrez databases. While potentially useful by themselves, the full potential of E-Utilities is realized when successive web service calls are combined to create a data pipeline (Sayers & Miller, 2010). Supplementing the E-Utilities-based data pipelines
provided by Sayers and Miller (2010) with additional functionality, six “Entrez Applications” were designed (see Table 3.1).

### Table 3.1
Programs Included in the “Entrez Applications” Case Study

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Pipeline Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database Links</td>
<td>Retrieve links in database A to records in database B matching a list of unique identifiers</td>
</tr>
<tr>
<td>Text DataDocs</td>
<td>Retrieve data records and document summaries matching an Entrez text query</td>
</tr>
<tr>
<td>ID Docs</td>
<td>Retrieve document summaries matching a list of unique identifiers</td>
</tr>
<tr>
<td>ID Linked DataDocs</td>
<td>Retrieve data records and document summaries from database A linked to records from database B matching a list of unique identifiers</td>
</tr>
<tr>
<td>Text Linked DataDocs</td>
<td>Retrieve data records from database A linked to records from database B matching an Entrez text query</td>
</tr>
<tr>
<td>Spelling DataDocs</td>
<td>After 3s idle time or F1 key is pressed, retrieve spelling suggestions, PubMed IDs, and data records</td>
</tr>
</tbody>
</table>

#### 3.1.2 .NET Bio Case Study

The “.NET Bio” project is an open source library of bioinformatics tools and functions, primarily targeting genomics. Similar to BioJava (Holland et al., 2008) and BioPerl (Stajich et al., 2002), .NET Bio includes: parsers for common
bioinformatics file formats; algorithms for analysis and transformation of DNA, RNA, and protein sequences; and connectors to biological web services, such as NCBI BLAST. According to the project’s statistics pages there have been over 19,200 downloads of the .NET Bio library, and at over 210,000 lines of code. .NET Bio is likely both the largest and most used code library in .NET Framework-based software development for the life sciences.

While containing useful code for data analysis and computation, like many other bioinformatics toolkits, the .NET Bio library itself does not contain any graphical support. To assist developers in “bridging the gap” to creating graphical (and more practical) software, several open source applications are provided to demonstrate integration patterns and exemplify best practices when using .NET Bio. For example, the “Bio Sequence Visualizer” application provides visualizations and analysis of genomic data. The developers of this application accomplished this by using Microsoft Prism alongside .NET Bio to manage and coordinate data flow and component interactions.

3.1.3 Case Study Procedure

The goal of the program implementation process was to obtain two functionally identical implementations of both case studies, one set using Best Practices and one using Legwork, to enable a pairwise comparison and assessment. Using Prism’s design pattern implementations identified in Brumfield et al. (2011) and following Microsoft’s self-described “best practices” instructions (Microsoft, 2011d) the “Best Practices” implementations of the EA case study was written. For the .NET Bio case study, the code provided by Microsoft was used directly. Both Best Practices implementations were then verified to function as intended using unit tests.
Based on these two Best Practices implementations, a “Pattern and Concept Map” was created to assist in linking concept and pattern implementations found in the Best Practices to those provided by Legwork (see Table 3.2 below).

<table>
<thead>
<tr>
<th>Name</th>
<th>Best Practices</th>
<th>Legwork</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous Event Pattern</td>
<td>Defined via APM</td>
<td>Servable</td>
</tr>
<tr>
<td>Event Pattern</td>
<td>NotificationObject</td>
<td>Servable</td>
</tr>
<tr>
<td>Event Aggregator Pattern</td>
<td>EventAggregator</td>
<td>DataStream</td>
</tr>
<tr>
<td>Command Pattern</td>
<td>DelegateCommand</td>
<td>ObservableCommand</td>
</tr>
<tr>
<td>ViewModel Base</td>
<td>NotificationObject</td>
<td>ProxyViewModel</td>
</tr>
</tbody>
</table>

Using the completed Best Practices case study applications as starting points, Legwork implementations were substituted for best practice implementations based on the mappings provided in the above table. Both the Entrez Applications and .NET Bio Legwork implementations were then also verified to have the same functionality as the Best Practices implementations by passing the same unit tests mentioned previously.

3.2 Evaluation Methodology

Following the program design categorization in McConnell (2004), the evaluation of reuse was considered at the class, subsystem, and system levels. We employed a quantitative evaluation at the class level, and qualitative evaluation methods at the subsystem and system program levels.
3.2.1 Class Level Reuse Evaluation

Class level evaluation was a metric-based assessment and considered measurements of class-level reuse metric values between the Best Practices and Legwork implementations of the case studies. Class level evaluation had two parts: comparing and analyzing the effect of substituting Legwork implementations on mean metric values within each case study.

