CLOSING THE DEFECT REDUCTION GAP BETWEEN SOFTWARE INSPECTION AND TEST-DRIVEN DEVELOPMENT: APPLYING MUTATION ANALYSIS TO ITERATIVE, TEST-FIRST PROGRAMMING

by

Jerod W. Wilkerson

Copyright © Jerod W. Wilkerson 2008

A Dissertation Submitted to the Faculty of the

COMMITTEE ON BUSINESS ADMINISTRATION

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY
WITH A MAJOR IN MANAGEMENT

In the Graduate College

THE UNIVERSITY OF ARIZONA

2008
As members of the Dissertation Committee, we certify that we have read the dissertation
prepared by Jerod W. Wilkerson

entitled Closing the Defect Reduction Gap Between Software Inspection and Test-Driven Development: Applying Mutation Analysis to Iterative, Test-First Programming

and recommend that it be accepted as fulfilling the dissertation requirement for the

Degree of Doctor of Philosophy

Jay F. Nunamaker, Jr.  Date: May 28, 2008

Susan A. Brown  Date: May 28, 2008

Bongki Moon  Date: May 28, 2008

Final approval and acceptance of this dissertation is contingent upon the candidate’s submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director:  Jay F. Nunamaker, Jr.  Date: May 28, 2008
STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: ____________________________________
                      Jerod W. Wilkerson
ACKNOWLEDGEMENTS

I want to thank my dissertation advisor Jay Nunamaker for his support, guidance and helpful advice during my time as a Ph.D. student and especially during my dissertation research. I also want to thank the other members of my dissertation committee (Sue Brown, Bongki Moon, and Leon Zhou) for their time, support, and willingness to answer questions, and Matt Jenson and Mark Borgstrom for help with statistical analysis.

Finally, I want to thank my parents Gerald and Chella Wilkerson for their constant support throughout my life, and my father and mother in-law Grant and Dixie Taylor, without whose support, I would not have been able to complete this undertaking.
DEDICATION

To my wife Emily without whose constant support and understanding I could never have even begun this project, and to my children (Joshua, Rebekah, Daniel, Hannah, and Adam) who sacrificed four years of their dad's time and attention without complaint.
# TABLE OF CONTENTS

**LIST OF FIGURES** ..................................................................................................................10  
**LIST OF TABLES** ..................................................................................................................11  
**ABSTRACT** ...............................................................................................................................12  
**CHAPTER 1: INTRODUCTION** .................................................................................................14  
  1.1 Background ..........................................................................................................................14  
    1.1.1 Software Inspection .........................................................................................................15  
    1.1.2 Test-Driven Development .............................................................................................15  
    1.1.3 Mutation Analysis ..........................................................................................................17  
    1.1.4 Change Impact Analysis ..............................................................................................18  
  1.2 Purpose and Research Questions .........................................................................................19  
    1.2.1 Part 1 Research Questions and Hypotheses ..................................................................20  
    1.2.2 Part 2 Research Questions and Hypotheses ..................................................................23  
  1.3 Dissertation Overview .........................................................................................................26  
**CHAPTER 2: LITERATURE REVIEW** .......................................................................................27  
  2.1 Software Inspection .............................................................................................................27  
  2.2 Test-Driven Development ...................................................................................................32  
  2.3 Mutation Analysis ...............................................................................................................35  
  2.4 Change Impact Analysis .....................................................................................................39  
    2.4.1 Program Change Identification ....................................................................................41  
    2.4.2 Dependency Graph Construction ..................................................................................43  
  2.5 Automated Test Generation ................................................................................................46  
**CHAPTER 3: COMPARISON OF SOFTWARE INSPECTION AND TEST-DRIVEN DEVELOPMENT: ESTABLISHING THE DEFECT REDUCTION GAP** .......................................................................................49  
  3.1 Method ..................................................................................................................................49  
    3.1.1 Participants ....................................................................................................................50  
    3.1.2 Experimental Procedures .............................................................................................51  
      3.1.2.1 Software Inspection .................................................................................................51  
      3.1.2.2 Test-Driven Development .......................................................................................54  
    3.1.3 Measurement ..................................................................................................................55  
      3.1.3.1 Defects Remaining ....................................................................................................55  
      3.1.3.2 Cost ..........................................................................................................................58  
    3.1.4 Threats to Internal Validity .............................................................................................59  
      3.1.4.1 Selection Bias ............................................................................................................59  
      3.1.4.2 Mortality Bias ...........................................................................................................59  
      3.1.4.3 Maturation Bias ........................................................................................................60  
      3.1.4.4 Order Bias ...............................................................................................................60  
    3.1.5 Threats to External Validity .............................................................................................61  
  3.2 Results ....................................................................................................................................63  
    3.2.1 Defects Remaining .........................................................................................................65  
    3.2.2 Initial Development Cost ...............................................................................................68  
  3.3 Discussion and Implications .................................................................................................70
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4 Summary</td>
<td>71</td>
</tr>
<tr>
<td>CHAPTER 4: DESIGN SCIENCE RESEARCH APPROACH</td>
<td>73</td>
</tr>
<tr>
<td>4.1 Design Science Research Overview</td>
<td>73</td>
</tr>
<tr>
<td>4.1.1 Design as an Artifact</td>
<td>75</td>
</tr>
<tr>
<td>4.1.2 Problem Relevance</td>
<td>76</td>
</tr>
<tr>
<td>4.1.3 Design Evaluation</td>
<td>77</td>
</tr>
<tr>
<td>4.1.4 Research Contributions</td>
<td>77</td>
</tr>
<tr>
<td>4.1.5 Research Rigor</td>
<td>77</td>
</tr>
<tr>
<td>4.1.6 Design as a Search Process</td>
<td>77</td>
</tr>
<tr>
<td>4.1.7 Communication of Research</td>
<td>78</td>
</tr>
<tr>
<td>4.2 Design Science Development of a Change Impact Analysis Algorithm</td>
<td>78</td>
</tr>
<tr>
<td>4.2.1 Design as an Artifact</td>
<td>78</td>
</tr>
<tr>
<td>4.2.2 Problem Relevance</td>
<td>79</td>
</tr>
<tr>
<td>4.2.3 Design Evaluation</td>
<td>79</td>
</tr>
<tr>
<td>4.2.3.1 Simulation Experiment</td>
<td>80</td>
</tr>
<tr>
<td>4.2.3.2 Measurement</td>
<td>85</td>
</tr>
<tr>
<td>4.2.3.3 Complexity Analysis</td>
<td>88</td>
</tr>
<tr>
<td>4.2.4 Research Contributions</td>
<td>90</td>
</tr>
<tr>
<td>4.2.5 Research Rigor</td>
<td>90</td>
</tr>
<tr>
<td>4.2.6 Design as a Search Process</td>
<td>90</td>
</tr>
<tr>
<td>4.2.7 Communication of Research</td>
<td>91</td>
</tr>
<tr>
<td>4.3 Design Science Application of Mutation Analysis to Iterative, Test-First Programming</td>
<td>91</td>
</tr>
<tr>
<td>CHAPTER 5: CHA-AS: AN ARBITRARY STARTING POINT CHANGE IMPACT ANALYSIS ALGORITHM</td>
<td>93</td>
</tr>
<tr>
<td>5.1 Algorithm Description</td>
<td>93</td>
</tr>
<tr>
<td>5.1.1 Initialize Impacted Method Queue</td>
<td>96</td>
</tr>
<tr>
<td>5.1.2 Process Fields Set by Impacted Method</td>
<td>99</td>
</tr>
<tr>
<td>5.1.3 Method Processing Ranges</td>
<td>101</td>
</tr>
<tr>
<td>5.1.4 CHA Reference Expansion</td>
<td>102</td>
</tr>
<tr>
<td>5.2 Simulation Experiment Results</td>
<td>103</td>
</tr>
<tr>
<td>5.2.1 Descriptive Statistics</td>
<td>103</td>
</tr>
<tr>
<td>5.2.2 Model Assumptions</td>
<td>105</td>
</tr>
<tr>
<td>5.2.3 Hypothesis Testing</td>
<td>107</td>
</tr>
<tr>
<td>5.2.4 Follow-Up Analysis</td>
<td>109</td>
</tr>
<tr>
<td>5.3 Complexity Analysis Results</td>
<td>110</td>
</tr>
<tr>
<td>5.3.1 Time Complexity Analysis Results</td>
<td>111</td>
</tr>
<tr>
<td>5.3.2 Space Complexity Analysis Results</td>
<td>115</td>
</tr>
<tr>
<td>5.4 Discussion and Implications</td>
<td>119</td>
</tr>
<tr>
<td>5.5 Summary</td>
<td>121</td>
</tr>
<tr>
<td>CHAPTER 6: APPLICATION OF CHA-AS TO MUTATION ANALYSIS IN AN ITERATIVE, TEST-FIRST DEVELOPMENT ENVIRONMENT</td>
<td>122</td>
</tr>
<tr>
<td>6.1 Application of CHA-AS to Mutation Analysis</td>
<td>122</td>
</tr>
<tr>
<td>6.2 Enhanced TDD Method</td>
<td>124</td>
</tr>
<tr>
<td>6.3 Additional Research</td>
<td>125</td>
</tr>
<tr>
<td>6.4 Tool Support</td>
<td>126</td>
</tr>
<tr>
<td>6.5 Summary</td>
<td>127</td>
</tr>
<tr>
<td>CHAPTER 7: CONCLUSIONS</td>
<td>129</td>
</tr>
<tr>
<td>7.1 Summary of Research Contributions</td>
<td>129</td>
</tr>
<tr>
<td>7.2 Limitations</td>
<td>131</td>
</tr>
<tr>
<td>7.2.1 Part 1 Limitations</td>
<td>131</td>
</tr>
<tr>
<td>7.2.2 Part 2 Limitations</td>
<td>132</td>
</tr>
<tr>
<td>7.3 Future Research</td>
<td>134</td>
</tr>
<tr>
<td>7.3.1 Comparisons of Defect Reduction Methods</td>
<td>134</td>
</tr>
<tr>
<td>7.3.2 Test-Driven Development</td>
<td>134</td>
</tr>
<tr>
<td>7.3.3 Mutation Analysis</td>
<td>135</td>
</tr>
<tr>
<td>7.3.4 Change Impact Analysis</td>
<td>135</td>
</tr>
<tr>
<td>7.4 Concluding Remarks</td>
<td>135</td>
</tr>
<tr>
<td>APPENDIX A: FIXED-SIZE GROUPING GENETIC ALGORITHM</td>
<td>136</td>
</tr>
<tr>
<td>A.1 Abstract</td>
<td>136</td>
</tr>
<tr>
<td>A.2 Introduction</td>
<td>137</td>
</tr>
<tr>
<td>A.2.1 Fixed-Size Partitioning Problems</td>
<td>137</td>
</tr>
<tr>
<td>A.2.2 Genetic Algorithms</td>
<td>138</td>
</tr>
<tr>
<td>A.3 Literature Review</td>
<td>141</td>
</tr>
<tr>
<td>A.4 Fixed-Size Grouping Genetic Algorithm</td>
<td>145</td>
</tr>
<tr>
<td>A.4.1 Initialization</td>
<td>145</td>
</tr>
<tr>
<td>A.4.2 Evaluation</td>
<td>145</td>
</tr>
<tr>
<td>A.4.3 Selection</td>
<td>146</td>
</tr>
<tr>
<td>A.4.4 Inversion</td>
<td>147</td>
</tr>
<tr>
<td>A.4.5 Crossover</td>
<td>147</td>
</tr>
<tr>
<td>A.4.6 Mutation</td>
<td>149</td>
</tr>
<tr>
<td>A.4.7 Immigration</td>
<td>149</td>
</tr>
<tr>
<td>A.4.8 Greediness Features</td>
<td>150</td>
</tr>
<tr>
<td>A.4.8.1 Top-K</td>
<td>150</td>
</tr>
<tr>
<td>A.4.8.2 Best Two</td>
<td>151</td>
</tr>
<tr>
<td>A.4.8.3 Best Parent Child</td>
<td>151</td>
</tr>
<tr>
<td>A.4.9 Convergence</td>
<td>151</td>
</tr>
<tr>
<td>A.5 Results</td>
<td>151</td>
</tr>
<tr>
<td>A.5.1 Experiment 1</td>
<td>152</td>
</tr>
<tr>
<td>A.5.1.1 Comparison Heuristic</td>
<td>153</td>
</tr>
<tr>
<td>A.5.1.2 Results</td>
<td>153</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS - Continued

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.5.2 Experiment 2</td>
<td>154</td>
</tr>
<tr>
<td>A.5.2.1 Results</td>
<td>155</td>
</tr>
<tr>
<td>A.6 Conclusions, Limitations, and Future Research</td>
<td>155</td>
</tr>
<tr>
<td>APPENDIX B: JAVA AND OBJECT-ORIENTED DESIGN PRETEST</td>
<td>156</td>
</tr>
<tr>
<td>APPENDIX C: PROGRAMMING ASSIGNMENT</td>
<td>166</td>
</tr>
<tr>
<td>APPENDIX D: INSTRUCTIONS TO PARTICIPANTS</td>
<td>173</td>
</tr>
<tr>
<td>APPENDIX E: CODE PROVIDED TO PARTICIPANTS</td>
<td>185</td>
</tr>
<tr>
<td>E.1 Code Listing for SpamFilter Class</td>
<td>186</td>
</tr>
<tr>
<td>E.2 Code Listing for AbstractRuleParser Class</td>
<td>189</td>
</tr>
<tr>
<td>E.3 Code Listing for Rule Interface</td>
<td>195</td>
</tr>
<tr>
<td>E.4 Code Listing for Message Class</td>
<td>200</td>
</tr>
<tr>
<td>E.5 Code Listing for ArgumentParser Class</td>
<td>205</td>
</tr>
<tr>
<td>APPENDIX F: CHA-AS ALGORITHM PSEUDOCODE</td>
<td>208</td>
</tr>
<tr>
<td>F.1 CHA-AS Algorithm Pseudocode</td>
<td>209</td>
</tr>
<tr>
<td>F.2 CHA-AS Algorithm Data Structures</td>
<td>219</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>222</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

FIGURE 2.1: Software Inspection Overview .......................................................... 28
FIGURE 2.2: TDD Overview .................................................................................. 32
FIGURE 2.3: Overview of Impact Analysis Algorithms ......................................... 44
FIGURE 3.1: Treatment Conditions and Defect Counting .................................. 57
FIGURE 3.2: Estimated Marginal Means of Defects Remaining ......................... 66
FIGURE 3.3: Estimated Marginal Means of Total Cost ....................................... 69
FIGURE 4.1: Role of Systems Development in IS Research ............................... 74
FIGURE 4.2: Simulation Program Interface ......................................................... 81
FIGURE 4.3: Component-Level Collaboration Diagram ...................................... 83
FIGURE 5.1: Top-Level CHA-AS Activity Diagram .......................................... 95
FIGURE 5.2: Initialize Impacted Method Queue Activity Diagram ..................... 97
FIGURE 5.3: Process Set Fields Activity Diagram ............................................. 100
FIGURE 5.4: Comparison of Total and Impacted Method Counts .................... 104
FIGURE 5.5: Comparison of Total Mutatable and Impacted Mutatable Method Counts 104
FIGURE 5.6: Comparison of Total and Impacted Mutant Counts ..................... 105
FIGURE 5.7: Comparison of MuJava and CHA-AS Execution Times ............... 105
FIGURE 5.8: Curve Fit for MuJava Execution Time .......................................... 112
FIGURE 5.9: Curve Fit for Combined Change Impact Execution Time ............ 112
FIGURE 5.10: Curve Fit for Chianti Only Execution Time ................................. 113
FIGURE 5.11: Curve Fit for CHA-AS Only Execution Time ............................... 113
FIGURE 5.12: Comparison of MuJava and CHA-AS Time Complexity .......... 115
FIGURE 5.13: Curve Fit for MuJava Peak Memory Usage ............................... 116
FIGURE 5.14: Curve Fit for Combined Change Impact Peak Memory Usage .... 117
FIGURE 5.15: Curve Fit for Chianti Peak Memory Usage ................................ 117
FIGURE 5.16: Curve Fit for CHA-AS Peak Memory Usage ............................. 118
FIGURE 5.17: Comparison of MuJava and CHA-AS Space Complexity ........... 119
FIGURE 6.1: Mutation Analysis Enhanced TDD Method .................................. 125
FIGURE C.1: Package and Class Diagram ......................................................... 168
LIST OF TABLES

TABLE 3.1: Reasons for Participant Exclusion ................................................................. 51
TABLE 3.2: Participants Excluded by Group ................................................................. 51
TABLE 3.3: Normality Assumption Tests ................................................................. 64
TABLE 3.4: Descriptive Statistics of Defects Remaining by Group ..................... 65
TABLE 3.5: ANOVA Summary for Number of Defects Remaining ..................... 67
TABLE 3.6: Mean Initial Development Cost by Group ........................................... 68
TABLE 3.7: Descriptive Statistics of Initial Development Cost by Group ......... 68
TABLE 3.8: ANOVA Summary for Initial Development Cost ................................ 70
TABLE 3.9: Summary of Hypothesis Testing Results ........................................... 72
TABLE 5.1: Descriptive Statistics for Comparison Variables .............................. 103
TABLE 5.2: Comparison of Original and Re-sampled Means and Standard Errors .... 107
TABLE 5.3: Results of Wilcoxon Signed Ranks Test Comparison of Means ....... 108
TABLE 5.4: Paired Differences to Program Size Correlations ............................ 109
TABLE 5.5: Results of Wilcoxon Signed Ranks Test on Reduced Data Set Execution
   Time Pair ........................................................................................................ 110
TABLE 5.6: Descriptive Statistics for Reduced Data Set Execution Time Variables .... 110
TABLE 5.7: Time Complexity Equations and Fit Statistics .................................. 114
TABLE 5.8: Space Complexity Equations and Fit Statistics .................................. 118
TABLE 5.9: Summary of Hypothesis Testing Results ........................................... 121
TABLE 6.1: Mutant Impact of Test Changes ............................................................ 123
TABLE A.1: Experiment 1 Objective Value Comparison ....................................... 154
TABLE A.2: Experiment 1 Means and Standard Deviations by Group .......... 154
TABLE B.1: Answers to Pretest ............................................................................. 156
ABSTRACT

The main objective of this dissertation is to assist in reducing the chaotic state of the software engineering discipline by providing insights into both the effectiveness of software defect reduction methods and ways these methods can be improved. The dissertation is divided into two main parts. The first is a quasi-experiment comparing the software defect rates and initial development costs of two methods of software defect reduction: software inspection and test-driven development (TDD). Participants, consisting of computer science students at the University of Arizona, were divided into four treatment groups and were asked to complete the same programming assignment using either TDD, software inspection, both, or neither. Resulting defect counts and initial development costs were compared across groups. The study found that software inspection is more effective than TDD at reducing defects, but that it also has a higher initial cost of development. The study establishes the existence of a defect-reduction gap between software inspection and TDD and highlights the need to improve TDD because of its other benefits.

The second part of the dissertation explores a method of applying mutation analysis to TDD to reduce the defect reduction gap between the two methods and to make TDD more reliable and predictable. A new change impact analysis algorithm (CHA-AS) based on CHA is presented and evaluated for applications of software change impact analysis where a predetermined set of program entry points is not available or is not known. An estimated average case complexity analysis indicates that the algorithm's time and space complexity is linear in the size of the program under analysis, and a simulation
experiment indicates that the algorithm can capitalize on the iterative nature of TDD to produce a cost savings in mutation analysis applied to TDD projects. The algorithm should also be useful for other change impact analysis situations with undefined program entry points such as code library and framework development.

An enhanced TDD method is proposed that incorporates mutation analysis, and a set of future research directions are proposed for developing tools to support mutation analysis enhanced TDD and to continue to improve the TDD method.
CHAPTER 1:
INTRODUCTION

The software development industry is in a state of chaos (*The Chaos Report*, 1994; *The Chaos Report*, 2006), with defective products, budget and schedule overruns, and cancelled software projects being the norm. A recent government study (Tassey, 2002) estimated that software defects are costing the U.S. economy approximately $59.5 billion per year. Several high-profile software failures, such as the Northeast power blackout in 2003, the failure of the Los Angeles airport's air traffic control system in 2004, and two failed NASA mars missions in 1999 and 2000 have helped to focus attention on this problem.

Although the importance of software defect reduction varies across software development organizations, depending on the cost of defects to the organization, all software development organizations, their customers, and eventually society in general would benefit from a reduction in software defects and a lowering of the cost of software defect reduction. The overarching objective of the research reported in this dissertation is to assist in reducing the chaotic state of the software engineering discipline by providing additional insights into both the effectiveness of software defect reduction methods and ways these methods can be improved.

1.1 Background

Various approaches to software defect reduction have been proposed in the academic literature and applied in software development practice. Two of these approaches are
software inspection, and test-driven development (TDD). Both have advantages and
disadvantages, and both are capable of reducing software defects (George & Williams,
2004; Maximilien & Williams, 2003; Parnas & Lawford, 2003; Shull et al., 2002).
However, prior to the start of this dissertation, the defect reduction benefits and initial
development costs of these two approaches had never been directly compared in an
experimental setting. This comparison is the focus of the first part of the dissertation.
The focus of the second part of the dissertation is to analyze a method of both increasing
the defect reduction effectiveness of TDD and making the approach more predictable and
measurable.

1.1.1 Software Inspection

Software inspection has been the focus of over 400 academic research papers since its
introduction by Michael Fagan (1976). Software inspection is a formal method of
inspecting code (and other software artifacts) to identify defects. This method has been
in use for over 30 years and has been found to be very effective at reducing software
defects. Fagan (1976) reported software defect reduction rates between 66 and 82%.
However, software inspection is expensive and is often not used because of a lack of
support from the authors (programmers, designers, etc.) whose software artifacts are to be
inspected.

1.1.2 Test-Driven Development

TDD is an iterative, test-first (ITF) programming practice in which programmers
write unit tests before program code. New tests are written before features are added or
changed, and the new features or changes are considered complete only when the new
tests and any previously written tests succeed. A fundamental principle of TDD is that no
program code is written until an automated unit test requires the code in order to succeed.
Although results have been mixed, some research has shown that TDD can reduce
software defects by between 18 and 50% (George & Williams, 2004; Maximilien &
Williams, 2003), with the added benefit of eliminating defects at an earlier stage of
development than code inspection.

The software development industry has been adopting TDD in recent years as part of
the Extreme Programming software development methodology (for reasons that include
but are not limited to software defect reduction); however, the existing research does not
sufficiently assess whether TDD is a useful supplement or a viable alternative to the
defect reduction benefits of software inspection. A study conducted as the first part of
this dissertation used a quasi-experimental design in a university laboratory environment
to compare the defect reduction benefits and initial development costs of software code
inspection and TDD. The study found that code inspection is significantly more effective
at reducing software defects than TDD, but that it is also significantly more expensive.
This added expense, combined with a reluctance of many software developers to have
their code subjected to inspection, highlights the desirability of improving unit-testing
based methods such as TDD to obtain the defect reduction benefits of software inspection
with lower cost and increased support of the development staff. The comparison study
also highlighted the need to add structure to the TDD method. Although promising as a
defect reduction method, TDD is a relatively new method that lacks the structure
necessary to adequately assess whether it is being properly and consistently applied. This makes it difficult to assess the value of the method and to compare it to other methods. Addition of this structure is a major focus of this dissertation.

1.1.3 Mutation Analysis

Mutation analysis is a technique that has shown promise in improving the software defect reduction benefits of unit testing. Mutation analysis involves the automated creation of several slightly modified versions of a computer program, and execution of these modified versions against a working automated test suite as a means of evaluating the completeness and correctness of the tests. These modified versions are called "mutants". Mutants that do not result in test failures (called "live" mutants) highlight potential missing or incorrect test cases—and often software defects that were undetected by the existing test suite. One of the outputs of mutation analysis is a mutation adequacy score that can be used as a measure of unit-testing effectiveness. When applied to TDD, this measure can be used as an important stopping criterion for the iterations of TDD and as a means to assess the effectiveness of the application of TDD.

Mutation analysis is inexpensive from a human labor point-of-view, so it has the potential to improve the defect reduction performance of TDD and other unit testing-based techniques while maintaining an initial development cost advantage over software inspection. However, the large number of modified versions of a program required by the approach, shown empirically to be a quadratic function of the number of lines of code under test (Offutt, Lee, Rothermel, Untch, & Zapf, 1996), has resulted in performance
issues that have prevented the widespread application of mutation analysis in industrial software development practice.

After demonstrating the existence of a defect reduction gap between software inspection and test-driven development, this dissertation presents an empirical evaluation of a technique that has potential to reduce the cost of applying mutation analysis to TDD by applying change impact analysis.

1.1.4 Change Impact Analysis

Change impact analysis is a method of identifying the impact of a set of changes to software artifacts such as source code. Change impact analysis involves the identification of source code changes and the "ripple-effects" (Stevens, Meyers, & Constantine, 1974; Yau & Collofello, 1980) of these changes on other unchanged code. For example, a change to a method may impact any method that invokes the changed method, and any method invoked by the changed method. These impacted methods may in turn impact any methods they invoke or are invoked by. By identifying a complete set of changed code and all code potentially impacted by these changes, mutation analysis performance can be improved by reducing the number of mutants that need to be executed in any iteration to only those that mutate potentially impacted code. Mutants on code that is unaffected by the changes in the current iteration will also be unaffected, so their status from the previous iteration (as "live" or "killed") can be reported without the need to re-generate and re-execute them.

This dissertation presents a combination of algorithm analysis results and a simulation experiment to measure the expected performance costs and benefits of
applying change impact analysis to mutation analysis in a TDD environment, to
determine if a significant cost savings can be realized.

1.2 Purpose and Research Questions

This dissertation is divided into two main parts with two specific purposes. The
purpose of the first part is to compare the defect rates and relative costs of software code
inspection and TDD. The purpose of the second part is to use a design science research
approach as described by Hevner, March, Park and Ram (2004) to explore ways to
improve the defect reduction performance of TDD and other iterative, test-first
programming methods by applying mutation analysis. As is shown in later chapters, the
combination of TDD and mutation analysis results in a synergistic effect, with the
iterative and incremental nature of TDD providing a means to improve the performance
of mutation analysis, while the addition of mutation analysis to TDD adds a useful
method of assessing test adequacy for each iteration of TDD (Zhu, Hall, & May, 1997).
This assessment of test adequacy can be used by software developers to determine when
iterations should be deemed complete.

Beck (2002, p. 86) makes a brief reference to using mutation analysis with TDD
using the Jester mutation analysis system. However, he does not address issues related to
how mutation analysis is best applied to TDD, or how to effectively use mutation
analysis with its inherent performance issues as part of the rapid, iterative programming
method that TDD is intended to be. Exploring potential solutions to these issues is the
purpose of the second part of the dissertation.
1.2.1 Part 1 Research Questions and Hypotheses

The purpose of the first part of this dissertation is to answer the following research questions:

1. Which software defect reduction method is the most effective at reducing software defects?
2. Are there interaction effects associated with the combined use of these methods?
3. What are the relative costs of these software defect reduction methods?

The literature cited in chapter 2 indicates that both methods can be effective at reducing software defects. However, TDD is a relatively new method, whereas software inspection has been refined through over thirty years of research. Prior research has clearly defined the key factors involved in successfully implementing software inspection, such as optimal software review rates (Fagan, 1986; Gilb & Graham, 1993; Humphrey, 1989, 1995) and inspector training requirements (Gilb & Graham, 1993), whereas TDD is not as clearly defined due to its lack of maturity.

Currently, the defect reduction results for TDD have been mixed with a maximum reported defect reduction rate of 50% (Maximilien & Williams, 2003). Defect reduction from software inspection has consistently been reported at above 50% since Fagan's introduction of the method in 1976. This leads to a hypothesis that software inspection is currently more effective than TDD at reducing defects. However, the defect reduction effect of TDD will most likely improve over time (probably at a higher rate than the more mature software inspection method) as additional research uncovers improvements to the
TDD method.

**H1:** *Software inspection is more effective than TDD at reducing software defects.*

Although research has not been conducted on interaction effects between the two methods in question, there likely is significant overlap between the types of defects found by inspection and those found by TDD. This would result in a lower incremental benefit of applying one of the methods if the other method is already in use because some of the defects that would have been found by the new method would have already been identified by the other method. This leads to a hypothesis that the effect of either method is reduced by the use of the other method.

**H2:** *The defect reduction effect of either method is reduced by use of the other method.*

Software inspection and TDD have fundamental differences that likely result in each method finding or preventing defects that the other method misses. With TDD, the automated unit tests are written by the same programmer who writes the code. Therefore, any misconceptions held by the programmer about the requirements of the system will result in the programmer writing incorrect tests and incorrect code to pass the tests. These "requirement misconception" defects are less likely in code that undergoes software inspection because it is unlikely that all of the inspectors will have the same misconceptions about the requirements that the programmer has—especially if the requirements document has also been inspected for defects. TDD is often combined with Pair-Programming which has been described as "a continuous process of informal code review" (Williams, 2001). However, one of the purposes of this research is to isolate the
benefits of TDD from those of Pair-Programming, so any benefits of Pair-Programming or its integration with TDD are saved for investigation in future research.

Although susceptible to requirement misconception defects, TDD encourages the writing of a large number of unit tests, some of which may test conditions the inspectors overlook during the inspection process. This effect would likely be more noticeable when using inexperienced inspectors, but could occur with any inspectors. These differences between the methods indicate that each method will find defects that the other method overlooks, leading to a hypothesis that the combined use of the methods is more effective than either method alone.

**H3**: The combined use of software inspection and TDD is more effective than either method alone.

The existing literature does not support a hypothesis as to which method has the lowest initial development cost. However, the nature of the cost differs between the two methods, with the cost from TDD resulting from programmers spending additional time writing tests, and the cost from software inspection resulting from both the time spent by the inspectors and the time spent by programmers correcting identified defects. This leads to a hypothesis that the methods differ in initial development cost, with initial development cost being defined as the cost of development up to—but not including—acceptance testing.

**H4**: Software inspection and TDD differ in initial development cost.

For this research, initial development cost specifically excludes the cost of software maintenance. Both methods should result in reduced software maintenance costs but
hypotheses have not been formulated or tested about which method is most effective at reducing these costs.

1.2.2 Part 2 Research Questions and Hypotheses

The second part of the dissertation seeks to answer the following research questions:

4. Can the use of static change impact analysis improve mutation analysis performance when applied in a TDD environment—without reducing mutation analysis effectiveness?

5. How should the TDD process be changed to incorporate the use of mutation analysis?

TDD is a process that encourages developers to perform many development iterations, with each iteration adding the minimum functionality that can be tested with new automated tests (Astels, 2003; Beck, 2002). This means that the true change impact on each iteration should be small compared to the size of the entire project, especially on large projects or in later iterations of even small and medium sized projects. Prior research has shown that CHA and other static dependency graph construction algorithms are able to create reasonably precise dependency graphs that exclude a large number of methods that can be shown to be unaffected by a set of changes (Bacon, 1997; Bacon & Sweeney, 1996; Dean, Grove, & Chambers, 1995; Grove & Chambers, 2001). Most prior research on dependency graph construction has started from a known program entry point. The need for this research to start from a set of arbitrary changed methods and fields that can not be assumed to be a program entry point is expected to reduce the precision of the resulting dependency graph. However, the highly iterative nature of
TDD is still expected to result in dependency graphs for code changes that exclude a significant number of unaffected methods. This leads to a hypothesis that change impact analysis can significantly reduce the number of methods on which mutation analysis needs to be performed in any given iteration.

**H5:** Change impact analysis can reduce the number of methods on which mutation analysis needs to be performed in a TDD environment.

Code that can safely be shown to be unaffected by edits in a given iteration can be excluded from mutation analysis in that iteration. The mutation result from this unaffected code must also be unaffected, and can be reported based on prior iteration results without re-execution. Therefore, the hypothesis H5, combined with an assumption that mutants are randomly distributed across the source code during any given iteration leads to a hypothesis that change impact analysis can reduce the number of mutants that need to be executed without reducing precision\(^1\) or recall\(^2\).

**H6:** Change impact analysis can reduce the number of mutants that need to be executed—without reducing precision or recall—when applied to mutation analysis in a TDD environment.

The explanations for hypotheses H5 and H6 also lead to hypothesis H7:

**H7:** The mutation reduction benefit of applying change impact analysis to mutation analysis in a TDD environment is positively correlated with program size.

---

\(^1\) Precision is a measure of the degree to which a result is free of "false positives". In mutation analysis, this means the degree to which the result is free of "killed" mutants reported as "live".

\(^2\) Recall is a measure of the degree to which a result is free of "false negatives". In mutation analysis, this means the degree to which the result is free of "live" mutants reported as "killed".
Hypothesis H7 is important, because if true, it will indicate that change impact analysis is most useful in improving mutation analysis on the very cases that most need to be improved—the application of mutation analysis on large projects that result in a cost prohibitive number of mutants.

Prior research has shown that mutation analysis runs in average $O(n^2)$ (quadratic) time, where $n$ is the number of lines of code in the program being analyzed (Offutt, Lee, Rothermel, Untch, & Zapf, 1996). This poor performance result is a primary reason that mutation analysis is not heavily used in software development practice. The CHA algorithm, which is being adapted to this problem, runs in worst-case time of $O(n)$ (linear), so if the algorithm can be adapted to work from a set of arbitrary starting points while still maintaining its worst-case running time of $O(n)$, and if hypothesis H6 is supported, the application of change impact analysis should result in an overall cost savings.

**H8:** The application of change impact analysis to mutation analysis in a TDD environment will result in an overall mutation analysis cost savings.

In addition to testing these eight hypotheses, this dissertation is the beginning of a design science investigation into the answer to research question five. Although specific hypotheses addressing this question are not being formulated or tested here, an initial recommendation is made in chapter 6.
1.3 Dissertation Overview

This dissertation explores the research questions and hypotheses by first positioning this work within the current body of related research in chapter 2. Chapter 3 presents the comparison study of software inspection and TDD to address the research questions and hypotheses for the first part of the dissertation. Chapter 3 also establishes the existence of a gap between the defect reduction effectiveness of software inspection and TDD. Chapter 4 provides general information on the design science research approach and how this approach was used to address the research questions and hypotheses of the second part of the dissertation. Chapter 5 presents and analyzes a new change impact analysis algorithm for use on problems without a predefined program entry point, and chapter 6 presents a method for applying this algorithm to the application of mutation analysis to TDD and other iterative, test-first development methods. Chapter 7 concludes by providing a summary of dissertation findings, contributions, and limitations, and presents additional research questions and issues that need to be explored to further address the application of mutation analysis to TDD.
CHAPTER 2:

LITERATURE REVIEW

The literature review for this dissertation is divided into five main subsections. Each of the first four subsections addresses one of the following major topics of the research: Software Inspection, Test-Driven Development, Mutation Analysis, and Change Impact Analysis. The fifth subsection addresses Automated Test Generation—a topic that is closely related to both TDD and mutation analysis, and will be particularly relevant for investigations of the enhanced TDD approach proposed in chapter 6 and the future research described in chapter 7.

2.1 Software Inspection

Michael Fagan introduced the concept of formal software inspection in 1976 while working at IBM. His original techniques are still in widespread use and are commonly called "Fagan Inspections". Fagan Inspections can be used to inspect the software artifacts produced by all phases of a software development project. However, for the research described in this dissertation, software inspections are limited to software code inspections, commonly called code inspections. Inspection teams normally consist of between three and five participants, including a moderator, the author (designer or programmer), and one to three inspectors. The moderator may also participate in the inspection of the artifact. Fagan Inspections consist of the following phases: Overview (may be omitted for code inspections), Preparation, Inspection, Rework, and Follow-up,
as shown in figure 2.1. The grey arrow between follow-up and inspection indicates that a re-inspection is optional—at the moderator's discretion.

**Figure 2.1: Software Inspection Overview**

In the overview phase, the author provides an overview of the part of the system being addressed by the inspection, followed by a detailed explanation of the artifact(s) to be inspected. Copies of the artifact(s) to be inspected, as well as copies of other materials such as requirements documents and design specifications, are distributed to inspection participants. During the preparation phase, participants independently study the materials received during the overview phase in preparation for the inspection meeting. During the inspection phase, a reader explains in detail the artifact being inspected, covering each piece of logic and every branch of code at least once. During the reading process inspectors identify errors which are recorded either by the moderator or by a designated scribe. The author corrects the errors during the rework phase and all corrections are verified during the follow-up phase. The follow-up phase may be a re-inspection of the artifact or a verification performed only by the moderator.

Fagan (1976) reported defect yield rates (the percentage of the total number of defects found by inspection) between 66 and 82%, where the total number of defects in the
product prior to inspection (t) is
\[ t = i + a + u, \]
where 'i' is the number of defects found by inspection, 'a' is the number of defects found during acceptance testing, and 'u' is the number of defects found during the first six months of use of the product. The defect yield rate (y) is
\[ y = \frac{i}{t} * 100. \]

Several variations of Fagan Inspections for use with code inspections have been reported in the academic literature. Two papers summarizing much of the existing software inspection literature, including inspection variations for use in code inspections, are Laitenberger and DeBaud (2000), and Aurum, Petersson, and Wholin (2002). These summaries indicate that code inspection variations differ mainly in the reading technique used in the Inspection phase of the review. Reading techniques include Ad Hoc Reading (Ackerman, Buchwald, & Lewski, 1989), Checklist-Based Reading (Fagan, 1976, , 1986; Gilb & Graham, 1993; Humphrey, 1995), Reading by Stepwise Refinement (Basili & Selby, 1987; Linger, 1993), Usage-Based Reading (Thelin, Runeson, & Regnell, 2000; Thelin, Runeson, & Wohlin, 2003), and Scenario (or Perspective)-Based Reading (Basili et al., 1996; Denger, Ciolkowski, & Lanubile, 2004; Laitenberger & DeBaud, 1997; J. Miller, Wood, & Roper, 1998). Several comparison studies of reading techniques have been performed (Basili, Caldiera, Lanubile, & Shull, 1996; Basili & Selby, 1987; Porter & Votta, 1994; Porter, Votta, & Basili, 1998; Shull, Lanubile, & Basili, 2000).

Porter and Votta (1994) and Porter, Votta and Basili (1998) performed experiments comparing Ad Hoc Reading, Checklist-Based Reading, and Scenario-Based Reading
using both student and professional inspectors. With Ad Hoc Reading, inspectors were not given specific guidelines to direct their search for defects. With Checklist-Based Reading, inspectors were given a checklist to guide their search for defects. With Scenario-Based Reading, each inspector was given one of the following primary roles or responsibilities to fill during the search for defects: 1) search for data type inconsistencies; 2) search for incorrect functionality; and 3) search for ambiguities or missing functionality. Porter et al. found that Scenario-Based Reading was the most effective, producing improvements over both Ad Hoc Reading and Checklist-Based Reading from 21 to 38% for professional inspectors and from 35 to 51% for student inspectors. They attributed this improvement to efficiency gains resulting from a reduction in overlap between the types of defects for which each inspector was searching. This result was also supported by prior research (Basili, 1997).

Another important advance in the state of software inspections was the application of group support systems (GSS) to the inspection process. Years of prior research have shown that the use of GSS can improve meeting efficiency. Reasons given for these efficiency improvements include reductions in dominance of the meeting by one or a few participants, reductions in distractions associated with traditional meetings, improved group memory, and the ability to support distributed collaborative work (Nunamaker, Briggs, Mittleman, Vogel, & Balthazard, 1997; Nunamaker, Dennis, Valacich, Vogel, & George, 1991). Johnson (1994) notes that the application of GSS to software inspection can overcome obstacles encountered with paper-based inspections, thereby improving the efficiency of the inspection process. Genuchten, Dijk, Scholten, and Vogel (2001) also
found that the benefits of GSS can be realized in software inspection meetings. They conducted a field study at Baan Development where they applied a GSS tool to the preparation and meeting phases of software inspection and found that proper use of GSS can improve defect detection rates by up to 40%. They attributed this success to a reduction in meeting distractions, the ability to see what defects other inspectors had previously logged during preparation—thereby reducing redundancy, and the ability to spend more time in inspection meetings searching for new defects instead of using the meeting time to log defects already identified. Other studies (Biffli, Grünbacher, & Halling, 2006; Lanubile, Mallardo, & Calefato, 2003; Tyran & George, 2002; van Genuchten, Cornelissen, & van Dijk, 1998; Vitharana & Ramamurthy, 2003) have also found improvements in the software inspection process as a result of GSS.

Although comparisons between software inspection and TDD have not been performed, several studies have compared software inspection with more traditional forms of testing. Runeson, et al. (2006) summarized nine studies comparing the effectiveness and efficiency of software inspection and testing in finding code defects. They concluded that "the data doesn't support a scientific conclusion as to which technique is superior, but from a practical perspective it seems that testing is more effective than code inspections." Boehm (1981) analyzed four studies comparing software inspection and unit testing, and found that software inspection is more effective and efficient at identifying up to 80% of code defects.
2.2 Test-Driven Development

TDD is a software development practice that involves the writing of automated unit tests before program code, followed by coding which is deemed complete when the new tests and all previously written tests succeed. TDD consists of the following steps, derived from (Beck, 2002), which are completed iteratively until the software is complete: 1) pick a small piece of functionality to implement; 2) write unit-tests; 3) verify that the unit tests fail (because the functionality they are intended to test has not been implemented); 4) write code to get the tests to pass; and 5) Refactor (Fowler, Beck, Brant, Opdyke, & Roberts, 2000) to eliminate any duplication introduced in step 4. Figure 2.2 provides an illustration of these steps.

Figure 2.2: TDD Overview

Several benefits of TDD have been identified and documented. Maximilien and Williams (2003) list the following benefits of TDD compared to a more traditional "test-last" approach:

2. Production of a more reliable system.

3. Improvement of the quality of the test effort (including an increase in the number of automated unit tests).

4. Reduction of the test effort.

5. Minimization of the schedule.

6. Creation of a thorough regression test bed which can easily determine whether a coding change introduces defects in existing functionality.

Müller and Hagner (2002) compared test-first programming to traditional programming in an experiment involving 19 university students. They used a two-group design involving a control group and a treatment (test-first) group. The participants in both groups performed the same task of completing a program for which they were given a specification, a design, and a set of method declarations. The researchers concluded that test-first programming did not increase program reliability or accelerate the development effort; however, their study was somewhat underpowered, with a statistical power of 0.645.

In a pair of studies by Maximilien and Williams (2003) and George and Williams (2004), the researchers found that TDD did result in higher code quality when compared to traditional programming. Maximilien and Williams performed a case study at IBM on a software development team that developed a Java-based point-of-sale system. The team adopted TDD at the beginning of their project and produced 50% fewer defects than a more experienced IBM team that had previously developed a similar system using traditional development methods. Although the case study lacked the experimental
control necessary to establish a causal relationship, the development team attributed their success to the use of the TDD approach.

George and Williams (2004) conducted a set of controlled experiments with 24 professional pair programmers—programmers who worked in two-person teams and wrote code together using a shared computer. One group of pair programmers used a TDD approach while the other group used a traditional waterfall approach. The researchers found that the TDD group passed 18% more black-box tests and spent 16% more time developing the code than the traditional group. They also reported that the pairs who used a traditional waterfall approach often did not write the required automated test cases at the end of their development cycle.

Erdogmus, Morisio, and Torchiano (2005) conducted an experiment designed to test a theory postulated in previous studies (George & Williams, 2004; Maximilien & Williams, 2003) that the cause of higher quality software associated with TDD is an increased number of automated unit tests written by programmers using TDD. Erdogmus et al. used a two-stage design where they first tested a hypothesis that TDD results in a larger number of unit tests, and then tested hypotheses that a larger number of tests results in both higher quality software and more productive software development. They found that TDD does result in more tests, and that more tests result in higher productivity, but not higher quality.

The mixed results of the TDD literature highlight the need for more research on TDD to develop and test theories about how TDD can be controlled in a way to consistently
produce higher quality software than traditional methods, and whether TDD combined with other methods will produce more consistent code quality improvements.

Software inspection is almost exclusively a software defect reduction method, whereas TDD has several purported benefits—only one of which is software defect reduction. The focus of this dissertation is on the software defect reduction capabilities of the two methods and how these capabilities compare on defect reduction effectiveness and cost. One of the main differences between software inspection and TDD is the point in the software development process in which defects are identified and eliminated. Software code inspection identifies defects at the end of a development cycle, allowing programmers to fix defects previously introduced, whereas, TDD identifies and removes defects during the development process at the point in the process where the defects are introduced. Earlier elimination of defects is a benefit of TDD that can have significant cost savings as explained by Boehm (1981, pp. 39-41).

2.3 Mutation Analysis

The initial ideas for mutation analysis were formulated by Richard Lipton during his graduate studies, and were first published in 1978 (DeMillo, Lipton, & Sayward, 1978). Mutation analysis is an error-seeding approach of test adequacy measurement that automatically inserts errors into a program by generating slightly mutated versions of the code (called “mutants”). Mutants are generated with the use of mutation operators which specify rules for transforming the code to generate mutants. The adequacy measure used in mutation analysis to determine whether a test set is sufficient for testing a piece of
software is the mutation adequacy score (MAS) defined by the formula: 

\[ MAS = \frac{D}{M - E} \]

where \( D \) is the number of mutants detected by existing tests, \( M \) is the total number of mutants generated, and \( E \) is the number of equivalent mutants (mutants resulting in a program that is semantically equivalent to the original).

The mutation analysis method is based on assumptions from two underlying theories. The first is the Competent Programmer Hypothesis, which assumes that the program being analyzed was developed by a competent programmer and as a result, is close to being correct. If this assumption is true, most small deviations from the program (mutants) should be less correct than the original program, and should be detectible by automated tests. The second underlying theory is the Coupling Effect Hypothesis, which states that complex errors are coupled to simple errors, so test data that detects simple errors will also detect complex errors. This theory is used as the reasoning for limiting mutation analysis to first-order mutants (mutants that have been derived from the original program and not from another mutant). In an empirical evaluation of the Coupling Effect Hypothesis, Offutt (1992) demonstrated that a test set that was effective at identifying first-order mutants was also effective at identifying second-order mutants. However, it is not clear that second-order mutants are representative of the complex faults existing in software systems.

Mutation analysis requires the generation and execution of numerous mutated versions of the software under test. Offutt et al. (1996) showed experimentally that the number of mutants (generated from the set of mutation operators used in a majority of previous mutation analysis research) is a quadratic function of the number of lines of
code in the program. As a result, much of the mutation analysis research has focused on improving the performance of the method. Untch (1995) defined the following three categories for classifying mutation analysis performance improvement strategies: ‘do fewer’, ‘do smarter’, and ‘do faster’.

Various strategies have been proposed and evaluated. Offutt et al. (1996) performed an experimental evaluation of N-Selective mutation—a ‘do fewer’ approach where the N operators that produce the most mutants are not used during mutation analysis to reduce computation time. They found in simulation testing involving 10 test programs that 6-Selective mutation (eliminating the 6 highest mutant producing operators from the traditional set of 22 mutation operators) resulted in only a .28% reduction in mutation analysis score (from 99.99% to 99.71%), and that while the number of mutants generated using the reduced set of operators was still quadratic as a function of program size, the reduction represented a substantial performance savings. They further concluded that 5 of the 22 operators are sufficient for mutation analysis, resulting in an average mutation analysis score over the 10 test programs of 99.5%, with the resulting computation cost being reduced to a linear function of the number of data references in the program.

Two primary ‘do faster’ approaches have been proposed for mutation analysis: mutant schemata, and compiler supported mutant patches. The concept of mutant schemata was presented by Untch et al. (1993). This method involves the automated creation of a single “meta-program” to represent all desired mutations of the original code. The use of a single program eliminates the need to compile multiple mutated versions of the code, and therefore, results in a cost savings during mutant generation.
The meta-program is configured during each run of the test suite to represent a different mutated version of the original program. Although this is a promising approach to making mutation analysis practical for industrial use, it has not been implemented in any currently available version of a Java mutation analysis system.

DeMillo et al. (1991) presented the concept of using compiler generated patches to improve the efficiency of mutant generation. This method involves the modification of a programming language compiler to allow the compiler to create “patches” of object or assembly code instructions that can be applied by a patch applicator tool to a compiled version of a program to represent individual mutants. Although the method shows promise for improving mutation analysis performance, it has not been widely used because of the added complexity of modifying a compiler and the requirement that mutation analysis systems based on this approach cannot operate with a traditional compiler.

Several software systems have been developed to automatically create mutants, and to execute an existing test suite against those mutated program versions, reporting both the mutation adequacy score and lists of mutants that are and are not covered by existing tests. The most widely used mutation analysis system is Mothra (K. N. King & Offutt, 1991), a research based system for analysis of Fortran 77 programs. Mothra is the mutation system used in a large majority of prior mutation analysis research. The two primary systems for mutation analysis of Java programs are MuJava³ (Offutt, Ma, &

³ http://ise.gmu.edu/~ofut/muja
Kwon, 2004), and Jester\textsuperscript{4} (Moore, 2001). MuJava supports the set of mutation operators found by Offutt et al. (1996) to be sufficient for mutation analysis, and is a Java version of Mothra. MuClipse\textsuperscript{5} (an Eclipse\textsuperscript{6} plug-in version of MuJava) supports the use of MuJava from the Eclipse integrated Java development environment using JUnit\textsuperscript{7} (Gamma & Beck, 1998) test cases. MuClipse represents an important evolution of the MuJava system because of the widespread usage of both Eclipse and JUnit. Although MuJava is useful for mutation analysis research, performance issues prevent this and other mutation systems from being heavily used in industrial software development practice. Jester supports a very simplistic version of mutation analysis, but does not support the mutation operators found to be effective by the academic mutation analysis research community. However, until the introduction of MuClipse, Jester was the only Java mutation analysis system with direct supported for JUnit test cases.