3.2.1.1. Metric Selection

Previous work has demonstrated the effective use of several metrics for the evaluation of the reusability of source code in object-oriented software (Capiluppi & Boldyreff, 2008; Gorton & Zhu, 2005). Among the many metrics supported, those selected have proven to be particularly useful for reuse evaluation with regard to both high level software composition and low level object coupling and cohesion (Capiluppi & Boldyreff, 2008; Gorton & Zhu, 2005). Table 3.3 contains the names, descriptions, references to previous work demonstrating the usage of these metrics in evaluating code reuse, and the interpretation of the metrics.
Table 3.3
Quantitative Metrics Used in Measuring Code Reusability

<table>
<thead>
<tr>
<th>Name</th>
<th>Description and Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efferent Coupling (EC)</td>
<td>Number of dependencies on other classes (Smacchia, 2012). Lower values indicate better reusability (Gorton &amp; Zhu, 2005; Kean, 2007).</td>
</tr>
<tr>
<td>Cyclomatic Complexity (CC)</td>
<td>Number of decisions that can be taken in a procedure (Watson &amp; McCabe, 1996). Lower values indicate better reusability (Smacchia, 2012).</td>
</tr>
<tr>
<td>Lack of Cohesion of Methods (LCOM)</td>
<td>Percentage of methods that do not access a class variable averaged over all variables in the class (Henderson-Sellers et al., 1996). Lower values indicate better reusability (Sandhu et al., 2009).</td>
</tr>
<tr>
<td>Lines of Code (LOC)</td>
<td>The number of lines in the source code, excluding spaces and comments (Dandashi, 2002). Lower values indicate better reusability (Smacchia, 2012).</td>
</tr>
</tbody>
</table>

3.2.1.2. Data Collection and Analysis Using NDepend

As outlined in Table 1.1, code reuse metric measurements were acquired through static analysis of source code files using the “NDepend” code analysis tool (Smacchia, 2012). The NDepend software was specifically selected for two reasons.
First, NDepend natively supports the C# programming language and the .NET Framework (de Souza Pereira Moreira, Mellado, Montini, Dias, & Marques da Cunha, 2010). Second, NDepend provides automated measurement for each of the reuse metrics selected (provided in Table 3.3) each having been previously demonstrated to be effective in conducting code reusability evaluations (Gorton & Zhu, 2005; Smacchia, 2012).

Table 3.4 contains the procedure for extracting code metric measurements using the NDepend software. With two case studies and two code versions (Best Practices and Legwork) this procedure was executed a total of four times, producing four separate HTML report files. Microsoft Excel was then used to import the data from each HTML report file for subsequent statistical analysis.

Table 3.4
Procedue to Extract Code Metric Measurements Using NDepend

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Select the “New Project...” menu option</td>
</tr>
<tr>
<td>2</td>
<td>Select the “Code to Analyze” tab</td>
</tr>
<tr>
<td>3</td>
<td>Click “Add Assemblies in Folder” and add code files to the project</td>
</tr>
<tr>
<td>4</td>
<td>Select the “Report” tab</td>
</tr>
<tr>
<td>5</td>
<td>Select the “Use Standard Report” option</td>
</tr>
<tr>
<td>6</td>
<td>Under “Report Sections” select only the “Types Metrics” option</td>
</tr>
<tr>
<td>7</td>
<td>Click the “Run Analysis on Current Project” button</td>
</tr>
</tbody>
</table>

3.2.2 Subsystem Level Reuse Evaluation

Subsystem design was assessed by examining the control flow of the Legwork and Best Practices implementations of the ELinkService subsystem (component)
from the EA case study. As with all “Service” components in the EA case study, the ELinkService component is responsible for several actions in making calls to an E-Utilities web service. During runtime, the code for the Execute method used to trigger these actions was analyzed to determine execution sequence. A code control flow diagram was then created for each implementation corresponding to the execution sequence.

3.2.3 System Level Reuse Evaluation

In the context of reuse at the system level, the ability to express intent through code composition and interaction is an important consideration. To evaluate this ability, we examined the Best Practices and Legwork code for the “Text Linked DataDocs” data pipeline from the EA case study (see Table 3.1). While this represents a single point of assessment, it is critical to understand that this evaluation was from the perspective of pattern implementations; these implementations are repeated throughout both case studies. The “Text Linked DataDocs” data pipeline was specifically selected as it contained implementations of all design patterns and concepts evaluated.

First, an activity diagram of the workflow for the “Text Linked DataDocs” data pipeline was created from the perspective of the ViewModel class. The corresponding code from the Legwork and Best Practices implementations was then identified. Sequence diagrams were generated from source code using Visual Studio 2010’s built in functionality. The options used to generate this diagram were: “Call Level” set to “2,” the option “Include Solution and External References” was enabled, and the option “Exclude Properties and Events” was also enabled. Finally, visual linkages between the activity diagram, code, and sequence diagrams were added through color coding.
3.3 Hypothesis Evaluation

As stated in the introduction, the objective of this work is to present evidence in support of rejecting the Null Hypothesis and accepting the Alternative Hypothesis in the context of a mixed quantitative and qualitative evaluation. Referring back to the hypothesis in the introduction, we define an improvement in reuse as meeting three key criteria, given in Table 3.5. We make the case that these three criteria collectively provide ample evidence to reject the Null Hypothesis and accept the Alternative Hypothesis.