2.4 Change Impact Analysis

Software change impact analysis has been defined as "identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change" (Bohner & Arnold, 1996). Software change impact analysis can be divided into two broad categories: traceability analysis and dependency analysis. Traceability analysis is concerned with the ability to trace relationships between software artifacts. The most common example of traceability analysis is requirements traceability, where the

\textsuperscript{4} \url{http://jester.sourceforge.net/}
\textsuperscript{5} \url{http://muclipse.sourceforge.net/}
\textsuperscript{6} \url{http://www.eclipse.org/}
\textsuperscript{7} \url{http://www.junit.org/}
relationship between requirements and their associated design and code modules is analyzed. Requirements traceability supports assessments of the impact of requirement changes on design and code artifacts.

Dependency analysis examines detailed relationships between source code level entities such as variables, modules, functions, and methods. Dependency analysis is focused on code-level relationships that are more specific than the broad relationships analyzed by traceability analysis, and is the type of change impact analysis relevant to this dissertation. Two broad categories of dependency analysis include data dependency and control dependency (Bohner & Arnold, 1996). Data dependency analysis involves the analysis of program statements that define or modify data (variables or constants) in a program, and the other parts of the program affected by these statements. Control dependency analysis consists of the statements in a program controlling the flow of execution (such as function and method calls) and the parts of the program affected by these statements. Most applications of dependency analysis consist of the following three steps for both data and control dependencies:

1. Identify Program Changes
2. Create Dependency Graph
3. Use Dependency Graph to Identify Change Impact

The relevant literature relating to the first two steps is described below. The use of the dependency graph to identify the change impact varies depending on the problem domain, and is described for the problem of applying mutation analysis to TDD in chapters 5 and 6.
2.4.1 Program Change Identification

Extensive research has been performed to identify changes from one version of a program to another. Some of the earliest work was the creation of the 'diff' algorithm and corresponding utility for the UNIX operating system (Hunt & Mcllroy, 1976). Diff is a general purpose algorithm that analyzes textual changes between two versions of a file. Although useful as a starting point in identifying program changes, Diff does not take program syntax and semantics into account, and as a result reports changes that are not relevant for software change impact analysis (such as changes in whitespace) and does not have the capability to decompose source code edits into a set of atomic changes with semantic meaning to enable further analysis.

Kung et al (1994) performed early work on identifying the impact of changes in object-oriented systems. Identifying atomic changes from source code edits is particularly challenging for object-oriented programming languages because of features involving virtual method invocation. These features (such as method overriding, method overloading, and changes to class inheritance hierarchies) often result in changes to one source module impacting another source module that is not directly referenced by the impacted module. Ryder and Tip (2001) identified an initial set of atomic changes for object-oriented programming languages that includes these virtual method invocation impacts. Later work (Ren, Shah, Tip, Ryder, & Chesley, 2004; Ren et al., 2004), expanded on this set of atomic changes to identify a complete set of atomic changes covering all features of the Java programming language for use in the Chianti change
impact analysis tool for Java programs. The following is the complete list of Java atomic changes identified for use by Chianti:

1. Add an empty class
2. Delete an empty class
3. Add an empty method
4. Delete an empty method
5. Change body of a method
6. Change virtual method lookup
7. Add a field
8. Delete a field
9. Change definition of an instance field initializer
10. Change definition of a static field initializer
11. Add an empty instance initializer
12. Delete an empty instance initializer
13. Change definition of an instance initializer
14. Add an empty static initializer
15. Delete an empty static initializer
16. Change definition of a static initializer

Any Java program edit can be represented by a set of one or more of these atomic changes.
The purpose of Chianti is to identify a safe set\(^8\) of unit tests potentially impacted by source code changes made from one version of a program to another (Ren, Shah, Tip, Ryder, & Chesley, 2004; Ren et al., 2004). This is the set of unit tests that need to be re-executed on the modified version of the program to ensure complete regression testing of the program. Chianti identifies potentially impacted tests by first analyzing two versions of a Java program and producing a safe set of atomic changes (from the above list of 16 possible atomic changes), along with the directly impacted methods or initializer blocks for each atomic change. This phase of Chianti’s execution is particularly relevant to this dissertation. Chianti then constructs a dependency graph showing the methods and initializer blocks directly accessible from each unit test. Any unit test identified on the call graph as potentially reaching an impacted method or initializer block is reported as an impacted test.

2.4.2 Dependency Graph Construction

The next step in dependency analysis, after identification of a safe set of atomic changes, is the creation of a dependency graph. Several algorithms have been developed for statically constructing safe dependency graphs (Grove & Chambers, 2001). Statically constructed dependency graphs are constructed strictly from the source code, without the need to execute the code during graph construction. These static graph construction algorithms differ primarily on the precision of the resulting graph and the execution time and memory space required to create the graph. Figure 2.3 shows four such algorithms,

---

\(^8\) A "safe" set is a set that is guaranteed to contain all relevant items. The set may include irrelevant items, but it may not omit any relevant items.
listed in order from least precise and least expensive (in terms of execution time and memory space) to most precise and most expensive (Bacon & Sweeney, 1996; Grove & Chambers, 2001; Tip & Palsberg, 2000). The increasing precision and cost is a result of an increasing amount of information used to constrain the edges reported in the resulting call graph.

**FIGURE 2.3: Overview of Impact Analysis Algorithms**

RA is a simple algorithm that constructs call graphs by including edges from each method call site to each method whose name matches the name specified at the call site (Srivastava, 1992; Tip & Palsberg, 2000). This algorithm is inexpensive but also imprecise—generating edges for methods whose names match the name specified at the call site but could never be invoked from that site at runtime.

CHA improves the precision of RA by using class inheritance information to further constrain the set of edges generated from a call site (Dean, Grove, & Chambers, 1995). The set of edges in the resulting call graph are restricted to matching method names in the type referenced at the call site and any subtypes.

RTA starts with a call graph generated by CHA and then improves the precision of the graph by further restricting the edges in the call graph to methods from classes that have been instantiated between the starting point of the program and the call site (Bacon, 1997; Bacon & Sweeney, 1996). Any methods from classes that have not been instantiated in the program before reaching the call site cannot be the target of virtual
method calls, and can safely be eliminated from the call graph at that point in the program. Although slightly slower than CHA, the algorithm still runs in $O(n)$ (linear) time, where $n$ is the size of the program being analyzed (Bacon & Sweeney, 1996; DeFouw, Grove, & Chambers, 1998). The use of class instantiation information in the algorithm means that RTA can only be used in cases where the algorithm can analyze an entire program from a known program entry point (such as the `main(...)` method in Java and C programs).

The 0-CFA algorithm iteratively propagates sets of classes forward as it constructs a call graph, incrementally adding new edges to the graph (Shivers, 1988, 1991). Like RTA, 0-CFA must start from a known program entry point. Although more precise than RTA, the algorithm runs in worst-case time of $O(n^3)$ where $n$ is the size of the program being analyzed (DeFouw, Grove, & Chambers, 1998).

For the application of mutation analysis to TDD, neither RTA nor 0-CFA can be used to analyze the impact of changes made in a TDD iteration because of the need to start these algorithms from a known program entry point. The purpose of using change impact analysis when applying mutation analysis to TDD is to capitalize on the iterative nature of TDD to reduce the number of mutants that need to be executed. However, there is no requirement that iterations in TDD incrementally build code starting from a program entry point, so the basic assumptions of RTA and 0-CFA cannot be satisfied by TDD. Furthermore, TDD can be used to develop class libraries or frameworks for which there is no program entry point until the library or framework is used in some other program.
Some call graph construction algorithms construct dependency graphs based on runtime information (Law & Rothermel, 2003; Orso, Apiwattanapong, & Harrold, 2003; Orso, Apiwattanapong, Law, Rothermel, & Harrold, 2004). These algorithms, known as dynamic impact analysis algorithms, construct dependency graphs based on the actual runtime call information generated from running the program with known inputs. These algorithms are more precise than static algorithms for the specific inputs used to generate the graphs; however, they are not guaranteed to be safe for all possible inputs.

Dynamic impact analysis algorithms are only appropriate for cases where all relevant program inputs are known in advance. This is true of Chianti's generation of a set of potentially impacted unit tests. Of all the possible inputs to the program being analyzed, only those represented by the existing set of test cases are relevant to Chianti's determination of the set of potentially impacted tests. However, this is not true of the application of mutation analysis to TDD. The reason for applying mutation analysis to TDD is to identify missing test cases and functionality that is not tested by the existing set of test cases. Therefore, dynamic impact analysis algorithms are not useful for the main problem being addressed by the second part of the dissertation.

2.5 Automated Test Generation

One of the most difficult and human-labor intensive aspects of mutation analysis is the creation of test cases that adequately exercise the code under test. Creating test cases that fully execute a target program has been shown to be equivalent to the halting problem, and is therefore undecidable (Howden, 1975; Turing, 1936). However, useful
approximations can be made. In one of the earliest works on automated test generation, Howden (1975) describes a methodology for automatically generating test data based on analysis of program execution paths. Clarke et al. (Clarke, 1989; Clarke, Podgurski, Richardson, & Zeil, 1985) expanded on Howden's work by comparing and evaluating data flow path selection criterion used in automated test case generation. Several prototype systems based at least in part on Howden's methodology were developed starting in the mid-1970's (Boyer & Elspas, 1975; Clarke, 1976; Howden, 1977; J. C. King, 1975; E. F. Miller & Melton, 1975; Ramamoorthy, Ho, & Chen, 1976).

In their paper describing a genetic algorithm for generating test cases, Pargas, Harrold, and Peck (1999) classified existing test case generation algorithms into four main categories: 1) random generators (Mills, Dyer, & Linger, 1987; Thévenod-Fosse & Waeselynck, 1993; Voas, Morell, & Miller, 1991); 2) structural or path-oriented generators (Boyer & Elspas, 1975; Clarke, 1976; DeMillo & Offutt, 1991; Howden, 1977; Ramamoorthy, Ho, & Chen, 1976); 3) goal-oriented generators (Ferguson & Korel, 1996; Korel, 1990); and 4) intelligent generators (Chang, Carlisle, & Brown, 1992; Michael, McGraw, Schatz, & Walton, 1997; Tracey, Clark, & Mander, 1998). Random generators select test case inputs from some distribution of possible values. Structural or path-oriented generators select paths using the program's control-flow graph, and generate data to traverse these paths. Goal-oriented generators generate test cases to achieve a specific goal (such as executing a particular statement), without regard to the path taken to reach the goal. Intelligent generators use sophisticated code analysis and
artificial intelligence techniques to generate test cases. Pargas et al. classify their algorithm as goal-oriented.

Demillo and Offutt (1991) described the first test case generation algorithm based on mutation testing. The algorithm constructs constraints necessary to reach a particular mutant by using execution path analysis similar to several previous algorithms. The algorithm also includes constraints that are necessary (but not necessarily sufficient) to kill the specific mutant reachable on that path. Constraints are represented as algebraic expressions. The system of constraints is then solved using a combination of heuristics and random value generation (Offutt, 1988). The system is implemented in a program called Godzilla (Offutt, 1990), and integrated with the Mothra mutation analysis system for Fortran 77 programs. An empirical evaluation of the Godzilla system showed that the system was able to generate test data to kill between 95 and 100% of mutants generated for each of five test programs.

Programs have also been developed for automatically generating test data for Java code, including JAX (Stotts, 2002), Eclat (Pacheco & Ernst, 2005), Korat (Boyapati, Khurshid, & Marinov, 2002), TestEra (Marinov & Khurshid, 2001), and an unnamed system based on the Java PathFinder model checker (Havelund & Pressburger, 2000; Visser, Păsăreanu, & Khurshid, 2004). Two of these systems, JAX and Eclat, generate JUnit test cases.
CHAPTER 3:

COMPARISON OF SOFTWARE INSPECTION AND TEST-DRIVEN DEVELOPMENT: ESTABLISHING THE DEFECT REDUCTION GAP

The purpose of the empirical comparison of software inspection and TDD is to answer the following research questions:

1. Which software defect reduction method is the most effective at reducing software defects?
2. Are there interaction effects associated with the combined use of these methods?
3. What are the relative costs of these software defect reduction methods?

In this study, software inspections were limited to software code inspections, and did not include inspections of requirements or design documents.

The main findings of the study are that software inspection is more effective than TDD at reducing software defects, but that inspection is also more expensive. These findings highlight the need to find less expensive defect reduction methods, and are the motivation for the rest of the dissertation. Details of the method used to complete the comparison study, the results, and a discussion of the implications of these results to industrial software development practice are provided.

3.1 Method

The research questions were evaluated in a quasi-experiment using a two-by-two, between-subjects, factorial design. Participants in each research group were required to
independently complete a programming assignment according to the same specification using either inspection, TDD, both (Inspection+TDD), or neither. The programming assignment involved the creation of part of a spam filter using the Java programming language. Participants were given detailed specifications (see Appendix C) and some pre-written code (see Appendix E) and were instructed to use the Java API to read-in an XML configuration file containing the rules, allowed-list, and blocked-list for a spam filter, and to represent this information with a set of Java objects.

3.1.1 Participants

Participants were undergraduate computer science students recruited from an Object-Oriented Programming and Design course at the University of Arizona. Most students were Juniors or Seniors. All students in the class were invited to take a pretest (see Appendix B) to assess their Java programming and object-oriented design knowledge. The 40 students with the highest pretest scores were selected for inclusion in the study.

Each participant was objectively assigned to one of the four research groups—with 10 participants assigned to each group—by a genetic algorithm (described in Appendix A) that attempted to minimize the difference between the groups in both pretest average score and standard deviation. The algorithm was very successful in producing equalized groups without researcher intervention in the grouping; however, several participants either dropped the course, failed to complete the programming assignment, or had to be excluded from the study for other reasons (see tables 3.1 and 3.2 for details). This resulted in unequal research groups, so pretest score was used as a control variable during data analysis.
### Table 3.1: Reasons for Participant Exclusion

<table>
<thead>
<tr>
<th>Reason for Exclusion</th>
<th>Number Excluded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dropped the class for which the programming assignment was a requirement.</td>
<td>2</td>
</tr>
<tr>
<td>Failed to complete the programming assignment.</td>
<td>4</td>
</tr>
<tr>
<td>Cheated on the assignment (submitted the same solution as another participant).</td>
<td>2</td>
</tr>
<tr>
<td>Submitted code that was not testable (see section 3.1.3.1 for definition of ‘testable’ code).</td>
<td>3</td>
</tr>
<tr>
<td>Excluded as outliers (see section 3.2 for explanation).</td>
<td>4</td>
</tr>
</tbody>
</table>

### Table 3.2: Participants Excluded by Group

<table>
<thead>
<tr>
<th>Assigned Group</th>
<th>Number Excluded</th>
<th>Number Remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>TDD</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Inspection</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Both</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>15</strong></td>
<td><strong>25</strong></td>
</tr>
</tbody>
</table>

3.1.2 Experimental Procedures

All participants were given classroom instruction on TDD and the use of JUnit to create automated unit tests prior to the start of the experiment. Participants were given a detailed specification and instructed to individually write Java code to satisfy the requirements of the specification. They were also instructed to record all time spent on the project (in 15 minute increments) in a spreadsheet that was provided. Participants were given two weeks to complete the project in two separate one-week iterations. Instructions provided to participants in each group are included in Appendix D.

3.1.2.1 Software Inspection

Participants in the Inspection and Inspection+TDD groups had their code inspected by a team of three inspectors and were then given one week to resolve the major defects...
found by inspection. These inspections will be referred to as 'Method Inspections'.

Inspectors were students, but were not participants in the study. Inspections were performed according to Fagan's method (Fagan, 1976, 1986) with three exceptions. First, the authors were not invited to participate in the inspection process. The inspection process took two weeks to complete because of the number of inspections performed, and inviting authors to the inspection meetings would have given authors whose code was inspected early in the process extra time to correct their defects. This would have introduced an unwanted variable into the experiment and would have created an inequity in the course grades assigned for the programming assignment. As a result, the moderator also assumed the role of the authors in the inspection meetings.

Second, the inspectors used a collaborative inspection logging tool for both the inspection preparation and the meetings. Each inspector logged issues in the collaborative tool as the issues were found. This allowed the inspectors to see what had already been logged and to spend their preparation time finding issues that had not previously been found by another inspector. Use of the tool also allowed more time in the inspection meetings to find new issues instead of using the meeting time to report and log issues found during preparation. Another benefit of using the collaborative tool is that it significantly reduced meeting distractions, allowing inspectors to remain focused on finding new issues (Rodgers, Dean, & Nunamaker, 2004).

Third, the inspectors used the scenario based inspection approach described by Porter, Votta and Basili (1998). Each inspector was assigned a primary role of searching for either missing functionality, incorrect functionality, or incorrect Java coding. Inspectors
were instructed not to limit themselves to these roles, but to spend extra effort finding all of the defects that would fall within their assigned primary role.

The inspector assigned to search for incorrect Java coding was instructed to use the Java inspection checklist created by Christopher Fox (1999) to guide the search for defects. The checklist was also made available to the other inspectors, although searching for this type of defect was not their primary role. The checklist items most likely to uncover major defects were annotated with the words 'Major' or 'Possible Major' and inspectors were instructed to focus their efforts on these checklist items in addition to their assigned primary role.

Inspectors were given four hours of training—approximately 1 and 1/2 hours on the inspection process and 2 and 1/2 hours on XML processing with Java (the subject matter under inspection). Inspectors were instructed to spend one hour in preparation for each inspection, and inspection meetings were held to within a few minutes of one hour. The number of inspection meetings was limited to two per day to avoid reduced productivity due to fatigue as noted by Fagan (1986) and confirmed by Gilb and Graham (1993). The maximum inspection rate, which has long been known to be a critical factor in inspection effectiveness (Fagan, 1986; Gilb & Graham, 1993; Humphrey, 1989, 1995), was also controlled. Fagan recommends a maximum inspection rate of 125 non-commentary source statements (NCSS) per hour (Fagan, 1986), whereas Humphrey recommends a maximum rate of 300 lines of code (LOC) per hour (Humphrey, 1995). The mean inspection rate in this study was 180 NCSS/hour with a maximum rate of 395 NCSS/hour. Although the maximum rate was slightly above Humphrey's
recommendation, this slightly higher rate seems justified considering that the inspectors were inspecting multiple copies of code written to the same specification, and as a result, became very familiar with the subject matter of the inspections.

Inspectors categorized the issues they found as being either 'major' or 'minor' and were instructed to focus their efforts on major issues as recommended by Gilb and Graham (1993). After all inspections were completed, each author was given an issue report showing all of the issues logged by the inspectors. Authors were then given one week to resolve all major defects and to return the issue report with each defect categorized by the author into one of the following categories: Resolved, Ignored, Not a Defect, or Other. Authors were required to write an explanation for any issue categorized as either 'Not a Defect' or 'Other'.

3.1.2.2 Test-Driven Development

Participants in the TDD and Inspection+TDD groups were instructed to develop automated JUnit tests and program code iteratively while completing the programming assignment. They were instructed to write JUnit tests first whenever creating new functionality or modifying existing functionality and to use the passing of the tests as an indication that the functionality was complete and correct. Participants in the Inspection+TDD group were also instructed to use TDD during correction of the defects identified during inspection.

The Eclipse Plug-in of the Clover\(^9\) test coverage tool by Cenqua was used to provide an objective measure of TDD effectiveness. Code coverage results showed an average of

\(^9\) http://www.atlassian.com/software/clover/
83.44% coverage (including both statement and branch coverage) with a standard deviation of 9.64. Only three participants achieved less than 84% coverage.

3.1.3 Measurement

Most prior research on defect reduction methods has reported 'yield' as a measure of method effectiveness, where 'yield' is the ratio of the number defects found by the method to the total number of defects in the software artifact prior to inspection (Fagan, 1976; Humphrey, 1995). However, 'yield' cannot be reliably calculated for TDD because the TDD method eliminates defects at the point of introduction into the code, making it impossible to reliably count the number of defects eliminated by the method. Therefore, the number of defects remaining after application of the method was used as a substitute for 'yield'. The cost of initial development of the software using the assigned method was used as a second dependent variable.

3.1.3.1 Defects Remaining

The inspection literature contains a well established practice of categorizing defects as either 'major' or 'minor' (Fagan, 1986; Gilb & Graham, 1993), where a major defect is defined as any defect that would eventually "cause a malfunction or unexpected result if left uncorrected" (Fagan, 1986). The total number of defects remaining was defined as the summation of the number of major defects found by software inspection after the application of the defect reduction method and the number of failed automated acceptance tests representing unique defects not found by inspection (out of 58 JUnit tests covering all requirements). This is consistent with the measure used by Fagan (1976) with the exception of the exclusion of the number of defects identified during
actual use of the software during the first six months of use.

As illustrated in figure 3.1, all code—regardless of whether inspection was part of the development method—was subjected to a measurement inspection after completion of the assignment for the purpose of counting defects. For the Inspection and Inspection+TDD groups, the measurement inspection was a separate inspection performed after the original inspection and subsequent correction of identified defects. For the TDD and control groups, the measurement inspection was the only inspection performed. A separate inspection team performed the measurement inspections to prevent bias in favor of the Inspection and Inspection+TDD groups. The same method was used for the measurement inspections as that used for the inspection group inspections except that only two inspectors in addition to the moderator were used due to resource constraints.

The automated JUnit tests were executed after completion of the measurement inspections and any test failures representing defects not already found by inspection were added to the defect counts. The automated tests were written before the start of the experiment with minor adjustments and additions made before the final test run for counting purposes.
One of the automated tests was used as a baseline, with all other tests being variations of this baseline test. The baseline test executed the code against a sample configuration file that was provided to the participants at the start of the experiment. If the baseline test failed, the possibility that some unexpected condition (other than what the tests were intended to check) was causing failures within the test suite could not be ruled out, so code was only considered to be testable if it passed the baseline test and none of the test methods resulted in unexpected exceptions. For some projects, minor changes were made in order to make the code testable, and in such cases, each required change was logged as a defect. Three participants submitted code that would have required extensive changes (debugging) to make it testable according to the above definition. Because it was impossible to ensure that these changes would not alter the author's original intent, these participants were excluded from the study.

Two adjustments to the resulting defect counts were necessary to arrive at the final number of defects remaining in the code. First, the inspection moderator performed an
audit of defects identified by inspection and eliminated false positives. This would have resulted in an understatement of the effect of software inspection if the moderator inadvertently eliminated any real defects, making it less likely to find the reported result.

Second, in several cases, defects that were originally identified by the method inspections and reported to the authors for correction were either ignored or were attempted but not corrected. In a "real world" inspection setting, these ignored defects would have been caught during the iterative 'Follow-Up' inspection phase (see figure 2.1) and the code would not have exited the inspection process until all previously identified major defects were corrected. However, due to time and resource constraints, iterative cycles of Follow-Up, Inspection, and Rework to ensure that all identified major defects were eventually corrected could not be performed, so the Follow-Up phase was modified to identify but not require repeated checking and eventual correction of ignored defects. These ignored defects were eliminated from the final defect counts as if they had been corrected. This adjustment resulted in more accurate defect counting with an understatement of the cost for the inspection method.

3.1.3.2 Cost

The total cost of each method is reported in total man-hours instead of dollars because of the arbitrary nature of assigning an hourly rate to experiment participants and software inspectors. An assumption was made that the hourly rates of all software inspectors and all programmers was the same to facilitate the comparison of defect detection costs across methods; however, man hours are also reported separately for inspectors and programmers in the software inspection case to allow for the application of these findings.
where programmer rates and software inspector rates differ.

The total cost for the software inspection method is the sum of the original development hours spent by the author of the code, the software inspector and moderator hours (preparation time plus meeting time), and the hours spent by the author correcting defects identified by inspection. The total cost for the TDD method is the sum of the development hours used to write both the automated tests and the code.

3.1.4 Threats to Internal Validity

The following four threats to internal validity were considered: 1) selection bias; 2) mortality bias; 3) maturation bias; and 4) order bias.

3.1.4.1 Selection Bias

Selection bias refers to the possibility that the participants in the study were divided unequally into groups, and as a result, the findings are at least partly due to these differences and not to the effects of the treatment conditions. Although many differences in the participants could potentially contribute to a selection bias, Java programming ability seems to be the most likely cause of selection bias in this study. This possibility was accounted for by using a quasi-experimental design with participants assigned to groups by pretest score using the genetic algorithm described in Appendix A.

3.1.4.2 Mortality Bias

Mortality bias refers to the possibility that the groups became unequal after the start of the study as a result of participants either dropping out or being eliminated. The study did suffer from a high mortality rate—starting with 40 participants and ending with 25—partially due to outlier elimination (see tables 3.1 and 3.2). However, pretest scores were
used to measure and control for this effect. A T-Test was also used to compare the means of those who remained in the study (24.52) and those who did not (21.60) and found the difference to be not significant with an alpha of 0.05.

3.1.4.3 Maturation Bias

Maturation bias is the result of participants learning at unequal rates within groups during the experiment. Due to the nature of the experiment, these results may include effects of a maturation bias. As a normal part of the inspection process, participants in both the Inspection and Inspection+TDD groups were given an opportunity to correct defects identified in their code approximately two weeks after submitting the original code, but participants in the control and TDD groups were not given this opportunity. All participants were enrolled in an Object-Oriented Programming and Design course during the experiment and may have gained knowledge during any two-week period of the course that would have made them better programmers and less likely to produce defects. Only the participants whose code was inspected had an opportunity to use any knowledge gained to improve their code, and since this potential maturation effect was not measured, it is not possible to eliminate or quantify the possible effects of a maturation bias.

3.1.4.4 Order Bias

Order bias is an effect resulting from the order in which treatments are applied to participants. This study is vulnerable to an order bias resulting from the order in which inspections were performed and whether inspections were the first or second inspection on the day of inspection. The potential for an order bias was handled in two ways. First, the measurement inspections were performed in random order within blocks of four, and
the method inspections (which involved only two groups) were performed on code from one randomly selected participant from each group each day, alternating each day on which group's inspection was performed first. Second, inspection order, and whether the inspection was performed first or second on the day of inspection, were used as control variables during data analysis.

3.1.5 Threats to External Validity

Four threats to external validity have been identified that limit ability to generalize these results to the software development community:

1. The participants in the study were undergraduate students rather than professional programmers, and therefore, did not have the same level of programming knowledge or experience as the average professional programmer. A decision to choose participants from a class consisting mostly of Juniors and Seniors, and to include only the students with the highest pretest scores in the study was intended to minimize this effect. However the participants still were not representative of the general population of programmers, and most likely represent novice programmers. This would have affected all of the research groups, but since the TDD method was most dependent on the ability of the programmers, it most likely biased the study in favor of software inspection.

2. Although they were not participants in the study, the inspectors were college students and did not have professional software inspection experience. Prior research has shown a positive correlation between inspector experience and the
number of defects found (Fagan, 1986; Gilb & Graham, 1993; Strauss & Ebenau, 1993). The main finding of this study—that inspection is more effective than TDD—is robust to this potential bias which would have had the effect of reducing the likelihood of finding software inspection to be more effective.

3. The nature of the experiment required changes to the software inspection process from what would normally be done in industry. Although the process was based on Fagan's approach, with enhancements based on Porter's findings of the effectiveness of using a scenario based reading approach, and Genuchten et al. and Tyran and George's (among others) findings of increased inspection effectiveness from the use of collaborative tools, two other changes were required. First, authors were not invited to participate in the inspections. Second, an iterative cycle of Rework, Follow-Up, and Re-Inspection to ensure that all identified defects were corrected was not used. Not inviting authors to participate would have resulted in understating the effectiveness of software inspection. However, the effect of not having an iterative cycle of Rework, Follow-Up and Re-Inspection was measured and adjust for, so this would not have affected the results.

4. The inspectors performed multiple inspections in a short period of time, of code that performs the same function and is written to the same specification. This was a necessary part of the experiment, but would rarely if ever occur in practice. This could have resulted in the inspectors finding more defects in later
inspections as they got more familiar with the specifications and with Java-based XML processing code. However, if this affected the results, an order bias should have been detected during data analysis. Use of inspection order as a control variable did not indicate an order bias in the results.

Field studies designed to test the same hypotheses are planned for future research to address these threats to external validity.

3.2 Results

This study included two dependent variables: the total number of major defects (Total_Majors) and the total number of hours spent applying the method (Total_Method_Hours). Bartlett's test of sphericity was used to determine whether the dependent variables were sufficiently correlated to justify use of a single MANOVA instead of multiple ANOVAs for hypothesis testing. According to Meyers, Gamst, and Guarino (2006), a p value less than 0.001 indicates correlation sufficiently strong to justify use of a single MANOVA. Bartlett's test yielded a p value of 0.842, indicating that a MANOVA was not justified, so a separate ANOVA was performed for each dependent variable.

Initial tests of normality indicated that Total_Method_Hours was not normally distributed, and Levene's test indicated Total_Method_Hours did not meet the homogeneity of variance assumption. These assumption violations were corrected by eliminating four outliers (all of the participants with more than 22 initial development hours). Of the four participants eliminated, one was from the TDD group and three were
from the Inspection+TDD group. After elimination of outliers, the following formulas recommended by Hair, Black, Babin, Anderson, and Tatham (2005) were used to obtain Z values for testing the normality assumptions across groups for both dependent variables:

\[
Z_{\text{skewness}} = \frac{\text{skewness}}{\sqrt{\frac{6}{N}}}, \quad Z_{\text{kurtosis}} = \frac{\text{kurtosis}}{\sqrt{\frac{24}{N}}}
\]

As shown in Table 3.3, the normality assumption was satisfied for both dependent variables both within groups and combined across groups.

**Table 3.3: Normality Assumption Tests**

<table>
<thead>
<tr>
<th>Group</th>
<th>Statistic</th>
<th>N</th>
<th>Z-Value</th>
<th>Significant (.05)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Skewness - Total_Majors</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>-0.889</td>
<td>7</td>
<td>-0.96023</td>
<td>NO</td>
</tr>
<tr>
<td>TDD</td>
<td>0.422</td>
<td>8</td>
<td>0.487284</td>
<td>NO</td>
</tr>
<tr>
<td>Inspection</td>
<td>0.623</td>
<td>6</td>
<td>0.623</td>
<td>NO</td>
</tr>
<tr>
<td>Inspection+TDD</td>
<td>-0.753</td>
<td>4</td>
<td>-0.61482</td>
<td>NO</td>
</tr>
<tr>
<td>Combined</td>
<td>-0.013</td>
<td>25</td>
<td>-0.02654</td>
<td>NO</td>
</tr>
<tr>
<td><strong>Kurtosis - Total_Majors</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>-1.083</td>
<td>7</td>
<td>-0.58489</td>
<td>NO</td>
</tr>
<tr>
<td>TDD</td>
<td>0.701</td>
<td>8</td>
<td>0.404723</td>
<td>NO</td>
</tr>
<tr>
<td>Inspection</td>
<td>1.132</td>
<td>6</td>
<td>0.566</td>
<td>NO</td>
</tr>
<tr>
<td>Inspection+TDD</td>
<td>0.343</td>
<td>4</td>
<td>0.140029</td>
<td>NO</td>
</tr>
<tr>
<td>Combined</td>
<td>-0.79</td>
<td>25</td>
<td>-0.80629</td>
<td>NO</td>
</tr>
<tr>
<td><strong>Skewness - Total_Method_Hours</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>-1.052</td>
<td>7</td>
<td>-1.13629</td>
<td>NO</td>
</tr>
<tr>
<td>TDD</td>
<td>1.193</td>
<td>8</td>
<td>1.377558</td>
<td>NO</td>
</tr>
<tr>
<td>Inspection</td>
<td>0.631</td>
<td>6</td>
<td>0.631</td>
<td>NO</td>
</tr>
<tr>
<td>Inspection+TDD</td>
<td>-0.579</td>
<td>4</td>
<td>-0.47275</td>
<td>NO</td>
</tr>
<tr>
<td>Combined</td>
<td>0.684</td>
<td>25</td>
<td>1.396209</td>
<td>NO</td>
</tr>
</tbody>
</table>
Elimination of the outliers corrected the assumption violations and did not change any of the significance findings for the Total_Majors dependent variable.

3.2.1 Defects Remaining

The Inspection+TDD and Inspection groups had the lowest mean number of defects remaining as shown in table 3.4, whereas the TDD group had the highest.

<table>
<thead>
<tr>
<th>Group</th>
<th>Mean</th>
<th>Std. Dev</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inspection+TDD</td>
<td>8.25</td>
<td>3.30</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>Inspection</td>
<td>12.00</td>
<td>7.40</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>Control</td>
<td>14.43</td>
<td>8.52</td>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>TDD</td>
<td>15.38</td>
<td>5.42</td>
<td>7</td>
<td>25</td>
</tr>
</tbody>
</table>

The estimated marginal means of the number of defects remaining (which are adjusted for the average values of the pretest score and inspection order control variables) show the TDD group with a slightly lower number of defects remaining than the control group. Figure 3.2 presents a summary of the estimated marginal means. The solid line represents the two groups that did not use TDD, whereas the dashed line represents the groups that did use TDD. The points with an x-axis value of 0 represent the two groups that did not use inspection, whereas the points with an x-axis value of 1 represent the two groups that did use inspection. Following each line from left to right shows the defect
reduction effect (in mean number of defects) of starting either with or without TDD and adding inspection to the method used. The ANOVA results described below show that the difference between the control and TDD groups is not statistically significant.

**Figure 3.2: Estimated Marginal Means of Defects Remaining**

ANOVA was used to test the hypotheses, and the Java and Object-Oriented Analysis and Design pretest score was used as a control variable to control for the effects of programmer ability. Measurement inspection order and whether the inspection was performed first or second on the day of inspection were used as control variables to control for the effects of inspection order. Eta squared values of 0.239 for the effect of software inspection and 0.427 for the pretest score indicate that software inspection and pretest score account for 23.9 and 42.7% respectively of the total variance in the number of defects remaining. Both the pretest score and whether Inspection was used as a defect reduction method were significant at the 0.05 level of alpha. Both the pretest score and the use of software inspection were negatively correlated with the number of defects.
The use of TDD did not result in a statistically significant difference in the number of defects. However, a low sample size resulted in low statistical power which may account for this lack of significance. The observed power for the effect of TDD was only 0.289. Similarly, the existence of an interaction effect between the use of inspection and TDD is not supported, but this again may be due to a low observed power of 0.152. The results of the ANOVA analysis are summarized in table 3.5, with variables listed in order from most to least significant.

**TABLE 3.5: ANOVA Summary for Number of Defects Remaining**

<table>
<thead>
<tr>
<th>Source</th>
<th>Type III Sum of Squares</th>
<th>Degrees of Freedom</th>
<th>Mean Square</th>
<th>F</th>
<th>Significance (p-value)</th>
<th>Observed Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pretest Score</td>
<td>385.879</td>
<td>1</td>
<td>385.879</td>
<td>13.432</td>
<td>0.002</td>
<td>0.934</td>
</tr>
<tr>
<td>Inspection</td>
<td>162.186</td>
<td>1</td>
<td>162.186</td>
<td>5.645</td>
<td>0.029</td>
<td>0.613</td>
</tr>
<tr>
<td>TDD</td>
<td>62.924</td>
<td>1</td>
<td>62.924</td>
<td>2.190</td>
<td>0.156</td>
<td>0.289</td>
</tr>
<tr>
<td>TDD * Inspection</td>
<td>27.243</td>
<td>1</td>
<td>27.243</td>
<td>0.948</td>
<td>0.343</td>
<td>0.152</td>
</tr>
<tr>
<td>Measure Inspection Order</td>
<td>20.461</td>
<td>1</td>
<td>20.461</td>
<td>0.712</td>
<td>0.410</td>
<td>0.126</td>
</tr>
<tr>
<td>Measure Inspection Order</td>
<td>18.263</td>
<td>1</td>
<td>18.263</td>
<td>0.636</td>
<td>0.436</td>
<td>0.118</td>
</tr>
</tbody>
</table>

Comparisons of group means, using one-tailed T-Tests with a Bonferroni adjustment for multiple comparisons, indicated that the difference between the Inspection+TDD group and the control group was significant at the 0.10 level of alpha. None of the other mean differences were significant.

These results show that hypothesis H1 is supported, indicating that software inspection is more effective than TDD at reducing software defects. Hypothesis H2 is not supported, indicating that the effect of either method is not significantly reduced by using the other method. However, as stated above, low statistical power for the
interaction effect prevent the ability to confidently reject hypothesis H2. Hypothesis H3 is supported at the 0.10 level of alpha, indicating that the combined use of the methods is more effective at reducing defects than either method alone.

3.2.2 Initial Development Cost

An analysis of the initial development costs associated with TDD and Inspection was performed, but an exploration of the cost-benefits of reducing software defects was not. Refer to Boehm (1981) or Gilb and Graham (1993) for in-depth treatment of cost savings associated with defect reduction. Cost was measured in man-hours and TDD was found to have the lowest mean cost and Inspection+TDD to have the highest, as shown in tables 3.6 and 3.7.

TABLE 3.6: Mean Initial Development Cost by Group

<table>
<thead>
<tr>
<th>Group</th>
<th>Programmer Hours</th>
<th>Inspector Hours</th>
<th>Total Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDD</td>
<td>9.36</td>
<td>N/A</td>
<td>9.36</td>
</tr>
<tr>
<td>Control</td>
<td>12.79</td>
<td>N/A</td>
<td>12.79</td>
</tr>
<tr>
<td>Inspection</td>
<td>13.89</td>
<td>7.00</td>
<td>20.89</td>
</tr>
<tr>
<td>Inspection+TDD</td>
<td>17.38</td>
<td>7.00</td>
<td>24.38</td>
</tr>
</tbody>
</table>

TABLE 3.7: Descriptive Statistics of Initial Development Cost by Group

<table>
<thead>
<tr>
<th>Group</th>
<th>Mean</th>
<th>Std. Dev</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDD</td>
<td>9.36</td>
<td>3.53</td>
<td>6.00</td>
<td>16.25</td>
</tr>
<tr>
<td>Control</td>
<td>12.79</td>
<td>4.87</td>
<td>4.00</td>
<td>18.00</td>
</tr>
<tr>
<td>Inspection</td>
<td>20.89</td>
<td>5.56</td>
<td>13.50</td>
<td>30.00</td>
</tr>
<tr>
<td>Inspection+TDD</td>
<td>24.38</td>
<td>8.69</td>
<td>13.00</td>
<td>34.00</td>
</tr>
</tbody>
</table>

Although the estimated marginal means of figure 3.3 show an indication of both an initial development cost savings for using TDD and an interaction effect indicated by the fact that the lines cross, neither of these effects is statistically significant. However, in
both cases, this lack of significance may be the result of low observed statistical power of 0.057 for the potential TDD cost savings and 0.319 for the potential interaction effect.

Figure 3.3: Estimated Marginal Means of Total Cost

ANOVA was used to test hypothesis H4 (that initial development cost differs between the two methods). As with the test for the number of defects remaining, pretest was used as a control variable. However, inspection order was not used as a control variable because the amount of time spent on inspections was held constant, leaving no opportunity for inspection order to affect cost. An eta squared of 0.552 was obtained for the effect of software inspection, indicating that whether software inspection was used accounted for 55.2% of the total variance in the initial development cost. The pretest score was not significant and had an eta squared of less than 0.01. As stated above, significance was not found for an interaction effect between TDD and Inspection. Table 3.8 presents a summary of these results, with variables arranged from most to least significant.
Table 3.8: ANOVA Summary for Initial Development Cost

<table>
<thead>
<tr>
<th>Source</th>
<th>Type III Sum of Squares</th>
<th>Degrees of Freedom</th>
<th>Mean Square</th>
<th>F</th>
<th>Significance (p-value)</th>
<th>Observed Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inspection</td>
<td>695.410</td>
<td>1</td>
<td>695.410</td>
<td>24.618</td>
<td>0.000</td>
<td>0.997</td>
</tr>
<tr>
<td>TDD * Inspection</td>
<td>68.967</td>
<td>1</td>
<td>68.967</td>
<td>2.441</td>
<td>0.134</td>
<td>0.319</td>
</tr>
<tr>
<td>Pretest Score</td>
<td>46.244</td>
<td>1</td>
<td>46.244</td>
<td>1.637</td>
<td>0.215</td>
<td>0.230</td>
</tr>
<tr>
<td>TDD</td>
<td>2.019</td>
<td>1</td>
<td>2.019</td>
<td>0.071</td>
<td>0.792</td>
<td>0.057</td>
</tr>
</tbody>
</table>

Comparisons of group means, using two-tailed T-Tests with a Bonferroni adjustment for multiple comparisons, indicated that the difference between the Inspection+TDD group and both the TDD and control groups was significant at the 0.05 level of alpha. The difference between the Inspection and TDD group was also significant at 0.05, while the difference between the Inspection and control group was not significant (yielding a p value of 0.111).

These results show that hypothesis H4 is supported, indicating an initial cost difference between the use of software inspection and TDD, with the initial cost of software inspection being higher.

3.3 Discussion and Implications

The main implication of this study is to show that software inspection is more effective than TDD at reducing software defects, but that it also has a higher initial development cost. It should be noted, however, that studies have shown that this initial cost is more than offset by cost savings that are realized in later stages of the software development process (Boehm, 1981; Fagan, 1976, 1986; Gilb & Graham, 1993; Humphrey, 1989).
Another implication of this research is the finding that TDD did not significantly reduce the number of defects. Although one possible explanation for this is the low statistical power observed in the study, it may also indicate that additional training and or experience is required to gain defect reduction benefits from TDD. If this is true, it may indicate that the minimum startup cost of TDD as a defect reduction method (defined as the minimum investment required to begin realizing defect reductions) is higher than the minimum startup cost of software inspection, since the inspectors in this study received no more training than the study participants but were still effective at reducing defects.

Finally, the finding that TDD did not increase the initial development cost, implies that software developers can begin using TDD to gain other known benefits of TDD—such as an increased ability to respond to requirement changes without introducing defects to existing functionality—without negatively impacting budgets or schedules. The finding that the combined use of the methods is more effective at reducing defects than either method alone provides support for using a combination of these methods.

3.4 Summary

This study compared the software defect rates and initial development costs associated with two methods of software defect reduction: software inspection, and test-driven development. Prior research has indicated that both methods can be effective at reducing defects, but the methods had not previously been compared.

The study found that software inspection is more effective than TDD at reducing defects, but that software inspection also has a higher initial development cost. Another
finding is that the combined use of these methods is more effective than either method alone. Finally, some evidence was found to indicate that TDD may result in an initial development cost savings, although somewhat conflicting results require additional research to verify this. Previous research has not shown an initial cost savings from TDD. This study did not show a statistically significant reduction in defects associated with the use of TDD and did not show an interaction effect associated with the combined use of these methods. Table 3.9 presents a summary of hypothesis testing results.

These findings have important implications for software development practitioners, and especially managers, but it should be noted that this was a quasi-experiment conducted with student programmers and student inspectors. Caution must be exercised with any attempt to generalize these findings outside of a university laboratory environment.

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>Software inspection is more effective than TDD at reducing defects.</td>
<td>Accepted *</td>
</tr>
<tr>
<td>H2</td>
<td>The defect reduction effect of either method is reduced by use of the other method.</td>
<td>Not Accepted</td>
</tr>
<tr>
<td>H3</td>
<td>The combined use of software inspection and TDD is more effective than either method alone.</td>
<td>Accepted b</td>
</tr>
<tr>
<td>H4</td>
<td>Software inspection and TDD differ in initial development cost.</td>
<td>Accepted a</td>
</tr>
</tbody>
</table>

* at the 0.05 level of alpha.

b at the 0.10 level of alpha.
CHAPTER 4:

DESIGN SCIENCE RESEARCH APPROACH

The research approach for the second part of the dissertation (consisting of questions 4 and 5, and hypotheses H5 – H8 from section 1.2) is based on the principles of design science research summarized by Hevner et al. (2004). This approach was used to develop and evaluate CHA-AS—a CHA-based dependency analysis algorithm that operates from a set of arbitrary starting points in code—and to assess the applicability of CHA-AS to mutation analysis in a TDD environment. The purpose of this chapter is to define and describe the design science research methodology, its role in IS research, and the approach taken to apply this methodology to the research questions and hypotheses for the second part of the dissertation. Section 4.1 describes the design science research approach in general. Section 4.2 describes how this approach was used to construct and evaluate the new change impact analysis algorithm, and section 4.3 describes how the approach was used to create an initial recommendation for an enhanced TDD process involving CHA-AS-enhanced mutation analysis.

4.1 Design Science Research Overview

In a critical review of prior research published in the journal Information Systems Research, Orlikowski and Iacono (2001) note that the majority of information systems research is not clearly focused on the core subject-matter of the discipline—the IT artifact. They issue a plea for IS researchers to refocus their research efforts on the IT artifact rather than on issues that might be described as peripheral to this core. While any
research method can be used to explore research questions related to the IT artifact (Robey, 1996), the design science research method is a research paradigm focused specifically on the construction and evaluation of IT artifacts.

Although systems development by itself is not research, the systems development process is an important component of IS theory development and is critical to IS research (Nunamaker, 1992; Nunamaker, Chen, & Purdin, 1991). Figure 4.1, adapted from (Nunamaker, Chen, & Purdin, 1991), illustrates the important role of systems development in IS research. Systems development informs theory building and produces IT artifacts upon which theories can be tested and refined using observation, experimentation, and other research methods.

Figure 4.1: Role of Systems Development in IS Research

Design science has been defined as "the other side of the IS research cycle, [which] creates and evaluates IT artifacts intended to solve identified organizational problems"
(Hevner, March, Park, & Ram, 2004). An important distinction can be made between design science research and system building (or engineering). Design science research makes a "clear…contribution to the archival knowledge base of foundations and methodologies" whereas system building "is the routine application of the knowledge base to known problems" (Hevner, March, Park, & Ram, 2004).

Hevner et al. specify the following seven guidelines that must be addressed in some way by design science research in order to be considered complete:

1) Design as an Artifact
2) Problem Relevance
3) Design Evaluation
4) Research Contributions
5) Research Rigor
6) Design as a Search Process
7) Communication of Research

The following sections briefly define and describe each of the seven guidelines.

4.1.1 Design as an Artifact

Design science research produces IT artifacts (Orlikowski & Iacono, 2001). Although IT artifacts include systems, parts of systems, and algorithms; artifacts are not limited to software. IT artifacts also include "the constructs, models and methods applied in the development and use of information systems" but not the "people" and
"organizations" aspects of the traditional definition\textsuperscript{10} of the IS discipline (Hevner, March, Park, & Ram, 2004). However, not all IT artifacts constitute IS research. For the construction of an artifact to be considered IS research, there must be significant doubt about the feasibility of constructing the artifact or the ability for the artifact to perform in a way that makes is useful (Hevner, March, Park, & Ram, 2004). The creation of the artifact in design science research then becomes a "proof by construction" (Nunamaker, Chen, & Purdin, 1991).

The IT artifacts produced by design science research are often prototypes or incomplete "proofs of concept" that demonstrate the ability to solve a particular problem, but may not be a complete instantiation of a solution (Hevner, March, Park, & Ram, 2004).

4.1.2 Problem Relevance

Design science research addresses in some way problems that are relevant to the community of IS practitioners, including those who "plan, manage, design, implement, operate, and evaluate information systems and those who plan, manage, design, implement, operate, and evaluate the technologies that enable their development and implementation" (Hevner, March, Park, & Ram, 2004). This latter group is the community of practitioners to which the IT artifacts developed in this dissertation are relevant.

\textsuperscript{10} The realm of IS research is at the confluence of People, Organizations, and Technology (G. Davis & Olson, 1985; Lee, 1999).
4.1.3 Design Evaluation

Design science research, along with all other forms of research, must include an evaluation to determine the extent to which the research problem is addressed. Evaluation involves the quantitative or qualitative analysis of data and usually involves more traditional forms of research such as observation and experimentation (Hevner, March, Park, & Ram, 2004; Nunamaker, Chen, & Purdin, 1991).

4.1.4 Research Contributions

Design science research must make a clear contribution to the body of knowledge within the IS discipline; otherwise, it is simply engineering (Hevner, March, Park, & Ram, 2004).

4.1.5 Research Rigor

Design science research does not imply a lack or reduction of research rigor. Rigorous principles of artifact construction and evaluation must be applied in design science research, just as rigorous principles of experiment design and data analysis must be applied in experimental research (Hevner, March, Park, & Ram, 2004).

4.1.6 Design as a Search Process

Design science research is an iterative search for solutions to the problems of interest. Each iteration should add to the base of knowledge within the discipline and should build on prior work previously added to this base of knowledge (Hevner, March, Park, & Ram,
As with any other type of research, design science research is a search for answers to questions or solutions to problems.

4.1.7 Communication of Research

No research can be considered complete until it is communicated to the relevant audience. Design science research should be communicated to the relevant communities of both researchers and practitioners (Hevner, March, Park, & Ram, 2004).

4.2 Design Science Development of a Change Impact Analysis Algorithm

Chapter 5 describes the construction and evaluation of a new IT artifact—a change impact analysis algorithm applicable for situations in which a clearly defined starting point for the algorithm does not exist or is not known at the time when change impact analysis needs to be applied. This section describes the approach used to construct and evaluate the algorithm, and how this approach satisfies the seven guidelines of IS design science research.