Table 3.5

<table>
<thead>
<tr>
<th>Program Level</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Level</td>
<td>For each case study, an improvement in mean metric measurements of the source code between the Best Practices and the Legwork implementations at a significance level of 5%.</td>
</tr>
<tr>
<td>Subsystem Level</td>
<td>Demonstrated improvement in code control flow, illustrated through “Code-Control Flow” diagram comparisons.</td>
</tr>
<tr>
<td>System Level</td>
<td>An improvement in expressing logical intent through code composition, demonstrated by a reduction in semantic distance. Logical intent was captured in an activity diagram from functional requirements and compared to sequence diagrams generated from source code.</td>
</tr>
</tbody>
</table>
3.4 Summary

In this chapter we reviewed the design of the study, data collection process, and the criteria used to evaluate the hypothesis. In the next chapter, the data is presented and analyzed.
CHAPTER 4. DATA ANALYSIS

In this chapter, we present the results obtained by executing the methodology outlined in the previous chapter. The data is presented from a bottom-up perspective; first the quantitative class level data, and then moving to the qualitative data at the subsystem level and system level.

4.1 Class Level

Table 4.1 contains the number of times Legwork code was used in the two case studies. The Entrez Applications Case Study code contained a high level of object interactions using the Event Pattern, calls to web services using the Asynchronous Event Pattern, and required composition using the Support Library. The .NET Bio Case Study code used the Event Aggregator Pattern, and, perhaps as a result, contained comparatively fewer object interactions using the Event Pattern.

<table>
<thead>
<tr>
<th></th>
<th>Entrez Applications</th>
<th>.NET Bio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-Based Patterns</td>
<td>44</td>
<td>21</td>
</tr>
<tr>
<td>Event Aggregator Pattern</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>Command Pattern</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>ViewModel Base</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Support Library</td>
<td>24</td>
<td>0</td>
</tr>
</tbody>
</table>
From a measurement perspective, the Entrez Applications Case Study contained 15 classes with 4 reuse metric measurements for each class, giving 60 total metric measurements. The .NET Bio Case Study contained 21 classes with 4 measurements for each class, giving 84 total metric measurements. Substituting Legwork implementations for Best Practices implementations in the Entrez Applications and .NET Bio case studies reduced 100% and 92% of the total measurements recorded, respectively. Between both case studies there was an improvement in 141 of 144 total measurements.

Figure 4.1 and Figure 4.2 show a summary of the class level data from the Entrez Applications and .NET Bio case studies, respectively (see Appendix A for the raw data). The inner quartiles of the measured range are displayed as boxes separated by a dark line indicating the median value, and overlayed by a blue diamond marker indicating the average value. The total width of the boxes indicates the interquartile (IQ) range. For each metric, the outer whiskers represent the range of data within 3 IQ of the median, and the red circular markers indicate any outlier data points beyond 3 IQ of the median.

Despite representing two different types of programs, the data from both case studies generally follow the same trends. First, and most relevant to the hypothesis, the mean measured values for Legwork are less than that of Best Practices for each metric measured. The magnitude of improvement appears to correlate to the number of code substitutions provided in Table 4.1. Second, the means and medians for each metric are close in value for both Legwork and Best Practices, indicating relatively symmetrical value distributions. Third, while a reduction in the data spread and skew is noticeable between Legwork and Best Practices, the overall shape of the data appears to be conserved.
Figure 4.1. Metric measurement summaries for the Entrez Applications Case Study ($N=15$). Means are represented as blue diamonds, outliers are represented as red circles. For each metric, lower values indicate better code reusability.
Figure 4.2. Metric measurement summaries for the .NET Bio Case Study (N=21). Means are represented as blue diamonds, outliers are represented as red circles. For each metric, lower values indicate better code reusability.
For each case study, the mean improvement for each metric was calculated and then tested by a paired t-test. Table 4.1 contains these data for the Entrez Applications and .NET Bio case studies. For both case studies, all mean improvement calculations resulted in positive numbers, indicating improvements in all mean metric measurements. For the Entrez Applications Case Study, improvements in mean values are significant at the 0.001 level, while improvements in mean values for the .NET Bio Case Study met at least a significance level of 0.05. This evidence meets the class level criteria for supporting our hypothesis that the use of Legwork improves code reusability.