4.2.1 Design as an Artifact

The Class Hierarchy Analysis, Arbitrary Start (CHA-AS) algorithm is an IT artifact that is useful to software developers and others who need to be able to assess the impact of a set of changes to program source code. The algorithm was developed as an extension to the well-known CHA algorithm (Dean, Grove, & Chambers, 1995), and uses the same class hierarchy analysis principle for identifying the methods and fields impacted by a set of source code changes. The algorithm differs from other algorithms
based on CHA (such as RTA) in that it does not assume a starting point that is known to be a program entry point. CHA-AS takes a set of modified methods and fields as input and creates and traces a call graph both forward and backward from these starting points to identify all potentially impacted code.

Although CHA-AS is intended to be a general purpose change impact analysis algorithm, it has been developed with a specific use in mind—the application of mutation analysis to iterative, test-first programming methods such as TDD.

4.2.2 Problem Relevance

The CHA-AS algorithm is relevant to the software development community because it addresses an important problem in the domain of change impact analysis that has not been specifically addressed by existing algorithms. See section 1.2.2 for details of the specific problem being addressed and the research questions and hypotheses being tested.

4.2.3 Design Evaluation

The usefulness of the algorithm was evaluated in two ways. First, a simulation experiment was performed to assess the algorithm's usefulness by determining whether a significant number of methods can be identified as "un-impacted", whether a significant number of mutants can be saved (not generated or executed), and whether there is a significant CPU time savings from using the algorithm. Second, a complexity analysis was performed on the data generated for the simulation experiment to estimate the algorithm's time and space complexity.
4.2.3.1 Simulation Experiment

Simulation experiments can be defined as experiments executed against models or simulations of algorithms, systems, or other artifacts of interest. One of the benefits of simulation experiments is that they can be an inexpensive way to test theory (Lehman, 1977, pp. 1-13). In this case, the current implementation of CHA-AS as described in pseudo-code in Appendix F is the model, and the purpose of the simulation experiment is to test the theory that change impact analysis can be used to improve the performance of mutation analysis in an iterative, test-first environment. The current implementation of the CHA-AS algorithm is intended to be a "proof by construction" of the theory.

After construction of the algorithm, the experiment was conducted by first creating a source code repository from an extensive TDD programming example from the book "Test-Driven Development: A Practical Guide" (Astels, 2003). The book devotes 233 pages to showing a step-by-step example of how to create a movie tracking system using TDD. The example shows the complete set of source code and tests to be written for each of 106 iterations. These iterations were coded in Java one at a time and saved in a source control system. Four additional iterations were added by dividing iterations 96 and 106 from the Astels book into three separate iterations each, bringing the total number of iterations to be used in the experiment to 110. The iterations were divided to bring them into conformance with the TDD process outlined by Beck (2002) and illustrated in figure 2.2. In both cases, the iterations from the book contained multiple test changes and multiple source code edits that were more properly handled as separate iterations.
An additional Eclipse Plug-In was created to manage the execution of the experiment. This program will be called the Simulation Program. The Simulation Program provided a user-interface consisting of a set of context-sensitive menu options in Eclipse's Project Explorer, for selecting an original program iteration and a comparison iteration as illustrated in figure 4.2.

![Simulation Program Interface](image)

**Figure 4.2: Simulation Program Interface**

To run the Simulation Program, an iteration to be used as the original project is selected in Eclipse's Project Explorer and then set as the original project by selecting the 'Set Original Project' menu item. The comparison project iteration is then selected in
Project Explorer and the ‘Comparison mutate against <original project>’ menu item is selected. The Simulation Program then executes to generate experiment data for the comparison project in the following steps: 1) execute MuJava to generate and execute all mutants for the comparison project; 2) execute Chianti to generate inputs for CHA-AS; 3) execute CHA-AS to determine the set of impacted methods and fields; 4) Determine the impacted mutants from the output of MuJava and CHA-AS and calculate and report experiment data. Figure 4.3 illustrates these steps of the Simulation Program and the interaction between the programs involved in the simulation experiment as a component-level UML Collaboration Diagram. In addition to controlling the interaction between the programs, the Simulation Program acts as a central point for collecting experiment data generated by each of the other programs and included in the data reported in step 4. Subsequent paragraphs describe each of these steps in greater detail.
Two additional menu items are shown in figure 4.2. The 'Mutate Project' menu item is used to invoke MuJava to generate and execute mutants for the first iteration. This is a separate process because for the first iteration, there is no other iteration against which to compare, so only the first step is relevant. The 'Compare existing mutation to <original project>' menu option is for testing purposes only. This option is used to compare mutants generated during a previous execution of the iteration to the original project. This option is a shortcut to eliminate the need to continually regenerate and execute mutants for a given iteration during testing.

The simulation experiment was conducted by first selecting 'Mutate Project' and then 'Set Original Project' for the first iteration of the data set, and then comparing each subsequent iteration to the iteration that preceded it. This resulted in the collection of 109
observations (iterations 2 – 110) for evaluation of the CHA-AS algorithm. The data for each observation consists of both the standard MuJava result and the result of applying CHA-AS to MuJava, making the experiment a within-subjects study.

Simulation Program step 1 involves executing MuJava to generate all mutants of the iteration representing the comparison project. The version of MuJava used for the experiment was a modified version of the program reported in (Offutt, Ma, & Kwon, 2004). Three main modifications were made to MuJava. First, Java ClassLoaders were used to eliminate the cumbersome setup requirements of the original program and to allow MuJava to execute and report results without copying source, byte and test code into specific directories. Second, an issue dealing with the termination of threads that execute mutants was corrected, resulting in a performance increase of more than a factor of 10 on a small sampling of the Astels data. Third, instrumentation was added to allow MuJava to report mutant execution and generation time information. None of these modifications constitute a design science research contribution according to the principles and guidelines reported in section 4.1, so a detailed evaluation of the changes was not performed and additional details of the modifications are not included in the dissertation.

Simulation Program step 2 consists of executing Chianti with the specified original and comparison project iterations as input to identify the set of atomic changes made between the original and comparison project. Changes are calculated for both the source and test code of the projects. See section 2.4.1 for additional details of Chianti and its output. The only modifications made to Chianti for this experiment were the addition of
instrumentation to report execution time, and the removal of some logic for generating dependencies between atomic changes which is not needed by CHA-AS.

Simulation Program step 3 is the execution of the CHA-AS algorithm with the atomic changes calculated by Chianti provided as input. CHA-AS calculates the complete set of potentially impacted methods and fields resulting from both source and test edits. Details of the CHA-AS algorithm are provided in chapter 5.

Simulation Program step 4 used output from MuJava, Chianti, and CHA-AS to calculate and report the experiment results for each iteration. The program appends data for each iteration to a .csv file to make the data available for analysis. Because CHA-AS is intended to be a general purpose change impact analysis algorithm, it does not contain logic to determine the impacted mutants from a set of impacted methods and fields. This was left to the Simulation Program to calculate in step 4. In a production version of CHA-AS enabled mutation, the mutation program would make this calculation from the CHA-AS output. A description of how to do this is included in chapter 6. Time required by the Simulation Program to perform this lookup was included in the total time reported for CHA-AS enabled mutation.

4.2.3.2 Measurement

Four hypotheses, H5 – H8 were tested in this part of the dissertation. Hypothesis testing consisted primarily of comparing the means between three pairs of variables. All variables are measurements taken on the iterations of the movie tracking system which was the subject of analysis by the simulation experiment. The first pair consists of measures of the total number of methods in the project (Total_Method_Count) and the
number of methods impacted by changes made between the current and previous iterations (Combined_Impacted_Method_Count). The second pair consists of measures of the number of mutants in the project (Total_Mutant_Count) and the number of mutants impacted by changes made between the current and previous iterations (Impacted_Mutant_Count). The last pair consists of measures of the total execution time required by MuJava (Combined_MuJava_Execution_Time) and the total execution time required by using the CHA-AS algorithm with MuJava (Adjusted_Algorithm_Execution_Time). Because MuJava does not mutate methods in GUI classes, abstract classes, or interfaces, these non-mutatable methods were subtracted out of the total and impacted method count to create a fourth pair (Total_Mutable_Method_Count and Impacted_Mutable_Method_Count) that was also analyzed.

H5 hypothesizes that change impact analysis can reduce the number of methods on which mutation analysis needs to be performed. This hypothesis can be restated to say that change impact analysis can identify a significant number of methods that are not impacted during the iterations of a TDD implementation—without reference to mutation analysis. Therefore, H5 relates primarily to the general usefulness of the CHA-AS algorithm, without regard to CHA-AS's applicability to mutation analysis. The testing of H5 involved the comparison of the means for the total number of methods in the iteration and the number of impacted methods reported by CHA-AS. One problem that had to be addressed for the restated version of the hypothesis is that an algorithm that simply returns zero as the number of impacted methods would result in the confirmation of the
hypothesis. Therefore, the Live_Mutant_Error_Count and Killed_Mutant_Error_Count variables, intended primarily for testing H6, were also used during testing of H5 to determine whether a significant number of precision or recall errors exist in the impacted method lists reported by CHA-AS. Live_Mutant_Error_Count is a measure of precision (the number of "killed" mutants reported as "live") whereas Killed_Mutant_Error_Count is a measure of recall (the number of "live" mutants reported as "killed").

H6 hypothesizes that change impact analysis can reduce the number of mutants that need to be executed in a TDD environment without reducing mutation analysis accuracy. To test H6, the mean of the total number of mutants was compared to the mean of the number of impacted mutants reported by CHA-AS. Live_Mutant_Error_Count and Killed_Mutant_Error_Count were used to determine whether a significant number of precision and recall errors exist in the reporting of mutant results.

H7 hypothesizes that the mutation reduction benefit of applying change impact analysis to mutation analysis in a TDD environment is positively correlated with program size. H7 was tested by checking for a positive correlation between the difference of each of the four pairs of variables used for hypothesis testing and the number of non-commentary source statements (NCSS) existing in the comparison project.

H8 hypothesizes a mutation analysis cost savings resulting from the use of CHA-AS. H8 was tested by comparing the mean MuJava execution time to Adjusted_CHAAS_Execution_Time, where Adjusted_CHAAS_Execution_Time is the total time required to execute the four steps of the Simulation Program minus the time required to generate and execute each mutant identified by CHA-AS as un-impacted.
4.2.3.3 Complexity Analysis

Average-case complexity analysis is calculated from a curve fitted to the expected probability distribution of the algorithm's time or space usage (Leiss, 2007, p. 11). Calculating an expected probability distribution for CHA-AS' time and space usage is beyond the scope of this dissertation, so the distribution obtained from executing the Astels data was used as a substitute—resulting in an estimated average-case complexity analysis. The analysis was performed on execution time and memory usage data generated while running the simulation experiment on the Astels data. The maximum memory usage data used for the space complexity analysis were collected using Java's built-in memory management API.

Both the time and space complexity analysis were performed by separately plotting in MatLab (MathWorks, 2007) the complexity variables (Combined_MuJavaExecution_Time and Combined_Change_Impact_Execution_Time for time, and Combined_MuJava_Peak_Memory and Change_Impact_Peak_Memory for space) on the y axis with NCSS on the x axis. MatLab's curve fitting module was used to fit a curve to the data plots and create an equation for the curve.

The least-squares method was used to determine the best fitting curve within each equation class. First through sixth order polynomial equations and an exponential (quadratic) equation were compared for each dependent variable. A visual inspection of the scatter plots indicated that logarithmic equations were not appropriate for any of the variables.

MatLab's curve fitting module calculated the best equation within each equation class and then a manual multi-step process of elimination was used to select the appropriate
equation as recommended in MatLab's curve fitting user's guide (Curve Fitting Toolbox: For Use with MatLab, 2006) and supported by Daniel and Wood (1980). Since the curve fitting procedure was used for complexity analysis, the objective was to find the lowest-order equation that adequately described the data rather than the best fitting curve. The first elimination step was to extend the range of y in the visual representation of the curve to approximately four times the maximum value of NCSS and eliminate any equation that exhibited a sudden downward trend beyond the range of predictor data. This is appropriate because a curve that exhibits such a sudden downward trend cannot be a reasonable representation of the population of interest.

Next, the equations were sorted on two goodness-of-fit statistics (Adjusted R^2 and Sum of Squared Errors) to identify equations that represent an obvious worse fit than other equations. Then, for the remaining equations, the 95% confidence region for the equation's coefficients were examined to identify the equation with the lowest range of coefficient values and any with excessively high ranges compared to the other remaining equations. In most cases the elimination procedure easily identified a single equation as a better choice than any other. The choice of equations was not obvious for only the Combined_MuJava_Peak_Memory variable. The quadratic equation represented the best fit, but the linear equation was reasonably close, so a conservative approach of selecting the lower order linear equation was chosen.

After selecting the most appropriate equation, both the time and space complexity for each variable were reported in Big-O notation by dropping the coefficients and all but the

4.2.4 Research Contributions

CHA-AS has applications not only for iterative software development and mutation analysis, but also framework development for which there is no single identifiable program entry point. The fact that no previous change impact analysis algorithm operates from a set of arbitrary starting points shed significant doubt on whether an efficient algorithm could be constructed to solve this problem; therefore, the development and evaluation of CHA-AS is a significant design science research contribution.

4.2.5 Research Rigor

CHA-AS was developed using the formal TDD approach developed by Beck (2002) and described in this dissertation. As described in section 4.2.3 and chapter 5, a rigorous process of statistical and algorithm analysis, based on prior research, was followed in evaluating CHA-AS and its applicability to mutation analysis.

4.2.6 Design as a Search Process

The development of CHA-AS was an iterative search process starting with the well-known CHA algorithm. Additional research will most likely improve the performance and precision of the algorithm, and will represent additional design science research iterations in the search for better solutions to the problem addressed by CHA-AS.
4.2.7 Communication of Research

The reporting in this dissertation of the construction and evaluation of the algorithm satisfies the minimum requirement for reporting of design science research results. However, additional improvements and evaluations of the algorithm and its applicability to iterative, test-first programming, and mutation analysis are planned for future research as reported in chapter 6 and section 7.3. These and future research results will be reported in both scholarly and practitioner journals.

4.3 Design Science Application of Mutation Analysis to Iterative, Test-First Programming

Although CHA-AS is a general purpose change impact analysis algorithm, it was designed for the specific purpose of supporting the application of mutation analysis to TDD and other iterative, test-first programming methods. The simulation experiment described in section 4.2.3.1 not only tested hypotheses related to the general usefulness of CHA-AS (H5 and H7), but also hypotheses related to the applicability of CHA-AS to mutation analysis (H6 – H8). As reported in chapter 5, the application of change impact analysis was shown to be a viable method of improving mutation analysis performance when applied to TDD.

These results led to the beginning of another iteration in the design science exploration of how TDD and other iterative, test-first programming methods can be improved by the application of change impact analysis and mutation analysis. The design science artifact to be created and evaluated in this iteration is an enhanced TDD method.
The relevance of the problem has already been described in chapter 1. Chapter 6 contains an initial recommendation of an enhanced TDD method incorporating the research results from chapter 5. However, the rigorous evaluation of the proposed method is a topic for future research—to be conducted after the creation and evaluation of development tools incorporating CHA-AS-enabled mutation for use on real, iterative, test-first software development projects.
CHAPTER 5:  

CHA-AS: AN ARBITRARY STARTING POINT CHANGE IMPACT ANALYSIS ALGORITHM

CHA-AS is a change impact analysis algorithm that identifies a safe set of methods and fields potentially impacted by changes made between two versions of a program. This chapter includes a detailed description of the algorithm in section 5.1. Section 5.2 presents results from the simulation experiment described in sections 4.2.3.1 and 4.2.3.2. Section 5.3 presents results of a complexity analysis of the algorithm. Section 5.4 discusses the implications and potential uses of the algorithm and section 5.5 contains summary information and future research directions related to the algorithm.

5.1 Algorithm Description

The prototype version of the CHA-AS algorithm was implemented as an Eclipse Plug-In that takes the names of two Eclipse projects from the current Eclipse workspace as input—one identified as the original program and the other identified as the edited program—and produces sets of impacted methods and fields as output. See Appendix F for a complete set of pseudo code for the algorithm.

The algorithm starts by comparing the original and edited projects and initializing a queue with information about each method that was either directly modified or directly impacted by source code edits made in the edited project. The initialization process is described in section 5.1.1. The queue is then processed sequentially to identify all methods potentially impacted by any method in the queue. Any potentially impacted
methods are also added to the queue so their impact can be included in the final result. This results in a recursive expansion of the queue until all potentially impacted methods have passed through the queue. The queue is processed until empty, with each method that is removed from the queue being added to a set of impacted methods. During initialization and processing of the queue, potentially impacted fields are also identified and added to an impacted fields set. Figure 5.1 presents a UML Activity Diagram containing a high-level description of this process.
The "Poll Queue" activity removes the next item from the queue. The removed item is the impacted method referenced in the remaining activities. Each impacted method may have a "ripple effect" that impacts other methods. Impacted methods that set field values may impact any other method that accesses those fields. The processing of fields set by impacted methods is described in section 5.1.2. Impacted methods may also impact any method that invokes the impacted method and any method invoked by the impacted method. Invoking and invoked methods are handled by adding each to the
queue so their ripple effects can also be processed. This results in the call graph being traversed recursively both up and down from each impacted method. Recursion stops on each branch when either no more invoking or invoked methods exist or when a method is found that does not exist in a class of the project being traversed.

When the queue is empty, the algorithm completes by returning the impacted method and impacted field sets. These two sets constitute a safe listing of the impact of the edits in the edited project—subject to the limitations described in the limitations section of chapter 7.

5.1.1 Initialize Impacted Method Queue

The "Initialize Impacted Method Queue" activity identifies the set of methods and fields directly modified or directly impacted by first invoking Chianti to identify a complete set of atomic changes describing the changes made in the edited project. These atomic changes are then used to identify a set of starting point methods to be added to the impacted method queue, and a set of starting point fields to be both added to the impacted fields set and to be processed to identify additional methods for the impacted method queue. The queue initialization activity is described in the following paragraphs and is also presented as an Activity Diagram in figure 5.2.
Starting point methods are all the methods identified by Chianti as either added, changed, or lookup changed, as well as any methods impacted by deleted methods.
Lookup changed methods are methods that are not changed in the edited project, but have had their virtual method invocation semantics changed by edits in the edited project. Methods impacted by deleted methods include any methods referencing fields that had been set by the deleted method, any methods that had been invoked by the deleted method, and any methods that had been invoking the deleted method. These starting point methods are all added to the impacted method queue.

Starting point fields are all the fields identified by Chianti as either added or changed. For the current version of the algorithm, a simplifying assumption was made that all fields are private. This was true in the Astels data used to test the algorithm. With this assumption in place, deleted fields need not be handled separately because only deleted fields referenced by other non-deleted methods or fields can affect the set of impacted methods or fields. Any deletions would require changes to the references, which would be listed by Chianti as changed methods or fields, or would result in Chianti reporting lookup changed methods which are handled elsewhere. The starting point fields are added to the impacted fields set, which is then recursively expanded to include any fields initialized by an impacted field. Any methods accessing any of the fields from the recursively expanded set are then identified and added to the impacted method queue.

The initialization activity also traverses the abstract syntax trees for all classes in both the original and edited projects to identify all method and field references. Method references are method invocations (including constructor invocations) from other methods or from field initializers. Field references are statements that access or set fields either from other methods or from field initializers. The current implementation of the
algorithm ignores class and instance initializers. Method and field references are cached for later use by the algorithm.

5.1.2 Process Fields Set by Impacted Method

Any method that accesses a field that is set by an impacted method is potentially impacted. This case is handled by first adding to a set of modified fields, all fields set by an impacted method and all fields initialized by an invocation of an impacted method. The set of modified fields is then processed by adding each method referencing one of these modified fields to the impacted method queue and each modified field to the impacted fields set. This process is illustrated as an Activity Diagram in figure 5.3.
Figure 5.3: Process Set Fields Activity Diagram
5.1.3 Method Processing Ranges

For simplicity in describing the algorithm, the previous sections and activity diagrams assumed that whole methods are always added to the impacted method queue. However, to reduce the ripple effect of impacted methods, sub-ranges of the code in impacted methods are added to the queue instead of entire methods whenever possible. These sub-ranges are referred to in the pseudo-code of Appendix F as method processing ranges. When entire methods are added to the queue, they are added with a method processing range that includes all code in the method. When processing a method from the queue, only method invocations and field accesses that occur within the method processing range of the impacted method are used to determine additional methods that should be added to the queue.

Methods that are added to the queue because they invoke an impacted method or contain a reference to an impacted field are added to the queue with method processing ranges that do not include the entire method. These processing ranges start with either the first character of the first reference that caused them to be included as an impacted method or the first character of the top-most enclosing loop containing the reference, and extend to the end of the method. Methods that are added to the queue as a result of being invoked from within the processing range of an impacted method are added with a processing range encompassing the entire method.

The use of method processing ranges complicates the addition of methods to both the queue and the impacted methods set. When a method is identified as being impacted, the impacted methods set must be checked to see if the method was previously processed. If
the method was previously processed, the previously processed range is compared against
the range to be processed to determine whether processing of all or part of the newly
identified processing range is necessary. This may result in an adjustment of the
processing range before adding the method to the queue, or in not adding the method to
the queue. If an adjusted processing range for a previously processed method is added to
the queue, the processed range of the existing method in the impacted method set must be
expanded to include the newly processed range when the adjusted range is processed.

The queue must also be checked before adding an impacted method to the queue to
determine whether the queue already contains the method—possibly with a different
processing range. If so, the range of the existing entry in the queue is adjusted as
necessary to include the new processing range instead of adding another copy of the
method to the queue. The effect on the algorithm of handling method processing ranges
can be seen by comparing the pseudo code of Appendix F with the descriptions and
activity diagrams from the previous sections of this chapter.

5.1.4 CHA Reference Expansion

Another simplifying assumption of the previous sections describing the algorithm is
that the exact target of a method reference can always be determined statically (when the
source code is parsed). In object-oriented languages, this is often not the case because of
virtual method invocation. This requires the expansion of the set of possible target
methods for each method invocation to include all methods with matching signatures in
classes derived from the reference type appearing in the source code. This target method
expansion is the main feature of the CHA algorithm upon which CHA-AS is based, and
the feature from which CHA's name was derived. CHA-AS performs the same target
method expansion as CHA by adding a CHA expanded set of methods to the impacted
method queue in each case where methods are added to the queue. If the reference type
is an Interface, the CHA expanded set of methods includes all matching methods from
any class implementing the interface. The effect on the algorithm of CHA target method
expansion can be seen by comparing the pseudo code of Appendix F with the
descriptions and activity diagrams from the previous sections of this chapter.

5.2 Simulation Experiment Results

This section describes the results of the simulation experiment used to evaluate the
CHA-AS algorithm. The section starts with a presentation of descriptive statistics,
followed by an analysis of the model assumptions for the statistical analyses performed.
The section concludes with a presentation of hypothesis testing results and the results of
follow-up analysis prompted by these results.

5.2.1 Descriptive Statistics

Table 5.1 summarizes the minimum, maximum, mean, and standard deviations of
each of the eight comparison variables, and figures 5.4 – 5.7 present graphical
comparisons of each of the four pairs of variables used for hypothesis testing.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total_Method_Count</td>
<td>4</td>
<td>164</td>
<td>88.2294</td>
<td>41.3536</td>
</tr>
<tr>
<td>Combined_Impacted_Method_Count</td>
<td>0</td>
<td>102</td>
<td>47.7523</td>
<td>29.7397</td>
</tr>
<tr>
<td>Total_Mutable_Method_Count</td>
<td>4</td>
<td>88</td>
<td>49.2752</td>
<td>23.9378</td>
</tr>
<tr>
<td>Impacted_Mutable_Method_Count</td>
<td>0</td>
<td>76</td>
<td>33.6514</td>
<td>23.3716</td>
</tr>
<tr>
<td>Table 1: Comparison of Execution Times and Method Counts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------</td>
<td>----------------</td>
<td>---------</td>
<td>----------------</td>
</tr>
<tr>
<td>Total Mutant Count</td>
<td>0</td>
<td>259</td>
<td>101.8349</td>
<td>73.9320</td>
</tr>
<tr>
<td>Impacted Mutant Count</td>
<td>0</td>
<td>258</td>
<td>79.8899</td>
<td>79.9883</td>
</tr>
<tr>
<td>Combined MuJava Execution Time</td>
<td>1051</td>
<td>98643</td>
<td>35760.1101</td>
<td>29141.4623</td>
</tr>
<tr>
<td>Adjusted Algorithm Execution Time</td>
<td>2814</td>
<td>105049</td>
<td>33110.7890</td>
<td>31983.9503</td>
</tr>
</tbody>
</table>

**Figure 5.4:** Comparison of Total and Impacted Method Counts

**Figure 5.5:** Comparison of Total Mutatable and Impacted Mutatable Method Counts
5.2.2 Model Assumptions

Because of the cumulative nature of the experimental data (resulting from the iterative nature of the process under analysis), there is no mean about which the values of
the dependent variables tend to congregate. As a result, the assumption of normality cannot and should not be expected to be satisfied, and tests requiring this assumption such as within-subject ANOVA, Student's Paired-T, and Pearson's Correlation are inappropriate. Therefore, non-parametric tests were used instead. The Wilcoxon Signed-Ranks Test was used to compare means and Spearman's Rank Correlation test was used to test for correlation.

Although not requiring a specific distribution, the Wilcoxon test does require independence of the differences between the means under comparison. The dependent variables under analysis in this study are significantly determined by the corresponding variables from the prior iteration, and therefore, should be expected to exhibit significant serial- or autocorrelation. Because of this autocorrelation, the variables are not independent. However, for each comparison, one of the two variables being compared is a measure from a control condition that extends through the entire series of data while the other variable is the same measure taken when a treatment condition is applied to the same data as the control condition. Therefore, even though autocorrelation exists in the data set, it has been controlled in the experiment design and should not be present in the paired differences calculated by and used in the Wilcoxon test.

To verify that the differences are in fact free of autocorrelation and that the independence assumption of the Wilcoxon test is satisfied, a re-sampling with replacement procedure was used whereby 500 random samples equal in size to the original data set were created. For each variable the mean and standard error of the
samples were calculated and compared to the mean and standard error of the original data set. These results are presented in table 5.2.

**TABLE 5.2: Comparison of Original and Re-sampled Means and Standard Errors**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Original Mean</th>
<th>Resampled Mean</th>
<th>Original SE</th>
<th>Resampled SE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method Count Difference</td>
<td>40.4770642</td>
<td>40.3118165</td>
<td>2.7748689</td>
<td>2.7325646</td>
</tr>
<tr>
<td>Mutatable Method Count Difference</td>
<td>15.6238532</td>
<td>15.5548073</td>
<td>1.5204344</td>
<td>1.4712049</td>
</tr>
<tr>
<td>Execution Time Difference</td>
<td>2649.321101</td>
<td>2648.32</td>
<td>1466.38878</td>
<td>1429.04</td>
</tr>
</tbody>
</table>

Since the original means and standard errors are almost identical to the re-sampled means and standard errors, the independence assumption of the Wilcoxon Signed-Ranks test appears to be satisfied.

In an attempt to be certain that the independence assumption was satisfied before proceeding, a Durbin-Watson test for autocorrelation of time-series data was also performed on the difference variables. The initial test reported significant autocorrelation on all four difference variables. However, after correcting for the autocorrelation, a subsequent paired-comparison of the means of each of the pairs yielded the same results at the same levels of alpha as the results reported before correcting for autocorrelation—providing additional evidence that findings of significance are not a result of the autocorrelation of the comparison variables.

### 5.2.3 Hypothesis Testing

The results of comparing each of the four pairs of variables using the Wilcoxon Signed Ranks Test are summarized in table 5.3. The comparisons of both of the method
count pairs and the mutant pair are significant at the 0.001 level of alpha, indicating that
the CHA2AS algorithm identifies a significant number of un-impacted methods and un-
impacted mutants. The savings in the number of methods is significant when all methods
are considered and when only mutatable methods are considered—indicating that the
algorithm is useful for both general change impact analysis use and application to
mutation analysis as hypothesized. However, the algorithm did not result in a statistically
significant execution time savings. These results indicate that hypothesis H5 is supported
and hypothesis H8 is not supported.

**Table 5.3: Results of Wilcoxon Signed Ranks Test Comparison of Means**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Rank Type</th>
<th>Count</th>
<th>Mean Rank</th>
<th>Sum of Ranks</th>
<th>Z Statistic</th>
<th>P Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method Count</td>
<td>Neg. Ranks</td>
<td>107</td>
<td>54.00</td>
<td>5778.00</td>
<td>-8.981</td>
<td>.000</td>
</tr>
<tr>
<td></td>
<td>Pos. Ranks</td>
<td>0</td>
<td>.00</td>
<td>.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ties</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutatable Method Count</td>
<td>Neg. Ranks</td>
<td>106</td>
<td>53.50</td>
<td>5671.00</td>
<td>-8.956</td>
<td>.000</td>
</tr>
<tr>
<td></td>
<td>Pos. Ranks</td>
<td>0</td>
<td>.00</td>
<td>.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ties</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutant Count</td>
<td>Neg. Ranks</td>
<td>75</td>
<td>38.00</td>
<td>2850.00</td>
<td>-7.614</td>
<td>.000</td>
</tr>
<tr>
<td></td>
<td>Pos. Ranks</td>
<td>0</td>
<td>.00</td>
<td>.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ties</td>
<td>34</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution Time</td>
<td>Neg. Ranks</td>
<td>28</td>
<td>93.18</td>
<td>2609.00</td>
<td>-1.175</td>
<td>.120</td>
</tr>
<tr>
<td></td>
<td>Pos. Ranks</td>
<td>81</td>
<td>41.80</td>
<td>3386.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ties</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For hypothesis H6 to be supported, the number of un-impacted mutants must be
significant and the number of precision and recall errors must not be significant. The
sum across all iterations of the number of mutants is 11,100. Of this total number of
mutants, only one was incorrectly classified as "live", and this misclassification was a
result of a callback method call which is described in section 7.2.2 as a known limitation
of virtually all change impact analysis algorithms. No mutants were incorrectly classified as "killed". Therefore, hypothesis H6 is supported.

As summarized in the Spearman Correlation data of table 5.4, the differences between both method count pairs and the mutant count pair are positively correlated with NCSS. The correlations were significant at the 0.001 level of alpha, indicating that hypothesis H8 is supported. Because the algorithm execution time was not found to be significantly different from the MuJava execution time, an execution time difference was not tested for correlation with NCSS.

### Table 5.4: Paired Differences to Program Size Correlations

<table>
<thead>
<tr>
<th>Difference Variable</th>
<th>Correlation Coefficient</th>
<th>P-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method Count</td>
<td>.809</td>
<td>.000</td>
</tr>
<tr>
<td>Mutatable Method Count</td>
<td>.623</td>
<td>.000</td>
</tr>
<tr>
<td>Mutant Count</td>
<td>.571</td>
<td>.000</td>
</tr>
</tbody>
</table>

5.2.4 Follow-Up Analysis

Although both the number of un-impacted methods and un-impacted mutants identified by the algorithm is significant, the algorithm did not reduce the number of mutants on a large percentage (28.44%) of the iterations, and the algorithm reduced the number of mutants by less than 10% on 70.64% of the iterations. This prompted a re-analysis of the execution time variable to determine if H8 would have been supported if only cases where the algorithm reduced the number of mutants by at least 10% were considered.

Exclusion of the cases where more than 90% of the mutants were identified as impacted resulted in a finding of significance on the Wilcoxon Signed Ranks test
between the execution time variables as summarized in table 5.5. Descriptive statistics for the execution time variables after exclusion are presented in table 5.6.

**TABLE 5.5: Results of Wilcoxon Signed Ranks Test on Reduced Data Set Execution Time Pair**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Rank Type</th>
<th>Count</th>
<th>Mean Rank</th>
<th>Sum of Ranks</th>
<th>Z Statistic</th>
<th>P Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time</td>
<td>Neg. Ranks</td>
<td>28</td>
<td>15.50</td>
<td>434.00</td>
<td>-4.681</td>
<td>.000</td>
</tr>
<tr>
<td></td>
<td>Pos. Ranks</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ties</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 5.6: Descriptive Statistics for Reduced Data Set Execution Time Variables**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combined_MuJava_Execution_Time</td>
<td>5418</td>
<td>89970</td>
<td>31463.5862</td>
<td>18750.2998</td>
</tr>
<tr>
<td>Adjusted_Algorithm_Execution_Time</td>
<td>2814</td>
<td>19159</td>
<td>9407.6552</td>
<td>4722.5975</td>
</tr>
</tbody>
</table>

The execution time savings is also positively correlated with program size, with a Spearman Correlation Coefficient of .874 and significance at the .001 level of alpha.

5.3 Complexity Analysis Results

An estimated average complexity analysis of the simulation experiment data indicates that the CHA2AS algorithm is more efficient in both execution time and memory usage than MuJava. This supports the theory that change impact analysis can be used to improve mutation analysis performance in iterative development environments such as TDD. The following subsections describe the result of both the time and space complexity analysis.
5.3.1 Time Complexity Analysis Results

The purpose of the time complexity analysis was to get some insight into the scalability of the CHA-AS algorithm and to compare the combined execution time required by MuJava to generate and execute mutants (represented by the Combined_MuJava_Execution_Time variable) with the time required by CHA-AS to determine the change impact of code changes between iterations (represented by the Combined_Change_Impact_Execution_Time variable).

Combined_Change_Impact_Execution_Time is made up of two components: the time required to execute Chianti (Chianti_Execution_Time) and the time required to derive the change impact from the Chianti output (Chaas_Execution_Time).

All four of the time complexity variables are a function of the number of non-commentary source statements in the code (NCSS_Count). The first step in the complexity analysis involved using MatLab's (MathWorks, 2007) Curve Fitting Toolbox to derive functions describing the data that resulted from plotting each of the time complexity variables against NCSS for each iteration of the simulation experiment data. Section 4.2.3.3 describes the procedure followed in deriving the functions. Figures 5.8 – 5.11 present the scatter plots for the four time complexity variables and the corresponding curve derived to describe each plot.
Figure 5.8: Curve Fit for MuJava Execution Time

Figure 5.9: Curve Fit for Combined Change Impact Execution Time
Figure 5.10: Curve Fit for Chianti Only Execution Time

Figure 5.11: Curve Fit for CHA-AS Only Execution Time
The data point in the upper right corner of figure 5.10 was treated as an outlier and did not affect the curve. Notice that this point resulted in figures 5.10 and 5.11 being presented at different scales. The slopes of the curves represented by these two figures (which when combined constitute the combined change impact execution time) are actually very similar. Table 5.7 presents the equations and goodness of fit statistics—Adjusted $R^2$ and Sum of Squared Errors (SSE)—representing each of the derived curves. For each equation, x represents NCSS.

**Table 5.7: Time Complexity Equations and Fit Statistics**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Equation</th>
<th>$Adjusted R^2$</th>
<th>SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MuJava Execution Time</td>
<td>$f(x) = 0.1652x^2 - 2.914x + 4407$</td>
<td>0.9217</td>
<td>7.05E+09</td>
</tr>
<tr>
<td>CHA-AS Execution Time</td>
<td>$f(x) = 7.293x + 1920$</td>
<td>0.9063</td>
<td>2.63E+07</td>
</tr>
<tr>
<td>Chianti Only Execution Time</td>
<td>$f(x) = 3.02x + 486$</td>
<td>0.7169</td>
<td>1.70E+07</td>
</tr>
<tr>
<td>CHA-AS Only Execution Time</td>
<td>$f(x) = 4.274x + 1433$</td>
<td>0.9766</td>
<td>2.14E+06</td>
</tr>
</tbody>
</table>

Dropping all coefficients and all but the highest order term from the equations makes it clear that the estimated average time complexity of MuJava is $O(n^2)$ (quadratic) and that the estimated average time complexity of CHA-AS is $O(n)$ (linear). The time complexity result for MuJava is also consistent with that reported by Offut et al. (1996). Figure 5.12 illustrates the difference in time complexity between the two algorithms.
5.3.2 Space Complexity Analysis Results

As with the time complexity variables, the space complexity variables are all a function of NCSS. The peak random access memory used by MuJava during generation and execution of mutants (Combined_MuJava_Peak_Memory), and the peak random access memory used to determine the change impact (Change_Impact_Peak_Memory) are the primary variables of interest for space complexity. Because for each observation, the change impact peak memory is the maximum of the Chianti peak memory usage and the CHA-AS only peak memory usage, both Chianti_Peak_Memory and Chaas_Peak_Memory are also included in the analysis.

Figures 5.13 – 5.16 present the scatter plots and corresponding curves for the space complexity variables.
Figure 5.13: Curve Fit for MuJava Peak Memory Usage
Figure 5.14: Curve Fit for Combined Change Impact Peak Memory Usage

Figure 5.15: Curve Fit for Chianti Peak Memory Usage
Table 5.8 presents the equations and goodness of fit statistics for each of the derived curves. For each equation, x represents NCSS.

Table 5.8: Space Complexity Equations and Fit Statistics

<table>
<thead>
<tr>
<th>Variable</th>
<th>Equation</th>
<th>Adjusted $R^2$</th>
<th>SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MuJava Peak Memory</td>
<td>$f(x) = 104.9x + 48190$</td>
<td>0.7216</td>
<td>2.07E+10</td>
</tr>
<tr>
<td>CHA-AS Peak Memory</td>
<td>$f(x) = 55.36x + 29560$</td>
<td>0.8485</td>
<td>2.67E+09</td>
</tr>
<tr>
<td>Chianti Only Peak Memory</td>
<td>$f(x) = 48.72x + 29490$</td>
<td>0.7897</td>
<td>3.08E+09</td>
</tr>
<tr>
<td>CHA-AS Only Peak Memory</td>
<td>$f(x) = 55.85x + 28750$</td>
<td>0.8585</td>
<td>2.51E+09</td>
</tr>
</tbody>
</table>

The estimated average space complexity for both MuJava and CHA-AS is $O(n)$ (linear). However, as illustrated in figure 5.13, MuJava's y intercept and slope are both approximately double that of CHA-AS', indicating that CHA-AS is more efficient in memory usage than MuJava.
5.4 Discussion and Implications

The main implication of the simulation experiment and the complexity analysis is to show that CHA-AS is useful as a general purpose, arbitrary start, change impact analysis algorithm. The algorithm was shown to efficiently identify a significant number of un-impacted methods between program iterations using only linear time and space.

Strong evidence was also presented to indicate that the algorithm can be useful in improving mutation analysis performance in a TDD environment with only a modest improvement in precision. The algorithm did identify a significant number of un-impacted methods for which mutants did not need to be generated and executed. However, additional reductions in the number of methods identified as impacted will have to be made before the algorithm will be useful for improving mutation analysis performance in a TDD environment.
One opportunity for an improvement in precision is to reduce the number of invoking methods listed as impacted. Currently, all methods that invoke an impacted method are considered to be impacted themselves. However, not all methods are actually impacted as a result of calling impacted methods. Methods that invoke impacted methods that have no side-effects are not impacted unless they use return values from the impacted methods. Also, not all invoking methods are impacted by the side-effects of the methods they invoke. Similarly, methods that are currently considered impacted because they are invoked by an impacted method may not actually be impacted if the invoking method has no side effects and doesn't pass data impacted by the change to the invoked method. Modifications to the CHA-AS algorithm to address any of these issues may result in a significant increase in the algorithm's precision.

The linear time complexity of the CHA-AS algorithm compared to the quadratic time complexity of MuJava indicates that a significant amount of additional processing can be performed in CHA-AS to further reduce the reported number of impacted methods and still result in an execution time improvement when applied to mutation analysis. This is especially true on large projects, which are the ones most in need of the performance enhancement. The difference in time complexity, along with the finding of a significant execution time improvement when impacted mutants are reduced by at least 10%, justifies additional work on improving the precision of the algorithm.
5.5 Summary

The results of the simulation experiment and the algorithm analysis indicate that
CHA2AS is a safe and efficient algorithm for identifying the change impact between two
versions of a Java program. The algorithm executes in time and space that is linear in the
number of non-commentary source statements, making it practical for large projects.
Table 5.9 presents a summary of the results of testing four hypotheses related to the
algorithm. Limitations of the algorithm are described in section 7.2.2.

**Table 5.9: Summary of Hypothesis Testing Results**

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>H5</td>
<td>Change impact analysis can reduce the number of methods on which mutation analysis needs to be performed in a TDD environment.</td>
<td>Accepted <em>a</em></td>
</tr>
<tr>
<td>H6</td>
<td>Change impact analysis can reduce the number of mutants that need to be executed in a TDD environment without reducing precision or recall.</td>
<td>Accepted <em>a</em></td>
</tr>
<tr>
<td>H7</td>
<td>The mutation reduction benefit of applying change impact analysis to mutation analysis in a TDD environment is positively correlated with program size.</td>
<td>Accepted <em>a</em></td>
</tr>
<tr>
<td>H8</td>
<td>The application of change impact analysis to mutation analysis in a TDD environment will result in an overall mutation analysis cost savings.</td>
<td>Not Accepted <em>b</em></td>
</tr>
</tbody>
</table>

*a* at the 0.001 level of alpha.

*b* accepted at the 0.001 level of alpha when cases with > 90% impacted mutants are excluded.
CHAPTER 6:

APPLICATION OF CHA-AS TO MUTATION ANALYSIS IN AN ITERATIVE, TEST-FIRST DEVELOPMENT ENVIRONMENT

One of the main purposes of this dissertation is to improve the efficiency of mutation analysis to enhance its usefulness when applied to TDD and other iterative, test-first software development methods. Chapters four and five describe and evaluate a new change impact analysis algorithm and show that it has potential for improving mutation analysis efficiency when used in a TDD environment. This chapter has two purposes, the first is to describe how to apply the CHA-AS algorithm to mutation analysis, the second is to propose an enhanced TDD method that incorporates mutation analysis to improve the defect reduction capabilities of TDD and to improve the reliability and measurability of the method. The chapter also contains information on additional research and tool support necessary to improve TDD with mutation analysis.

6.1 Application of CHA-AS to Mutation Analysis

The purpose of the CHA-AS algorithm is to identify a safe set of impacted methods and fields resulting from source code edits in a project. When applied to mutation analysis, the output from CHA-AS can be used to determine which methods and fields in the edited project must be mutated and which can have their mutation results reported from the previous iteration. Methods and fields that are impacted by source code edits need to have their mutants generated and executed in the edited project; however, changes to the automated unit tests can also impact the mutations that need to be
performed in the edited project. Table 6.1 summarizes the mutation analysis impact of different kinds of test changes. Test changes are changes made to a test method, not a test suite. If constructors, setUp() methods, or tearDown() methods of a test suite are changed or impacted (collectively referred to as test fixture impacts), all test methods of the test suite class and any of its subclasses are considered impacted.

**Table 6.1: Mutant Impact of Test Changes**

<table>
<thead>
<tr>
<th>Test Change</th>
<th>Mutation Analysis Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Test</td>
<td>All &quot;Live&quot; mutants from the previous iteration derived from methods and fields still existing in the edited project and reachable from the added test</td>
</tr>
<tr>
<td>Delete Test</td>
<td>All mutants killed in the previous iteration by the deleted test derived from methods and fields that still exist in the edited iteration</td>
</tr>
<tr>
<td>Change (or impact) Test</td>
<td>The union of the other two cases</td>
</tr>
</tbody>
</table>

To apply CHA-AS to mutation analysis, CHA-AS must be executed twice for each iteration. During the first run, the impact analysis is performed on the original and edited project's source code directory(s) as described in section 5.1. During the second run, the impact analysis is performed on the original and edited project's test directory(s) to identify the set of source code methods and fields impacted by test changes. The impact analysis on the test directory(s) is the same as described in section 5.1 with one exception: if any test fixture impacts are identified, all tests in the corresponding test suite and any subclasses are treated as impacted.

The complete set of impacted methods and fields are identified by taking the union of the output from both runs. The impacted methods and fields from classes in the test directory(s) are then eliminated and mutants are generated and executed for the remaining impacted methods and fields. Finally, the set of un-impacted methods and fields are
identified by subtracting those identified as impacted from the total set of methods and 
fields, and a lookup is performed on the mutants of the original project to identify any 
that were created from methods and fields that exist in the edited project but were un-
impacted. The status ("live" or "killed") of these un-impacted mutants is reported from 
the result of the previous iteration. These reported but un-executed mutants are also 
included in mutant counts for the edited project and in the calculation of the mutation 
analysis score.

6.2 Enhanced TDD Method

TDD is one of the preeminent "agile" software development methods, and as such, is 
intended to be a light-weight method that is free of unnecessary structure and process. 
Agile methods adhere to the principles of the Agile Manifesto ("The Agile Manifesto") 
which advocates "individuals and interactions over processes and tools", "working 
software over comprehensive documentation", and "responding to change over following 
a plan". Many Agile proponents would object to the addition of complex structure to 
TDD or any other Agile method. However, in its current form, TDD is too loosely 
defined to be applied consistently across and within organizations, and to be measured 
and compared against other software development methods, so some additional structure 
is necessary.

Figure 6.1 presents an enhanced TDD method that adds a process for applying 
mutation analysis after the refactoring step, followed by a decision point to determine if 
the next TDD iteration should be started or the current one should be continued by adding
additional tests or enhancing existing ones. Although not specifically shown in the
diagram, CHA-AS can be integrated into a tool used to support the new mutation analysis
step following the procedure outlined in the previous section.

![Mutation Analysis Enhanced TDD Method Diagram](image)

**Figure 6.1: Mutation Analysis Enhanced TDD Method**

The decision point uses mutation analysis score (MAS) to determine if an iteration is
complete. The MAS threshold used to determine iteration completeness can be set for
individual organizations or projects and should be refined as experience is gained with
the process. In the absence of additional data, a MAS threshold of 0.9 can be used as a
starting point.

6.3 Additional Research

The enhanced TDD method is proposed as a starting point for additional research to
improve the method. Others have advocated the use of mutation analysis with TDD
(Beck, 2002), but additional research is needed to develop tools to support the enhanced method, to identify and evaluate other improvements to the method and how they should be applied—such as automated test-case generation, to determine appropriate criteria such as MAS scores for deciding whether to continue an iteration or start a new one, and to measure how effectively the process was followed. Additional research is also needed to compare TDD to other methods, to explore the effectiveness of combining TDD with other methods, to measure the relationship between mutation analysis score and defect reduction, and to identify the key success factors in using TDD to obtain defect reduction benefits, code quality improvements, and higher software development productivity.

6.4 Tool Support

Additional and improved tool support is needed to support mutation enhanced TDD. These tools should be integrated into popular IDE’s such as Eclipse and NetBeans, and may include a change impact analysis algorithm such as CHA-AS to improve performance.

Mutation analysis tools for supporting TDD should provide a way for programmers to mark equivalent mutants and have these mutants optionally excluded from reporting in future iterations. This would require the ability to track mutants across iterations and to identify whether mutants from two different iterations are the same. Tools should also provide feedback during the development process on how well the developer is applying TDD. For example, they should provide real-time code coverage and mutation analysis statistics to encourage the writing of more complete test suites. These tools should also
provide capabilities for performing post-hoc analysis to determine how well TDD was applied to a previous project or iteration.

The tools should have the capability to run in the background during the development process to perform real-time mutation analysis the way current IDE’s perform real-time compilation. This would give programmers more immediate feedback on the quality of their TDD development effort, and would result in a perception of enhanced mutation analysis performance by offloading some mutation analysis processing to periods of inactivity during development iterations instead of waiting until the iterations are complete.

6.5 Summary

This chapter has explained how to apply the CHA-AS change impact analysis algorithm to mutation analysis and has proposed an enhanced TDD method that uses mutation analysis to improve the process and to make it more measureable, more reliable, and more effective as a defect reduction method. The application of mutation analysis to TDD requires improved tool support. Some of the features of these required tools have been described. Development and evaluation of a new CHA-AS enabled mutation analysis tool that supports some or all of these features is a topic for future research intended to be performed in the ongoing design science research exploration of how TDD can be improved.

TDD is a relatively new method of software development with numerous opportunities for research-based improvements. This chapter has proposed a few
directions for additional TDD research both in the method, and in tools to support the method. Similar research opportunities exist for other Agile methods.
CHAPTER 7:

CONCLUSIONS

This dissertation consists of two main parts. The first part is a quasi-experimental comparison of the defect reduction capabilities of Software Inspection and TDD. The second part is a design science exploration of how both TDD and mutation analysis can be improved by the application of change impact analysis. Both parts of the dissertation resulted in significant research contributions and both parts generated additional research questions to be explored in future research. This chapter contains a summary of the research contributions reported throughout the dissertation, an explanation of limitations from both parts of the dissertation, and a description of future research related to one or both parts of the dissertation.

7.1 Summary of Research Contributions

The following are the specific research contributions provided by this dissertation:

1) A comparison of the software defect reduction benefits and initial development costs of software code inspection and TDD.

2) Creation and evaluation of a safe, arbitrary starting point, change impact analysis algorithm for Java.

3) An evaluation of the applicability of change impact analysis to mutation analysis in an iterative, test-first environment.

4) Proposal of an enhanced TDD method that incorporates mutation analysis to reduce software defects and increase the measurability of TDD.
5) A list of additional research and tool enhancements needed to continue to improve TDD and make it more effective, measurable, and reliable.

The first part of the dissertation resulted in significant findings that software inspection is more effective than TDD at reducing software defects but also more expensive. These findings are both important and timely because of the current high adoption rate of TDD as part of the Extreme Programming software development methodology. This provides a valuable piece of information to software developers and managers, to assist them in choosing an appropriate defect reduction strategy. This research also highlights the need to improve TDD as a defect reduction method, and to add structure to the process to make it more measurable and predictable.