Table 4.2
Effect on Mean Metric Measurements

<table>
<thead>
<tr>
<th>Metric</th>
<th>Mean Improvement</th>
<th>Entrez Applications</th>
<th>.NET Bio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclomatic Complexity</td>
<td>55.6%***</td>
<td>38.6%*</td>
<td></td>
</tr>
<tr>
<td>Efferent Coupling</td>
<td>33.0%***</td>
<td>22.0%***</td>
<td></td>
</tr>
<tr>
<td>Lines of Code</td>
<td>58.8%***</td>
<td>26.4%*</td>
<td></td>
</tr>
<tr>
<td>Lack of Cohesion of Methods</td>
<td>94.1%***</td>
<td>57.8%**</td>
<td></td>
</tr>
</tbody>
</table>

*Significant at p<0.05, **Significant at p<0.01, ***Significant at p<0.001

4.2 Subsystem Level

In this section, we present the code and control-flow for a “Service” subsystem (component) from the Entrez Applications Case Study. Aside from the names of the web service and the corresponding type of the Model class used to store the result of the web service, the code for each Service component is identical.
For this evaluation, however, we considered code for the \texttt{ELinkService}, responsible for calling the "ELink" E-Utilities web service.

Each Service component is responsible for completing the workflow in calling a web service, provided in Table 4.3. To codify the functionality described, Best Practices dictate the combination of three design patterns; the Asynchronous Event Pattern in step 2, the Event Pattern in step 3 and step 6, and the Event Aggregator Pattern in step 5.

Table 4.3

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initialize web service client and declare a new web service request</td>
</tr>
<tr>
<td>2</td>
<td>Asynchronously invoke the web service</td>
</tr>
<tr>
<td>3</td>
<td>Notify the user interface of a status change (from &quot;Ready&quot; to &quot;Busy&quot;)</td>
</tr>
<tr>
<td>4</td>
<td>Store the result of the web service call in a new \texttt{Model} class</td>
</tr>
<tr>
<td>5</td>
<td>Pass the new \texttt{Model} class to interested parties</td>
</tr>
<tr>
<td>6</td>
<td>Notify the user interface of a status change (from &quot;Busy&quot; back to &quot;Ready&quot;)</td>
</tr>
</tbody>
</table>

Figure 4.3 contains the code for executing the workflow in Table 4.3 using Legwork. Lines 1 and 2 represent initializing the web service client and declaring a new web service request, respectively. As we can see in line 3, asynchronously invoking the web service, appropriately setting the status of the \texttt{ELinkService} class, and creating the new \texttt{ELinkModel} are all neatly encapsulated in a single line of code. The code is also easy to follow logically as it executes sequentially during run time as indicated by the blue arrows in the diagram.
Figure 4.3. Code-Control Flow diagram for calling a web service using Legwork. Execution sequence and control flow is indicated to the left of the code. The blue arrows indicate normal, sequential control flow.

```javascript
1: var client = new eUtilsServiceSoapClient();
2: var linkReq = new eLinkRequest();
   .StateChange(() => Status.Set('Busy'), () => Status.Set('Ready'))
   .Subscribe(result => ELinkResult.Set(new ELinkModel(result)));
4: return ELinkResult;
```

Figure 4.4 contains the original code from Best Practices. Lines 1 and 2 represent initializing the web service client and declaring a new web service request, respectively. Line 3 starts the code for the Asynchronous Event Pattern where we declare the “call back”, that is, the predefined code we want executed when the results are returned from the web service. In line 13, the web service is called asynchronously. When the web service returns with a result, the call back code provided in lines 4-11 is executed.

In passing the result of a web service call (that is, the creation of a new Model class) the Event Aggregator Pattern must be used within the Completed event handler of the asynchronous web service client. Before the Event Aggregator Pattern can be used, an extra Prism-specific class must be created that derives from Prism's CompositePresentationEvent class (Brumfield et al., 2011). These often “empty” derived classes must be created for every event. The corresponding class from Figure 4.4 is given in Figure 4.5. With this class in place, moving back to Figure 4.4, line 8 contains the line of code needed to use the Event Aggregator Pattern to publish the new ELinkModel class created in Line 7.
Figure 4.4. Code-Control Flow diagram for calling a web service using Best Practices. Execution sequence and control flow is indicated. Blue arrows indicate normal, sequential control flow. Red arrows indicate the interruption of normal control flow, or non-sequential "jumps" in code execution.

```csharp
1: var client = new eUtilsServiceSoapClient();
2: var linkReq = new eLinkRequest();
3: client.run_eLinkCompleted += (sender, e) =>
4: {
5: if (e.Result != null)
6: {
7: ELinkResult = new ELinkModel(e.Result);
8: EventAgg.GetEvent<ELinkCompletedEvent>().Publish(ELinkResult);
9: }
10: Status = "Ready";
11: }
12: Status = "Busy";
13: client.run_eLinkAsync(linkReq);
```

Figure 4.5. Usage of Prism’s Event Aggregator Pattern requires creating an additional Prism-derived class for each event.