The second part of the dissertation demonstrates the feasibility of creating a safe change impact analysis algorithm for Java that can operate from arbitrary starting points in code. This was a useful first step in applying mutation analysis to iterative, test-first development methods such as TDD. The research demonstrates that with some improvements to the precision of the algorithm, it can be used to improve the efficiency of mutation analysis in iterative, test-first environments without sacrificing accuracy. The algorithm will also add a missing aspect to software change impact analysis, and will likely have many uses unrelated to mutation analysis or TDD—such as the development of software libraries and frameworks.

Determining the costs and benefits of using change impact analysis to improve mutation analysis, and demonstrating its feasibility is an important first step in an ongoing exploration of how to improve TDD with mutation analysis. This research
justifies additional work on the topic of improving TDD with mutation analysis and improving mutation analysis with change impact analysis. This additional work should not only improve the defect reduction benefits of TDD, but also the measurability of the process—thereby facilitating further research into improving the method.

Another contribution of this research is the proposal of an enhanced TDD method. Although not specifically evaluated in the dissertation, the enhanced method is an outgrowth of the research and can be used as a focal point for additional research on how to add structure and tool support to the process to improve its measurability and predictability.

7.2 Limitations

Limitations have been identified for both the comparison study and the algorithm development and evaluation parts of the dissertation that affect the ability to generalize these findings.

7.2.1 Part 1 Limitations

The main limitations of the comparison study are a result of the use of inexperienced students as experiment participants and as software inspectors. This limits the ability to generalize findings from this dissertation to the software development community in general. Low statistical power also limits the ability to draw conclusions from the finding of non-significance for hypothesis H2 (that the defect reduction effect of either method is reduced by the other method).
Some requirements of the study that would not normally occur in practice, such as the need for inspectors to perform multiple inspections of similar code in a short period of time, and the need to exclude authors from the inspections, also limit the ability to generalize these findings outside of the laboratory environment. These limitations of external validity are further discussed in sections 3.1.5.

7.2.2 Part 2 Limitations

The main limitation of the CHA2AS algorithm is that it does not support all features of the Java language. Support for inner-classes, abstract classes, static fields of Interfaces, and both static and instance initializers were excluded to simplify construction and evaluation of the prototype algorithm. Another simplifying assumption made in the algorithm is that all fields are private. The impact of code executed within threads is not properly handled by CHA2AS, although adding this capability would not be difficult (requiring only the addition of implicit method invocations from Thread.start() to Thread.run()). None of the excluded features actually exist in the Astels data used for testing the algorithm, so although the unsupported language features represent limitations in the capability of the current implementation of the CHA2AS algorithm, they did not impact the evaluation results.

The CHA2AS algorithm does not detect callback method invocations occurring from classes outside of the set of classes in the edited project. This limitation is common to most if not all existing change impact analysis algorithms. The situation occurs when a reference to a class in the edited project is passed to an instance of a class from a library (.jar file in Java programs) that is accessible to the project, and this library instance later
uses the reference to invoke a method on the passed-in object. Because the algorithm
does not parse the source code of library classes (and cannot even be assumed to have
access to the source code) these types of callback invocations are not detectible without
compile-time instrumentation and runtime analysis of the code.

The CHA-AS algorithm does not detect change impacts resulting from an impacted
method that writes data to external storage when that data is later read in by another
method that changes its computation as a result of the externally stored data. As with the
previous limitation, this limitation exists in all current change impact analysis algorithms.
However, unit tests would not normally be written to expect certain values from external
storage because these values are almost always assumed to be changeable outside of the
system. Therefore, this limitation is unlikely to affect the accuracy (precision and recall)
of CHA-AS-enabled mutation analysis for the large majority of software.

The algorithm analysis performed to determine CHA-AS' time and space complexity
does not include a worst-case analysis, so the finding that the algorithm's time and space
complexity is O(n) is not guaranteed to hold for other data sets and other types of
software development projects.

The results from the simulation experiment are susceptible to a mono-operation bias
because the experiment was only performed on a single iterative software development
project. Replicas of the experiment on other TDD developed laboratory and field data
should be performed to assess this potential bias.
7.3 Future Research

Future research opportunities exist both for expanding the scope and generalizability of the comparison study and for continuing and expanding the work on change impact analysis, mutation analysis, and TDD. Many of these future research opportunities involve addressing the limitations reported in section 7.2.

7.3.1 Comparisons of Defect Reduction Methods

Additional comparison studies, both to replicate and expand the comparison study reported here would be a useful extension of this work. For example, a defect comparison study that also includes Pair-Programming would likely result in important new insights into the defect reduction capabilities of Agile methods. Studies with a larger sample size to address low statistical power issues, and field studies to increase generalizability are also likely areas for important research contributions.

7.3.2 Test-Driven Development

The comparison study highlighted the need to add structure to TDD to increase measurability and predictability. The current process as defined by Beck has important benefits but is so loosely defined that it is difficult if not impossible to replicate studies comparing TDD with other techniques. Studies exploring additional ways to add the necessary structure without converting what is intended to be an Agile process into a heavy-weight process may result in significant research contributions.
7.3.3 Mutation Analysis

Mutation analysis is a promising defect reduction method, but after 30 years of academic research, it still is almost non-existent in industrial software development practice. The application of additional techniques to improve its performance and to evaluate its usefulness outside of a laboratory environment would be useful to software development practitioners.

A study comparing the defect reduction benefits of a research-based mutation analysis tool like MuJava to a simpler approach such as the text replacement mechanism of Jester may also produce useful results.

7.3.4 Change Impact Analysis

The research on arbitrary start, change impact analysis algorithms is just beginning. Opportunities exist to create faster, more precise algorithms to solve the same or similar problems. Future research can also examine other existing algorithms to see if they can be modified to address the arbitrary start problem.

7.4 Concluding Remarks

Software defects, budget and schedule overruns, and the seeming inability of managers and programmers to accurately estimate budgets and schedules are some of the foremost issues facing the software development industry. These issues represent almost limitless opportunities for academic research that will improve the industry and positively impact society. This dissertation has addressed some of these issues in a small but hopefully significant way.
APPENDIX A:

FIXED-SIZE GROUPING GENETIC ALGORITHM

A.1 Abstract

The fixed-size partitioning problem is a special case of PARTITION in which the number of partitions and the number of items to be assigned to each partition is given as input to the problem. Because this problem is NP-Hard, enumerative methods for solving it with more than about 18 items to be assigned to groups is infeasible.

Various probabilistic and stochastic methods have been used to solve partitioning problems. The first attempts to solve partitioning problems with genetic algorithms were made in 1991 using Holland’s original encoding of the problem. Although these algorithms produced impressive results, they suffered from problems related to the encoding not matching the nature of the problem. Emanuel Falkenauer proposed a new encoding to solve these problems when using genetic algorithms on partitioning problems (which he calls “grouping problems”). Falkenauer’s encoding is very effective on partitioning problems, and genetic algorithms that use his encoding are commonly referred to as grouping genetic algorithms.

This appendix presents a grouping genetic algorithm for solving fixed-size partitioning problems. One instance of the fixed-size partitioning problem involves the assignment of study participants to research (treatment) groups based on pretest scores. Results are presented on two experiments using the fixed-size grouping genetic algorithm
(FGA) to assign students in an undergraduate Calculus course to hypothetical research groups of various sizes. The results show that this grouping genetic algorithm is very fast and effective at solving the fixed-size partitioning problem.

A.2 Introduction

Fixed-sized partitioning problems are a subset of the PARTITION problem originally identified by R. M. Karp (1972). The general partition problem involves dividing a set into two subsets in which the subsets are equal (or as close to equal as possible) according to some objective function. Garey and Johnson formally define this problem as follows:

**Instance:** A finite set \( A \) and a “size” \( s(a) \in Z^+ \) for each \( a \in A \)

**Question:** Is there a subset \( A' \subseteq A \) such that

\[
\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a) ?
\]

The partition problem has been proven to be NP-Hard (Garey & Johnson, 1979; Karp, 1972).

A.2.1 Fixed-Size Partitioning Problems

The fixed-size partitioning problem (or fixed-size grouping problem) is a special case of PARTITION in which the number of partitions and the number of items to be assigned to each partition is given as input to the problem. This problem is often encountered by researchers who need to divide study participants into equivalent research groups based on pretest scores. Because this problem is a special case of partition, it is NP-Hard.
Therefore, enumerative algorithms that are guaranteed to produce an optimal solution are only feasible when the total number to be assigned to groups is very small (around 18 or fewer).

A.2.2 Genetic Algorithms

Various algorithms have been designed to solve NP-Hard problems for which an enumeration of all possible solutions would take too long (in many cases thousands or more years even for the fastest computers) to be feasible. One of the most effective of these algorithms for many difficult problems is the genetic algorithm. John H. Holland first introduced the genetic algorithm in 1975 (Holland, 1975) as a probabilistic algorithm that can identify near optimal solutions by taking advantage of the principles of “natural selection” and “survival of the fittest” observed in nature, and first reported by Charles Darwin (1859).

Although other representations are possible, most genetic algorithms represent potential solutions to a problem with a binary string (called a chromosome or an individual). Each position in the string is called a gene, and the value of the gene (usually a 0 or a 1) is called an allele. The alleles associated with each gene of a chromosome represent one possible solution to the problem the genetic algorithm is intended to solve. Individual genes are identified by their location (loci) in the chromosome. Genetic algorithms operate by constructing an initial population of chromosomes (potential solutions), and then attempting to improve the initial population in subsequent iterations (or generations) until the population members converge on a solution to the problem.
During each iteration, each member of the current population is tested for fitness as a solution to the problem in question by invoking an objective function, and members are probabilistically selected for survival based on the value of the objective function. Survivors are selected at random, but chromosomes with higher fitness values are given a correspondingly higher probability of selection. The surviving chromosomes then undergo a series of operations to modify them in specific ways that are intended to produce more fit individuals in subsequent generations and to explore promising areas of the problem’s search space. These modified individuals then become the population evaluated in the next iteration.

The two main operations performed on surviving chromosomes are crossover and mutation. Crossover is performed by randomly picking survivors to participate in crossover according to a crossover probability specified as input to the algorithm. The chromosomes selected for crossover are randomly paired and alleles are exchanged between them to produce offspring. The offspring produced by crossover replace the parents in the next generation.

Mutation involves making random changes to a small number of genes within some members of the population. Mutation is performed after crossover, so survivors from the prior generation and offspring of the crossover operation may undergo small random mutations before being propagated to the next generation.

This process continues until some criteria is satisfied—referred to as ‘convergence’. Various convergence criteria can be used, but convergence is often defined as either a maximum number of generations, or a number of generations in which the average fitness
of individuals within the population does not improve. Often these two criteria are combined, in which case the maximum number of generations becomes an upper bound on the number of iterations performed by the algorithm. After convergence, the algorithm terminates and reports the results based on the objective value of the best chromosome created by the algorithm. Several variations of crossover, mutation, and other operations have been published since Holland’s introduction of the genetic algorithm, some of which are described below in section A.3.

Holland’s schemata theorem (Holland, 1975) is the main principle upon which genetic algorithms operate. A schema is any subset of alleles in a chromosome. The schemata theorem states that good solutions to a problem consist of good schemata. The crossover operator works by combining different schemata from relatively fit chromosomes. Because these ‘fit’ chromosomes contain good schemata, the combination of selection based on relative fitness, and crossover, results in genetic algorithms performing exponential sampling of different combinations of good schemata.

The mutation operator prevents premature convergence by forcing the algorithm to search areas of the search space not represented (and often not representable) by schemata in the initial population. The mutation operator is important in preventing convergence before a sufficient amount of the search space has been searched, resulting in convergence to local optimum values, but it can also disrupt good schemata that are necessary to promote convergence to optimal or near optimal values. Therefore, tuning genetic algorithms to the specific problem being solved is necessary to find the most effective combination of crossover and mutation probability and other factors.
The genetic algorithm presented here very effectively handles fixed-size grouping (partitioning) problems of various sizes by finding solutions that are near optimal, and in some cases are optimal, as described in section A.5.

A.3 Literature Review

Some of the first work using genetic algorithms to solve partitioning problems was performed in the early 1990’s (Jones & Beltramo, 1991; Talbi & Bessiere, 1991). These algorithms used Holland’s original encoding of the problem, by representing solutions as positional strings, with alleles assigned to those positions (loci) representing the group to which the corresponding object in the solution space was assigned. Although these papers presented impressive results, the algorithms suffered from problems as a result of the encoding. First, the encoding effectively expands the size of the search space, increasing the number of potential solutions the algorithm has to consider. For example, consider the following chromosome representing the assignment of six items to four different groups (where items are represented by loci and groups are represented by the numbers 1 – 4):

123144

This chromosome indicates that the first and fourth items are grouped together, the fifth and sixth items are grouped together, and the second and third items are each assigned to their own group. The only purpose of the group numbers is to indicate which items are grouped together, and which are assigned to their own groups. Therefore, the following chromosome represents the same solution:
The same solution can be represented by four different chromosomes in this example, effectively quadrupling the size of the search space.

The second problem with the standard encoding when applied to partitioning problems is that it promotes schema disruption. Schemata in partitioning problems are represented by subsets of items that are assigned to the same partition. Individual assignments are not meaningful by themselves (it is the grouping of items together that is meaningful). However, the standard crossover and mutation operators when applied to chromosomes encoded in this way is much more likely to break schemata apart than to keep them together (especially for large chromosomes where items in the same schema may not be physically close together), which has the effect of converting the crossover and mutation operators into nothing more than parallel random search operators.

In 1991, Emanuel Falkenauer proposed a new encoding for genetic algorithms that are intended to solve partitioning problems (Falkenauer, 1991). He refers to partitioning problems as grouping problems\textsuperscript{11}, and he refers to genetic algorithms that use his encoding as grouping genetic algorithms. Falkenauer’s encoding consists of two parts. The first part represents the assignment of items to groups in exactly the same way as Holland’s original encoding. However, this part is followed by a separator and then a second part of the chromosome that contains the label of each group represented in the first part of the chromosome. The following is an example of a chromosome using Falkenauer’s encoding:

\textsuperscript{11} Because of Falkenauer’s use of the word ‘grouping’ instead of ‘partitioning’, these words will be used interchangeably.
Crossover operators in grouping genetic algorithms operate only on the second part of the chromosome (the part after the colon), allowing them to combine entire groups (schemata) from multiple parents into new children, keeping the relationship of items assigned to the same group intact. This prevents the schema disruption caused by the standard encoding.

Falkenauer has published several papers that demonstrate the usefulness of the grouping genetic algorithm for solving various grouping problems. Much of this work is summarized in (Falkenauer, 1998). He compares the grouping genetic algorithm to the algorithm used by Jones and Beltramo for solving the Equal Piles problem in (Falkenauer, 1995). The grouping genetic algorithm consistently finds the same or better solutions as those found by the Jones and Beltramo algorithm, and it finds them in fewer generations (more than an order of magnitude fewer when a greedy crossover operator is used that would not be possible with the standard encoding).

Falkenauer’s encoding solves the schema disruption problem mentioned above (where the relative grouping of items is lost through crossover), but is still subject to a more coarse grained type of schema disruption. A set of two or more groupings can also be considered a schema, and the existence of these coarse grained schemata may significantly affect the objective value of a chromosome. Falkenauer’s crossover operator is biased in favor of coarse grained schemata where the groups to be considered together appear close together in the second part of the chromosome (even though the loci of the groups in the second part of the chromosome has no meaning in the search
space of the problem). To minimize the disruption of these course grained schemata, Falkenauer recommends the use of the (seldom used) inversion operator (Falkenauer, 1991). The inversion operator randomly changes the loci of the groups in the second part of the chromosome. This has no effect on the solution represented by the chromosome, but it allows the crossover operator to sample course grained schemata that would otherwise be under-sampled.

Various modifications have been proposed to the crossover operator to make it more selective in the schemata it samples, and to encourage it to reach convergence more quickly. Algorithms that use these modifications are usually referred to as “greedy” genetic algorithms. Ahuja, Orlin, and Tiwari (2000) consider two such modifications. The first is a greedy crossover operator that produces one child and propagates the best two individuals out of the three (the parents and the child) to the next generation. They found that for the quadratic assignment problem, this operator leads to premature convergence. However, they did achieve good results with a greedy crossover operator that replaces the parent that is most like the child if the child is better than both parents, and replaces the worst parent if the child is not better than both parents. This allows sufficient diversity to remain in the population to avoid premature convergence, while ensuring that the best of the three individuals is never replaced.

Some researchers (Ahuja, Orlin, & Tiwari, 2000; L. Davis, 1991) have used immigration in place of, or in addition to, mutation to introduce diversity into the population. Immigration involves the periodic replacement of population members with new members. Ahuja et al suggest periodic replacement of the least fit individuals with
immigrants. A probabilistic method of choosing individuals to be replaced (similar to that used to select survivors) can also be used.

A.4 Fixed-Size Grouping Genetic Algorithm

The genetic algorithm presented here for solving fixed-size partitioning problems uses Falkenauer’s two-part encoding and corresponding crossover operator, so it is referred to as the Fixed-Size Grouping Genetic Algorithm (or FGA). After creating an initial population, the FGA performs evaluation, selection, inversion, crossover, mutation and immigration (in that order) through multiple iterations until the convergence criteria is satisfied. Each of the FGA’s operators, along with optional features that affect the operation of one or more operators, is described below.

A.4.1 Initialization

The initial chromosomes in the population are created by randomly assigning the genes (based on their loci) to groups, ensuring that for each group, the number of genes assigned matches the size constraint for that group.

A.4.2 Evaluation

Although the FGA is intended to be useful for a variety of fixed-size partitioning problems, it has been developed specifically for the purpose of assigning study participants to research groups with a fixed size. This fixed size need not be the same for all groups. The intent is to equalize the groups based on pretest scores.
Two measures of group equality are combined in the objective function to allow equalization of groups based on both mean scores and score distribution. The similarity of group means is determined by first calculating the mean of each group, and then calculating the standard deviation of the resulting group means. The similarity of the score distributions is determined by first calculating the standard deviation of each group and then calculating the standard deviation of the resulting group standard deviations.

These two values are then added together (after multiplying the second value by two), and the resulting value is returned as the objective value. The second value (standard deviation of group standard deviations) is multiplied by two because experiments showed that without this modification the first value dominated the second, causing the algorithm to produce groups with almost perfectly matched averages but dissimilar distributions.

A.4.3 Selection

Selection of survivors is performed using the “roulette wheel” selection method originally proposed by Holland. The roulette wheel approach constructs a roulette wheel during each iteration of the algorithm, with the same number of slots as the number of individuals in the population. Slot sizes are proportional to the percentage of the total population fitness contributed by the individual corresponding to each slot. The wheel is spun once for each individual in the population by generating a random number, and the member corresponding to the slot containing that random number is selected as a survivor. This method favors the more fit individuals, and often results in multiple copies of the more fit individuals being propagated to the next generation. The literature refers to this as exponential sampling.
Because the objective is to minimize the differences between groups, the objective values returned by the objective function need to be modified before creating the roulette wheel so the more fit individuals (those with lower objective function values) correspond to larger roulette wheel slots. This is done by subtracting an individual’s objective value from the maximum objective value of all members of the population, and then adding to the result the minimum objective value of all members of the population. This converts the minimization problem to a maximization problem (as expected by the roulette wheel selection method) while preserving the relative fitness between each individual in the population.

A.4.4 Inversion

Individuals are selected for inversion based on an inversion probability specified as input to the algorithm. A random number between 0.0 and 1.0 is generated for each survivor, and any survivor whose random number is less than the inversion probability is selected for inversion.

Inversion is performed by randomly selecting two groups from the group part of the chromosome and swapping their positions. This has no effect on the solution represented by the chromosome, but it allows the crossover operator to sample different combinations of groupings.

A.4.5 Crossover

The crossover operator randomly selects survivors for crossover based on a crossover probability in the same way the inversion operator selects survivors for inversion. If an
odd number of survivors is selected for crossover, a random decision is made on whether
to remove a member from the crossover set, or add another member. The selection of the
individual to be either added or removed is also done randomly.

Each crossover results in two children that replace their parents in the survivor list.
Crossover is performed by randomly pairing the individuals in the crossover set and
performing a two-point crossover of the second part (the group list part) of the encoding
to determine which groups to exchange to produce the offspring. Groups are exchanged
to produce offspring as follows:

For each pair to be crossed, denote one member of the pair as X and the other
member as Y. Offspring are produced from this pair in the following steps:
1. Randomly select two crossover points in the group list part of both X and Y.
2. Create a copy Z of X.
3. Unassign any genes in Z that are currently assigned to the groups between the
crossover points of Y.
4. Inject the group(s) between Y’s crossover points into Z by changing group
assignments as necessary so all genes assigned to the groups injected from Y
are now assigned to the same groups in Z.
5. Fill any gaps created by step 3 with the values changed when injecting the
new groups in step 4.
6. Repeat steps 1 – 5 (with the roles of X and Y reversed) to produce a second
offspring.
Individuals not selected for crossover are automatically propagated to the next generation. This is referred to in the literature as reproduction.

A.4.6 Mutation

The mutation operator randomly selects two genes of a chromosome and swaps their group assignments. If the two genes happen to be assigned to the same group, this operation has no effect.

Individuals are randomly selected for mutation based on a mutation probability specified as input to the algorithm. The number of mutations to be performed on each selected individual is also specified as input. Different mutation probabilities and numbers of mutations can be specified for different tiers of survivors. For example, no mutation can be specified for the top X percent of individuals (based on fitness), 25% probability of 1 mutation can be specified for the next Y percent, and 50% probability of 2 mutations can be specified for the remaining individuals.

A.4.7 Immigration

The immigration operator randomly selects survivors to be replaced by randomly generated new chromosomes (immigrants). The immigration probability is specified as input to the algorithm and, as with mutation, different immigration probabilities can be specified for different tiers of survivors.
A.4.8 Greediness Features

The FGA provides three optional greediness features that affect the way the operators perform. The ‘Top-K’ feature can be used in conjunction with the other two, while the ‘Best Two’ and ‘Best Parent Child’ features are mutually exclusive.

A.4.8.1 Top-K

Top-K ensures that the best \( k \)% of individuals are automatically selected as survivors by the selection operator. This automatic selection occurs before the probabilistic selection using the roulette wheel, so when \( k \) is greater than 0, probabilistic selection is used to select the remaining \( POPULATION\ \SIZE - k\% \) survivors. The roulette wheel is constructed as if the top \( k\% \) of individuals did not exist.

The crossover operator performs the same way with Top-K as it does without, except after all children are created and have replaced their parents in the survivor list, the survivor list is checked to ensure that it contains each of the Top-K individuals from the previous generation. Any of these individuals not appearing in the survivor list (because they were selected for crossover and were replaced by their children) are added back to the survivor list by replacing the worst members of the survivor list.

The mutation and immigration operators may still mutate and/or replace individuals from the top \( k\% \) unless a mutation and/or immigration tier is created with a size of \( k\% \) and a probability of 0.
A.4.8.2 Best Two

The ‘Best Two’ feature only affects the crossover operator. When ‘Best Two’ is used, the crossover operator evaluates the fitness of both parents and both children involved in each crossover and selects the best two for inclusion in the survivor list.

A.4.8.3 Best Parent Child

Like the ‘Best Two’ feature, ‘Best Parent Child’ only affects the crossover operator. When ‘Best Parent Child’ is used, the crossover operator evaluates the fitness of both parents and both children involved in each crossover and selects the best parent and the best child for inclusion in the survivor list.

A.4.9 Convergence

Two input parameters are used to determine convergence: a ‘maximum generations’ value, and a ‘stable generations’ value. The FGA automatically terminates after it creates and evaluates the number of generations indicated by ‘maximum generations’. However, it terminates early if the average population fitness does not improve for the number of generations specified by the ‘stable generations’ parameter.

A.5 Results

Two experiments were performed to evaluate the effectiveness of the FGA in creating equal fixed-sized groups. Before each experiment, the algorithm was executed multiple times with the test data in an attempt to optimize the population size, crossover, inversion, mutation, and immigration probabilities, and the greedy features (including
percentage for Top-K) for each experiment. The convergence criteria for each experiment was set to 100,000 maximum generations with a stable generations value of 1,000. For each experiment, the algorithm was executed 10 times with the optimal identified settings. The average of the results for each of the 10 runs is reported for each experiment.

A.5.1 Experiment 1

Experiment 1 involved the grouping of 180 test scores into four unequally sized groups. Two of the groups were set to contain 60 participants each, and the other two groups were set to contain 30 participants each. Because of the large number of scores used in this experiment, it was impossible to enumerate all potential solutions and compare the FGA’s result to a known optimal value. Therefore, a heuristic (described below) was used for comparison.

The FGA settings used for this experiment were: population size 30, crossover probability .85, inversion probability .75, Top-K .04, mutation .00 for the top 7% of the population and .25 for the remaining members of the population, no immigration, and ‘Best Parent Child’ turned on.

The test scores used in this experiment were taken from the first two exams of an undergraduate Calculus course. The original dataset consisted of 182 scores, so two scores were randomly removed from the set.
A.5.1.1 Comparison Heuristic

The comparison heuristic consists of two phases. In the first phase, test scores are sorted in descending order and assigned to groups in order from highest to lowest score. The heuristic switches the direction in which it iterates the groups each time it reaches the last group (so the group that gets the lowest score on one round of assignments gets the highest score on the next round).