Comparing the Legwork code and control flow in Figure 4.3 to the Best Practices code and control flow in Figure 4.4 clearly indicates a reduction in the number of interruptions in normal code control flow. This evidence meets the
subsystem level criteria for supporting our hypothesis that the use of Legwork improves code reusability.

4.3 System Level

Particularly in the context of code reuse, the ability to express intent through composition is an important consideration. In this section, we present the relevant system level data for the “Text Linked DataDocs” data pipeline from the Entrez Applications case study. As provided in Table 3.1, the “Text Linked DataDocs” functionality was originally described by Sayers and Miller (2010) as retrieving data records through linking two databases together based on an initial Entrez text query. While having many potential usages, for this section we describe the scenario of extracting DNA sequences from NCBI’s “Nucleotide” database based on a text search of publications in the “PubMed” database. As depicted in Table 4.4 the data pipeline for this scenario requires the sequential execution of three different web services.

Table 4.4
Workflow Used for “Text Linked DataDocs” Data Pipeline

<table>
<thead>
<tr>
<th>Step</th>
<th>Web Service</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ESearch</td>
<td>Retrieve the PubMed IDs of search “hits” from PubMed</td>
</tr>
<tr>
<td>2</td>
<td>ELink</td>
<td>Retrieve Nucleotide IDs corresponding to each PubMed ID</td>
</tr>
<tr>
<td>3</td>
<td>EFetchSeq</td>
<td>Retrieve data records from Nucleotide corresponding to each Nucleotide ID</td>
</tr>
</tbody>
</table>

Figure 4.6 is a high-level activity diagram corresponding to the execution of the “Text Linked DataDocs” data pipeline. Execution of the data pipeline is triggered by the user interface through the ViewModel class, and consists of
sequential calls to three “Service” components: `SearchService`, `LinkService`, and `FetchSeqService` which in turn make calls to the “ESearch”, “ELink”, and “EFetchSeq” web services, respectively. The result of each call to a web service is stored within a web service-specific `Model` class and serves as input into the next web service call. We can see in the activity diagram that the optimal execution is step by step, sequential, and only returns the end result to the `ViewModel` class to be displayed in the user interface.

![Activity diagram](image)

Figure 4.6. Activity diagram for “Text Linked DataDocs” data pipeline.

Figure 4.7 and Figure 4.8 contain the code representing the logic in the activity diagram in Figure 4.6 using LegoWork and Best Practices, respectively. Both figures also contain sequence diagrams generated from runtime execution of the code presented. For clarity, the code and sequence diagrams have been color coded to match the logic from the activity diagram.

Figure 4.7 contains the logic from the activity diagram implemented in Figure 4.6 using LegoWork. Within a single line of code, each Service component is seamlessly linked together, following a well-defined “Execute-Subscribe” pattern.
The code precisely follows the logic depicted in the activity diagram, and only returns the end result of the data pipeline to the ViewModel class.

Figure 4.8 contains the code required to implement the logic from the activity diagram in Figure 4.6 using Best Practices. In order to link Service components together, the Best Practices implementation requires the use of the Event Aggregator Pattern seen in line 2 and line 4. While we can see the “Execute-Subscribe” pattern in the code, each call to an “Execute” method requires the creation of several temporary data storage objects in order to transfer the result of one Service call over as the input to another Service call. The Best Practices code also requires a “call back” to the ViewModel class after each call to a web service. These call backs interrupt the data pipeline workflow, and represent a departure from the ideal execution plan provided in the activity diagram.

While clearly visible in the code, a comparison of the Legwork sequence diagram in Figure 4.7, the Best Practices sequence diagram in Figure 4.8 with the data pipeline activity diagram in Figure 4.6 clearly indicates a reduction in semantic distance. Simply stated, in contrast to the Best Practices code, Legwork code does not require call backs nor temporary values in order to complete all three of the web services calls in sequence, which much more closely aligns with the workflow as described in Table 4.4, and as depicted in the activity diagram. This evidence meets the system level criteria for supporting our hypothesis that the use of Legwork improves code reusability.
Figure 4.7. Legwork code and sequence diagram from the “Text Linked DataDocs” data pipeline from the Entrez Applications Case Study. The code and diagram have been color coordinated to match the logic provided in the activity diagram in Figure 4.6.
1: SearchService .Execute(DbFrom, Query, "n","1");
2: var searchEvent = EventAgg .GetEvent<ESearchServiceCompletedEvent>();
3: var postToken = searchEvent .Subscribe(SearchModel => LinkService
 .Execute(DbFrom, DbTo, SearchModel.Ids));
4: var linkEvent = EventAgg .GetEvent<ELinkServiceCompletedEvent>();
5: var linkToken = linkEvent .Subscribe(LinkModel => FetchSeqService
 .Execute(DbTo, LinkModel.FetchIds));

Figure 4.8. Best Practices code and sequence diagram from the "Text
Linked DataDocs" data pipeline from the Entrez Applications Case Study.
The code and diagram have been color coordinated to match the logic
provided in the activity diagram in Figure 4.6.
4.4 Hypothesis Evaluation

In this section, we present the evaluation of the hypothesis. Recall from Chapter 3 there are 3 criteria that must all be met, one for each program design level:

- **Class Level**: An improvement in mean metric measurements of the source code between the Best Practices and the Legwork implementations at a significance level of 5%.

- **Subsystem Level**: Demonstrated improvement in code control flow, illustrated through “Code-Control Flow” diagram comparisons.

- **System Level**: An improvement in expressing logical intent through code composition, demonstrated by a reduction in semantic distance.

At the class level, Legwork resulted in improvements in means ranging from 22.0% to 94.1%, with each improvement meeting at least a significance level of 5% (see Table 4.1). At the subsystem level, Legwork improved code control flow by effectively removing the need for “code jumps” present in Best Practices code (see Figure 4.3 and Figure 4.4). At the system level, Legwork enabled code composition more closely aligned with the logical intent by eliminating the need for code callbacks and ad hoc temporary storage variables (see Figure 4.7 and Figure 4.8).

Based on the above, using Legwork meets each of the 3 established criteria defining an improvement in code. We are therefore able to reject the Null Hypothesis, and accept the Alternative Hypothesis.

4.5 Summary

Chapter 4 presented the results obtained by executing the evaluation methodology outlined in the previous chapter. Using our previously defined criteria, the results were analyzed to determine whether code reusability was improved by using Legwork at the class, subsystem, and system program levels.
CHAPTER 5. CONCLUSIONS AND FUTURE WORK

The focus of this dissertation was on improving conditions for the reuse of code in software development for the life sciences. In Chapter 1, we introduced the study and presented Legwork, a software library designed to improve code reusability over current Best Practices. In Chapter 2, we discussed necessary background information and current work. In Chapter 3, we established the evaluation methodology, and defined an improvement in reuse as meeting three criteria; improvements in code reusability at the class, subsystem, and system program levels (see Table 3.5). In Chapter 4 we presented the data and analysis resulting from executing the evaluation methodology. In this chapter, we discuss two main conclusions drawn from the findings and provide direction and recommendations for future work.

5.1 The Impact of Software Composition Model on Code Reuse

Software development work supporting life sciences research is usually data-centric and involves composing a variety of data retrieval, computation, and visualization components (Sayers & Miller, 2010). While some overlap can occur, applications are most commonly designed and developed using either a module-based model of composition or a workflow-based model of composition (Bowers, Ludascher, Ngu, & Critchlow, 2006). Both models are represented by the two case studies: the .NET Bio Case Study was primarily module-based, while the Entrez Applications Case Study instead used a workflow-based model. As seen in Table 4.1, though both case studies indicated improvements by using Legwork instead of Best Practices, the Entrez Applications Case Study improved by almost
twice as much. This difference in impact can be attributed to the different models of composition used.

5.1.1 Module-Based Composition

Compared to the workflow-based model, the module-based model of composition is often described as more “control-flow intensive”, as connecting independently created code modules commonly requires additional sub-processes to align input and output data structures (Bowers & Ludascher, 2005). The differences between these input and output data structures can often be complex and state-dependent, requiring developers to create intermediate data structures for temporary storage and “build in” low-level and specialized control-flow support for accessing, combining, and manipulating portions of each data structure explicitly (Bowers et al., 2006; Janert, 2010). The object interactions required by this type of code typically do not use design patterns and as a result are not as easily reused nor improved by using Legwork.

The actual process of composing modules together and integrating them into a cohesive application also requires the addition of module-based application infrastructure, and that each module conform to a common, predetermined structure. For example, using the Microsoft Prism framework requires each module implement the `IModule` interface, and the use of specialized "module composition" code to follow Prism’s four step module loading process (Brumfield et al., 2011). Our results align with previous work indicating that the inclusion of this additional infrastructure adds complexity resulting in code that is more difficult to reuse (Bowers et al., 2006).

5.1.2 Workflow-Based Composition

In contrast to module-based composition, the workflow-based model is centered around facilitating data flow, and draws upon data and computational
resources through web services (Tan, Zhang, & Foster, 2010). Using web services provides a higher level of abstraction, minimizing the need to manage low-level code interactions and reducing code complexity (Brehner, 2011; Tan et al., 2010). Web services are also often designed to be used together in workflows, and commonly include functionality that eliminates the need for the developer to maintain complex data structures for storing intermediate data products (Tan et al., 2010). For example, the web services used in the Entrez Applications Case Study provide integrated server-side storage for intermediate data products, along with functionality for directly piping these data products to other web services (Sayers & Miller, 2010). By effectively “offloading” temporary data storage and low-level processing to the server, the amount and complexity of the code is reduced and more object interactions may be expressed using design patterns (Bowers et al., 2006; Sayers & Miller, 2010). As a result, code from software that uses a workflow-based model of composition is generally more easily reused and improved by using Legwork.