Groups are combined in the second phase—with the group containing the highest average score being combined with the group containing the lowest average score—until each group has the correct number of scores. To make this combination possible, without violating the specified group sizes, the heuristic only operates on groups where each group is specified to contain the same number or some even multiple of the number of scores to be assigned to the smallest group. This ensures that the groups can be combined after phase 1 in a way that results in the correct number of scores being assigned to each group.

A.5.1.2 Results

The FGA performed significantly better than the heuristic, producing on average a grouping with an objective value less than 10% of the objective value of the grouping produced by the heuristic\(^\text{12}\). The objective values and the standard deviations from which the objective value is calculated are summarized in table A.1. The values reported for ‘FGA (mean)’ represent a 10 run average.

\(^{12}\) Note that this is a minimization problem, so low objective values are better than high values.
<table>
<thead>
<tr>
<th></th>
<th>Objective Value</th>
<th>STDev of Group Means</th>
<th>STDev of Group STDevs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heuristic</td>
<td>0.2166</td>
<td>0.0204</td>
<td>0.0981</td>
</tr>
<tr>
<td>FGA (best)</td>
<td>0.0113</td>
<td>0.0083</td>
<td>0.0015</td>
</tr>
<tr>
<td>FGA (mean)</td>
<td>0.0187</td>
<td>0.0100</td>
<td>0.0043</td>
</tr>
<tr>
<td>FGA (worst)</td>
<td>0.0280</td>
<td>0.0138</td>
<td>0.0071</td>
</tr>
</tbody>
</table>

Even the result of the worst of the 10 runs is almost an order of magnitude better than the heuristic. Table A.2 shows the means and standard deviations by group for the groupings produced by the heuristic and the best and worst groupings produced by the FGA during the 10 runs of the experiment.

<table>
<thead>
<tr>
<th></th>
<th>Means By Group</th>
<th>Standard Deviations By Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heuristic</td>
<td>58.55, 58.5167, 58.5, 58.5</td>
<td>18.0863, 18.0208, 17.8321, 18.0495</td>
</tr>
<tr>
<td>FGA (best)</td>
<td>58.5167, 58.5167, 58.5333, 58.5333</td>
<td>18.0162, 18.018, 18.0143, 18.0143</td>
</tr>
<tr>
<td>FGA (worst)</td>
<td>58.5333, 58.5167, 58.5, 58.5333</td>
<td>18.0171, 18.0227, 18.0143, 18.0032</td>
</tr>
</tbody>
</table>

The FGA converged in an average of only 1,381 generations, with a worst convergence value of only 2,273 generations. These results show that the FGA is very fast, effective and consistent at producing equivalent unequally sized groups for large datasets.

A.5.2 Experiment 2

Experiment 2 involved the grouping of 18 test scores into three equally sized groups. Because of the small number of test scores used in this experiment, it was feasible to enumerate all possible solutions to identify the optimal solution. Therefore, the FGA results for this experiment are compared against the optimal solution.
The FGA settings used for this experiment were: population size 30, crossover probability .85, inversion probability .75, Top-K .04, mutation .00 for the top 7% of the population and .25 for the remaining members of the population, no immigration, and ‘Best Parent Child’ turned on. The test data for this experiment was a random selection of 18 test scores from the same dataset used for experiment 1.

A.5.2.1 Results

The optimal objective value for this experiment was 4.3329. All 10 runs of the FGA found this value in an average of only 124 iterations. The slowest run took 235 iterations while the fastest took only 22.

A.6 Conclusions, Limitations, and Future Research

The Fixed-Size Grouping Genetic Algorithm is able to efficiently find optimal or near optimal solutions to fixed-size partitioning problems. This research shows the effectiveness of Falkenauer’s encoding scheme for solving partitioning problems with fixed-size partitions. It also presents a very effective method for assigning study participants to equivalent research groups based on pretest scores.

This research was limited by a small number of experiments addressing a very narrow range of problems. Additional research should be performed to expand the scope of experiments and to test the algorithm on problems requiring significantly different objective functions. Future research should also compare these findings with other stochastic search methods such as simulated annealing and tabu search.
APPENDIX B:

JAVA AND OBJECT-ORIENTED DESIGN PRETEST

This appendix contains the pretest used to assess the Java programming ability of participants in the quasi-experimental comparison of software inspection and TDD. Answers are provided in table B.1.

Pretest scores for each participant were provided as input to the genetic algorithm described in Appendix A to assign participants to equalized experiment groups based on Java programming knowledge.

<table>
<thead>
<tr>
<th>Table B.1: Answers to Pretest</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Question</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

* One acceptable answer to question 7

```java
public enum Gender {
    Male, Female;
}
```
Java and Object-Oriented Design Pretest

Name: _____________________________________________________________

Instructions

Most of the questions are multiple-choice. For questions that ask you to write code, write the shortest correct answer you can. Unless the wording of a question allows only one answer, questions may have more than one correct answer. You should circle all correct answers for each question. There are a total of 20 questions and you have 30 minutes to complete the exam (an average of 1 ½ minutes per question).

Grading

Grading will be on a scale of 0 to 3 according to the following grading scale:

- 3 Attempted to answer all questions (nothing left blank) and your name is present
- 2 Attempted to answer all but 1 question and name present
- 1 Attempted to answer all but 2 questions and name present
- 0 Did not answer 3 or more questions or name not present
1) (4 points) What does the following code print?

```java
public class Test
{
    public static void main(String [] args)
    {
        try
        {
            byte b1 = 10;
            byte b2 = 10;
            byte b3 = b1 + b2;
            System.out.print(b3);
        }
        catch(Exception ex)
        {
            System.out.print(" Caught exception ");
        }
        finally
        {
            System.out.print(" Finally!");
        }
    }
}
```

a) 20
b) 20 Finally!
c) Caught exception Finally!
d) Caught exception

e) Nothing. It does not compile.

2) (2 points) Which of the following are correct orderings for the import, class and package declarations when found in a single file?

a) package, import, class
b) class, import, package
c) import, package, class
d) package, class, import
e) None of the above
3) (3 points) In a web-based MVC architecture, where does the Controller typically reside?
   a) In the browser
   b) On the server
   c) In the database
   d) In an XML file
   e) None of the above

4) (2 points) Which of the following are correct formats to use to create the literal char value \textit{a}?
   a) ‘a’
   b) "a"
   c) new Character(a)
   d) \000a
   e) None of the above

5) (3 points) What is the legal range of the byte data type?
   a) –128 through 127
   b) –256 through 255
   c) –32,768 through 32,767
   d) None of the above

6) (5 points) Which of the following are standard (Gang of 4) design patterns?
   a) Singleton
   b) Lightweight
   c) Façade
   d) Decorator
   e) Abstract Bridge
7) (5 points) In the space provided, write the Java code for an enumerated type called Gender that can represent both males and females.

8) (3 points) What is the purpose of an Interface in Java?

   a) To allow user input
   b) To specify the contract of a set of classes
   c) To specify methods that must be implemented by all implementing classes
   d) To allow a partial implementation of a class
   e) None of the above

9) (4 points) Which of the following will compile without warning or error?

   a) float f = 1.3;
   b) char c = "a";
   c) boolean b = null;
   d) int i = 10;
   e) None of the above
10) (3 points) What will be printed if the MyProg class is executed with the indicated command line?

    public class MyProg
    {
        public static void main(String argv[])
        {
            System.out.println(argv[2]);
        }  
    }

    java MyProg good morning

a) MyProg  
b) good  
c) morning  
d) Nothing. ArrayIndexOutOfBoundsException is raised instead.

11) (2 points) When is delegation more appropriate than inheritance?

a) When a "has a" relationship does not exist  
b) When an "is a" relationship does not exist  
c) When a "be a" relationship does not exist  
d) Never. Inheritance is preferred wherever it is allowed.

12) (2 points) Which of the following are legal identifiers in Java?

a) 2variable  
b) variable2  
c) _whatavvariable  
d) _3_  
e) $anothervar  
f) #myvar
13) (2 points) What will happen when the following code is executed?

```java
class MyClass {
    static int i;
    public static void main(String argv[])
    {
        System.out.println(i);
    }
}
```

- a) A compiler error indicating variable i may not have been initialized
- b) An exception indicating variable i may not have been initialized
- c) null will be printed
- d) 1 will be printed
- e) 0 will be printed

14) (5 points) What will be the result of attempting to compile and run the Mine class?

```java
abstract class MineBase {
    static int i;
    abstract void amethod();
}

class Mine extends MineBase {
    public static void main(String argv[]) {
        int[] ar = new int[5];
        for (i = 0; i < ar.length; i++)
            System.out.println(ar[i]);
    }
}
```

- a) A sequence of 5 0's will be printed
- b) Either a compiler error or a runtime exception indicating ar is used before it is initialized
- c) A compiler error indicating that Mine must be declared abstract
- d) An IndexOutOfBoundsException exception
15) (2 points) What will be printed if you execute the following code?

```java
int i = 1;
switch (i)
{
    case 0:
        System.out.print("zero ");
        break;
    case 1:
        System.out.print("one ");
    case 2:
        System.out.print("two ");
    default:
        System.out.print("default ");
}
```

a) one  
b) one default  
c) one two default  
d) default

16) (5 points) Which of the following is the MOST LIKELY indicator of a bad design?

a) Overriden methods  
b) Use of the 'abstract' keyword  
c) Use of the 'instanceof' keyword  
d) Use of Interfaces  
e) None of the above (all are indicators of a good design)

17) (2 points) In the following code, what modifiers would be legal in place of the comment /*Insert modifier here*/?

```java
public class MyClass1
{
    public static void main(String argv[]){ }
    /*Insert modifier here*/ class MyInner { }
}
```

a) public  
b) private  
c) static  
d) void  
e) friend
18) (3 points) Given the following code, what change(s) could you make to invoke the parent constructor that prints the String “base constructor”?

class Base
{
    Base(int i)
    {
        System.out.println("base constructor");
    }

    Base(){ }
}

public class Sup extends Base
{
    public static void main(String argv[])
    {
        Sup s= new Sup();
        //One
    }

    Sup()
    {
        //Two
    }

    public void derived()
    {
        //Three
    }
}

a) Replace the comment //One, with Base(10);
b) Replace the comment //One, with super(10);
c) Replace the comment //Two, with Base(10);
d) Replace the comment //Two, with super(10);
e) Replace the comment //Three, with Base(10);
f) Replace the comment //Three, with super(10);
19) (5 points) What will be printed if you attempt to compile and run the following code?

```java
public class Pass {
    static int j = 20;
    public static void main(String argv[]) {
        int i = 10;
        Pass p = new Pass();
        p.amethod(i);
        System.out.println(i);
        System.out.println(j);
    }

    public void amethod(int x) {
        x = x*2;
        j = j*2;
    }
}
```

a) Nothing. It does not compile because of a parameter mismatch.
b) 20 and 40
c) 10 and 40
d) 10, and 20

20) (2 points) You are concerned that your program may attempt to use more memory than is available. To avoid this situation you want to ensure that the Java Virtual Machine will run its garbage collection just before you start a complex routine. What can you do to ensure that garbage collection will run when you want?

a) You cannot be certain when garbage collection will run
b) Use the Runtime.gc() method to force garbage collection
c) Ensure that all the variables you require to be garbage collected are set to null
d) Use the System.gc() method to force garbage collection
APPENDIX C:

PROGRAMMING ASSIGNMENT

This appendix contains the description of the programming assignment completed by participants in the quasi-experimental comparison of software inspection and TDD. The programming assignment description, along with a set of completion and submission instructions (included in Appendix D) was given to each participant at the start of the experiment.
Programming Assignment
Spam Filter

This assignment involves the development of some of the functionality of an e-mail Spam Filter. The functionality to be implemented involves the reading/parsing of an XML configuration file containing the rules, allowed-list, and blocked-list for the filter, and the application of this information to an object representing an e-mail message to determine if the message should be accepted or rejected. To make the project manageable within a reasonable amount of time, functionality for attaching the spam filter to actual e-mail accounts and the reading of real e-mail messages from a mail server is not part of this project. Also, some of the code has already been written for you and the high-level design work has already been done.

Your task is to implement the remaining functionality as described in the ‘Design’ section below. Much of the grading will be done in an automated fashion, so it is important to follow any design specifications exactly. For example, you must name your classes (including package names) as specified here, and you must include constructors for your classes with the exact signatures requested. Your code should compile and run under J2SE version 5.0 for Windows available from Sun’s website (http://java.sun.com). This is the same version of Java that is installed in the labs and recommended for download from the course web page.

This assignment should be done individually. If you need help or have a question, send e-mail to the course newsgroup (cs.course335 on server news.cs.arizona.edu—use your Lectura username and password to access the newsgroup), or to jwilkers@email.arizona.edu. Questions of general interest not sent to the newsgroup will be posted (along with the response) to the newsgroup. Please carefully review this document, the API documentation for the provided classes and interfaces, and any existing newsgroup postings before submitting questions.

Design

The combination class and package diagram of figure 1 shows the high-level design for the assignment. A few dependency relationships are not shown in the diagram; however, these are probably not relevant for what you need to do and can easily be inferred from the provided code if necessary.

Much of the functionality for this assignment is already provided. The ‘SpamFilter.zip’ file (available on the course website) contains the provided classes and interfaces,
Javadoc API documentation, a sample XML configuration file\textsuperscript{13}, and the DTD file that will be used during grading of your assignment.

You should add the functionality described below for the classes in the diagram without changing the design, without modifying any provided code, and without changing any method signatures. Additional information about the provided classes and interfaces can be found in the provided Javadoc documentation.

\textbf{Interfaces and Completely Implemented Classes}

Interfaces are provided and some of the classes are already completely implemented. They are provided for use by classes you will implement. Automated testing of your code will be performed according to the API specified by these classes and interfaces, so you should not change them. If you do change them, you are responsible to ensure that the API has not changed so all of our pre-written tests will work as expected. The provided classes are the following:

\footnote{The sample configuration file does not illustrate every condition against which your code will be tested during grading. Your spam filter should work properly against any possible configuration file that conforms to the provided DTD.}
**ArgumentParser** – Used by the main method of the *SpamFilter* class to parse command-line parameters. You should not need to use it directly.

**AbstractRuleParser** – Provides some of the low-level functionality required to parse the spam filter configuration file. Also specifies the interface for a subclass (named *RuleContext*) that you will write.

**SpamFilter** – The main class of the project. It accepts command-line parameters specifying the content of an e-mail message and the CLASSPATH relative path name of a configuration file and DTD, and applies the information from the configuration file to determine whether the message should be accepted or rejected. Uses the *RuleContext* and *FilterRuleContext* classes (which you will implement) to accomplish this.

The `parseConfigFile(…)` method uses the *RuleContext* class to parse the configuration file and ensure that the file is valid according to the information given in the ‘Validation’ section of this document.

The `filterMessage(…)` method throws a *FilterViolationException* if the message should be rejected according to the configuration file. If the method does not throw a *FilterViolationException*, the message is deemed to have been accepted. The following are the requirements it uses for determining whether a message should be accepted or rejected:

1. A match between a message’s sender and an entry in the allowed or blocked list results in the acceptance (no exception thrown) or rejection (FilterViolationException thrown) of the message without regard to whether any rules are violated.
2. Items in the allowed and blocked address lists take precedence over items in the allowed and blocked domains list.
3. Messages whose sender is not matched by any entry in either the allowed or blocked lists are rejected if any rule is violated; otherwise, they are accepted.

**Message** – Represents an e-mail message.

**Rule** – Represents a rule from the spam filter configuration file and specifies the interface for an implementing class you will create named *FilterRuleContext*.

**Type** – An enum class implemented in the *RuleContext* interface that specifies the type of a rule (either regular expression or text).

**Conjunction** – An enum class implemented in the *RuleContext* interface that specifies the meaning of multiple rule clauses in a rule. For details, refer to the Javadoc comment for the `getConjunction()` method of the *RuleContext* interface.
**ValidationException** – An exception thrown to indicate that a validation error was found in the spam filter configuration file.

**FilterViolationException** – An exception thrown to indicate that a message should be rejected based on the information in the spam filter configuration file.

---

**Classes to be Implemented by the Student**

The following two classes are to be implemented by the student. Your implementation may contain additional classes and additional methods not included in the provided abstract classes and interfaces; however, all functionality should be accessible through the API specified by the provided abstract classes and interfaces and any constructors you are specifically told to implement.

The provided classes contain extensive Javadoc comments to help you understand the requirements and design of the project. However, you are not required or expected to provide Javadoc comments for your code.

**FilterRule** – This class should be placed in the `edu.arizona.spamfilter.beans` package. You may create any constructors you think are appropriate. However, there must be a no-argument constructor. This class should implement the `Rule` interface (which is provided). The `FilterRule` class represents a rule from the configuration file. The `RuleParser` class (which you will also implement) creates an instance of this class for each rule in the configuration file. Refer to the Javadoc documentation for the `Rule` interface for additional details. For information on what the `validate()` method should do, refer to the ‘Validation’ section below.

**RuleParser** – This class is responsible for parsing the XML configuration file and returning the list of allowed addresses, allowed domains, blocked addresses, blocked domains, and rules when the appropriate method is called. `RuleParser` is a subclass of `AbstractRuleParser` and should implement all of `AbstractRuleParser`’s abstract methods. `RuleParser` should be placed in the same package as `AbstractRuleParser`, and should contain a constructor that takes two String arguments. The first argument is the CLASSPATH relative path name of the XML configuration file to be parsed. The second argument is the CLASSPATH relative path name of the DTD file to be used to validate the configuration file. A copy of the DTD file is provided for you to use for testing. Your constructor need only invoke the parent constructor and pass it the Strings.

---

14 The easiest way to specify the parameters is to ensure that the files are in a directory on the CLASSPATH, and then just include the file names as the parameters (without any directory information).
The most difficult part of this class is the `parse()` method. This method should extract the relevant information from the configuration file, making it available to the accessor (getter) methods specified in `AbstractRuleParser`. The `parse()` method should also throw `ValidationException` for any validation problems in the configuration file that are not caught by the DTD. However, you are not required to write the code that initializes the XML parser (Apache Xerces) and reads the configuration file—creating a DOM tree. This is already implemented in the `getRootElement()` method of `AbstractRuleParser`. Your `parse()` method should invoke the `getRootElement()` method which will create the DOM tree and return the root node of the tree. Your method can then extract the relevant information and perform any necessary validations by traversing the DOM tree from this root node. For this to work, any .jar files in the provided ‘lib’ directory must be available on the CLASSPATH, which will be set up for you in the Eclipse project. Refer to the ‘Validation’ section for information on the validations that should be performed.

**Validation**

Several validations (beyond what the DTD can do) must be performed on the configuration file. You are responsible to determine where each validation should be performed in the code. Any validation errors should result in a `ValidationException` being thrown. The exception message should clearly indicate the error that caused the exception. The validations should be performed within the specified API (you should not add ‘throws ValidationException’ to any method specified by a provided class or interface that does not already specify that it throws the exception), and all validations should be applied by the time the `parse()` method of `RuleParser` returns. The following validation requirements specify the validations that should be performed (not in any particular order):

1. The same e-mail address may not appear in the addresses section of both the allowed and blocked lists.
2. The same domain name may not appear in the domains section of both the allowed and blocked lists.
3. The same e-mail address does not appear twice in the addresses section of the same allowed or blocked list.
4. The same domain name does not appear twice in the domains section of the same allowed or blocked list.
5. Address and domain elements in both the allowed and blocked lists may not be empty.
6. Each entry in the addresses section of both the allowed and blocked lists must be a properly formatted e-mail address.
7. Each entry in the domains section of both the allowed and blocked lists must be a properly formatted domain name.
8. Rule Id’s must be unique integers.
9. Each rule must contain at least one of the following rule clauses: to, cc, bcc, from, subject, body.
APPENDIX D:  
INSTRUCTIONS TO PARTICIPANTS

This appendix contains the instructions for completion and submission of the programming assignment that were given to participants in the quasi-experimental comparison of software inspection and TDD. A separate set of instructions was provided for participants in each of the four experimental groups. Each set of instructions is included here.

Because all participants were recruited from an undergraduate college course, and completion of the assignment was required of all students—regardless of whether they consented to participate in the experiment—the instructions also contain assignment grading information.
Programming Assignment
Assigned Method and Submission Instructions

Name: ________________________________________________________________

You have been assigned to use **standard programming methods** during completion of
the attached programming assignment. You should not use any form of automated
testing or testing tools or frameworks such as JUnit.

**Recording of Time**
You must record all of your time spent on the project in the ‘Timesheet.xls’ spreadsheet
file which is available on the course web site. Instructions for completing the timesheet
can be found at the top of the timesheet file. You will be graded on the completeness and
accuracy of your timesheet but not on the amount of time spent.

**Assignment Submission**
Use the ‘turnin’ program to submit your work using a project named ‘SpamFilter’ with a
turnin folder and zip file name of
3_1_SectionLeaderName_YourFirstName_YourLastName. The first iteration is due
September 18th at 10:00 pm. On or before this date, you must submit your almost
complete ‘RuleParser’ class. All methods should work except `getRules()`, which
only need return a List of RuleFilter objects constructed with default values and the
default constructor (none of the getters need represent the true state of the Rule). You
must also submit the completed Timesheet.xls file showing all time spent on the
assignment up to this date. Of the validations listed on the last page of the assignment,
the following are required for iteration one:

1. The same e-mail address may not appear in the addresses section of both the
   allowed and blocked lists.
2. The same domain name may not appear in the domains section of both the
   allowed and blocked lists.

The second iteration is due September 25th at 10:00 pm and should include all
functionality. Submit a zip file named
3_2_SectionLeaderName_YourFirstName_YourLastName. You must submit your
Eclipse project with all source code files (including those previously submitted in
iteration one\(^\text{15}\), and all provided source files—whether you changed them or not), and
your timesheet (including all time from both the first and second iterations).

\(^{15}\) You may make changes to classes submitted in iteration one before resubmitting them in iteration two.
Grading Criteria, Iteration 1, 50 points (Subject to change):

+ 2 You turned in an Eclipse project with required files representing a valid attempt that compiles.
+ 4 Time sheet has been filled out (please record your time accurately).
+ 6 You did not utilize any unit testing (no unit tests or assertions are present).
+30 All methods in ‘RuleParser’ (except for getRules) work correctly.
The 30 points will be curved so 30 minus the highest score in this group will be added to everyone's score in this group. If the second highest score differs substantially from the highest score, then 30 minus the second highest score will be added to everyone in this group, which means the highest score would get a bonus.
+8 Style, Design, and Unanticipated Error (see bullets below).

---

Grading Criteria, Iteration 2, 50 points (Subject to change):

+ 2 You turned in an Eclipse project with required files representing a valid attempt that compiles.
+ 4 Time sheet has been filled out (please record your time accurately).
+ 6 You did not utilize any unit testing (no unit tests or assertions are present).
+30 The getRules method of RuleParser and all methods of FilterRule work correctly.
The 30 points will be curved so 30 minus the highest score in this group will be added to everyone's score in this group. If the second highest score differs substantially from the highest score, then 30 minus the second highest score will be added to everyone in this group, which means the highest score would get a bonus.
+8 Style, Design, and Unanticipated Error.

- Each class has your name and a description of its single responsibility
- Efficient (no inefficient code that could have been done with less runtime cost)
- Maintainable (readable code)
- You used intention-revealing names (meaningful identifiers)
- Used consistent spacing/indentation (format all code with Source > Format).
- Class names begin with an UpperCase letter
- All method and variable names begin with a lowerCase letter
- Placed one blank line between methods
- Used local variables when appropriate and instance variables when appropriate
- There are no unanticipated errors or breeches of good style or design
You have been assigned to use **formal software inspection** during completion of the attached programming assignment. You should not use any form of automated testing or testing tools or frameworks such as JUnit. After you complete and submit your assignment, your code will undergo a process of formal software inspection. You may be asked to participate in the inspection process. Approximately two-weeks after the assignment submission deadline you will receive a report of issues in your code that need to be resolved. You will be given additional time to resolve these issues and will then be required to resubmit your work with the issues resolved. You will be given resubmission instructions with your report of issues to be resolved.

**Recording of Time**
You must record all of your time spent on the project in the ‘Timesheet.xls’ spreadsheet file which is available on the course web site. Instructions for completing the timesheet can be found at the top of the timesheet file. You will be graded on the completeness and accuracy of your timesheet but not on the amount of time spent.

**Assignment Submission**
Use the ‘turnin’ program to submit your work using a project named ‘SpamFilter’ with a turnin folder and zip file name of 3_1_SectionLeaderName_YourFirstName_YourLastName. The first iteration is due September 18th at 10:00 pm. On or before this date, you must submit your almost complete ‘RuleParser’ class. All methods should work except `getRules()`, which only need return a List of RuleFilter objects constructed with default values and the default constructor (none of the getters need represent the true state of the