5.1.3 Summary

Though both models of composition are improved by using Legwork, the object interactions and additional infrastructure required by the module-based composition model results in code that is not as easily expressed using design patterns and reused. In order to receive the most improvement from using Legwork and maximize code reuse a workflow-based model of composition should be used when designing and developing software for the life sciences.

5.2 The Impact of Legwork on Code Reuse

Code developed using Legwork is more reusable due to several design and implementation features that specifically address three important reuse considerations: minimizing code dependencies, providing new implementations of
design patterns optimized for code reuse, and providing support for design pattern encapsulation, composition, and code control flow.

According to Brechner (2011) one of the key ingredients for reuse to be successful is effective code isolation. As seen in Figure 5.1 each design pattern implementation in Legwork contains no dependencies on other code. In contrast, frameworks like Prism make heavy use of class inheritance, resulting in a high number of reuse-averse implementation dependencies (Booch et al., 2007).

![Dependency Diagram for Legwork Source Code](image)

(a) Dependency Diagram for Legwork Source Code

![Dependency Diagram for Prism Source Code](image)

(b) Dependency Diagram for Prism Source Code

Figure 5.1. Dependency Diagrams Generated from Legwork and Prism Source Code

As we saw in the subsystem level and system level code evaluations, using the Asynchronous Event Pattern provided by Best Practices to call web services and
other asynchronous procedures results in a code “spaghetti” of callbacks and object interaction code (Skeet, 2010). Prism provides a work-around through the additional use of the Event Aggregator Pattern as a centralized mechanism for passing around data objects. While the Event Aggregator Pattern does assist in synchronously linking components together, it does not directly address the root cause of the problem; the mismatch between synchronous and asynchronous events. In addition, Prism’s implementation of the Event Aggregator Pattern requires knowledge of the implementation details and the creation of additional Prism-derived classes for every data object (see Figure 4.5). As a result, the inclusion of the Event Aggregator Pattern actually compounds the issue, resulting in increases in code complexity and decreasing code reusability.

Legwork encapsulates each design pattern in a single generically-typed implementation. This approach maximizes code reusability by providing a simplified logical abstraction that any object in the code may use without requiring any additional code changes, and without the developer needing to know how the design pattern was implemented. Legwork also supports design pattern composition and provides control over the sequence of code execution. The composition and code control functionality are implemented in Legwork as generic extension methods which effectively attaches these capabilities to every object in the code. Finally, Legwork directly addresses the mismatch between synchronous and asynchronous events by providing a new design pattern implementation that integrates support for these two event-based object interactions. This design pattern is contained within the generically-typed \texttt{Servable} class. Legwork’s use of generics in of each of these features improves reusability without needing to make any changes to the original code.
5.2.1 Summary

Legwork has been designed to improve code reuse in several ways, illustrated in Figure 5.2. First, by removing the need to write and design code around the patterns needing to be used, for example, the Asynchronous Event Pattern for making web service calls. Second, by providing each object with the “machinery” to effectively manage interactions on their own, which removes the dependency on an external object like the Event Aggregator Pattern. Third, through seamless pattern composition, removing the need for temporary transfer storage objects.
Best Practices Code

1: var client = new eUtilsServiceSoapClient();
2: var linkReq = new eLinkRequest();
3: client.run_elinkCompleted += (sender, e) =>
4: {
5:     if (e.Result != null)
6:     {
7:         ELinkResult = new ELinkModel(e.Result);
8:         EventAgg.GetEvent<ELinkServiceCompletedEvent>()
9:             .Publish(ELinkResult);
10:     }
11:     Status = "Ready";
12: }
13: Status = "Busy";
14: client.run_elinkAsync(linkReq);

Legwork Code

1: var client = new eUtilsServiceSoapClient();
2: var linkReq = new eLinkRequest();
4:     StateChange() => Status.Set("Busy"), () => Status.Set("Ready")
5:     Subscribe(result => ELinkResult.Set(new ELinkModel(result)));
6: return ELinkResult

Code Function

- Asynchronously Invoke Web Service  **(Asynchronous Event Pattern)**
- Notify User Interface of Status Changes  **(Event Pattern)**
- Store the Result of the Web Service in a new Model Class
- Pass the New Model Class to Interested Parties  **(Event Aggregator Pattern)**

Figure 5.2. An Example of Design Pattern Encapsulation and Composition Using Legwork
5.3 Directions for Future Work

While presenting an improvement in code reuse, this work also provides several potential areas for future studies. According to Poulin (2006) code designed for reuse can cost up to 300% more to initially develop. This large initial investment represents one of the largest barriers to considering reuse.

As provided in Table 4.1, using Legwork required 58.8% less code than using Best Practices in the Entrez Applications Case Study. For the .NET Bio Case Study, using Legwork required 26.4% less code than using Best Practices. Figure 5.3 contains development time and cost values corresponding to these reductions in LOC. The time calculation used (see Equation 5.1) is based on the average level of programming productivity for a scientific software development project using high level procedural languages, such as C# in this work (Reifer, 2004). The cost calculation used (see Equation 5.2) is based on the average salary for a genomics scientist working in industry in 2012 (Dunning, 2012).

\[
Time_{CaseStudy} = 195 \frac{LOC}{month} \times LOC_{CaseStudy} \quad (5.1)
\]

\[
Cost_{CaseStudy} = 130,000 \frac{dollars}{year} \times Time_{CaseStudy} \quad (5.2)
\]

In the Entrez Applications Case Study, using Legwork results in an initial reduction in development cost of almost $10,000 and 25 days of time. For the .NET Bio Case Study, the savings in development equated to about $5,000 and 12 days of time. These calculated savings in time and cost represent substantial savings; additional studies characterizing the realization of these savings in more extensive comparative studies would be a valuable investment of future effort.
Figure 5.3. Development Time and Cost for the Entrez Applications and .NET Bio Case Studies.
In addition to studies on potential time and cost savings, from the perspective of software development, we focused on the improvement of source code. Future extensions of this work should establish optimal usage conditions and methodologies from a developer perspective. Another area that should be explored is the extent and contexts that the demonstrated improvements in reuse may be generalized and applied: to other areas of science, integrated into existing application frameworks like Prism, and to other programming concepts and development platforms. By extending the work in this dissertation into these areas it becomes possible for software development to efficiently provide the tools needed to support the current rapid pace of advancement at the interface of science and technology.
LIST OF REFERENCES
LIST OF REFERENCES

Ackerman, L. (2010). *Patterns-Based Engineering: Successfully Delivering Solutions via Patterns*. Addison-Wesley Professional.


CHAPTER A. REUSE METRIC MEASUREMENTS

Figure A.1. Difference in Cyclomatic Complexity Between Legwork and Best Practices in the Entrez Applications Case Study
Figure A.2. Difference in Efferent Coupling Between Legwork and Best Practices in the Entrez Applications Case Study
Figure A.3. Difference in Lines of Code Between Legwork and Best Practices in the Entrez Applications Case Study
Figure A.4. Difference in Lack of Cohesion of Methods Between Legwork and Best Practices in the Entrez Applications Case Study
Figure A.5. Difference in Cyclomatic Complexity Between Legwork and Best Practices in the .NET Bio Case Study
Figure A.6. Difference in Efferent Coupling Between Legwork and Best Practices in the .NET Bio Case Study
Figure A.7. Difference in Lines of Code Between Legwork and Best Practices in the .NET Bio Case Study
Figure A.8. Difference in Lack of Cohesion of Methods Between Legwork and Best Practices in the .NET Bio Case Study
VITA
VITA

Nicholas V. Iannotti

**Professional Activities**

<table>
<thead>
<tr>
<th>Year</th>
<th>Position Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012-</td>
<td>Senior Associate, Research Business Technology. Pfizer, Inc.</td>
</tr>
<tr>
<td>2009-2010</td>
<td>Summer Student Intern, Worldwide R&amp;D. Pfizer, Inc.</td>
</tr>
<tr>
<td>2008</td>
<td>Co-Founder, Genomic Guidance, LLC.</td>
</tr>
<tr>
<td>2006-2009</td>
<td>Research Assistant, College of Technology. Purdue University</td>
</tr>
<tr>
<td>2004-2007</td>
<td>Research Associate, College of Pharmacy. Purdue University</td>
</tr>
</tbody>
</table>

**Honors and Awards**

<table>
<thead>
<tr>
<th>Year</th>
<th>Award Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012</td>
<td>Pfizer Corporate Innovation Award Nominee</td>
</tr>
<tr>
<td>2009-2012</td>
<td>Fredrick A. Andrews Doctoral Fellowship</td>
</tr>
<tr>
<td>2009</td>
<td>Purdue Information Technology Summit, Best Research Poster</td>
</tr>
<tr>
<td>2008</td>
<td>Purdue Information Technology Summit, Best Research Poster</td>
</tr>
</tbody>
</table>

**Patents**

<table>
<thead>
<tr>
<th>Year</th>
<th>Inventors</th>
<th>Title</th>
</tr>
</thead>
</table>
Publications and Conference Proceedings


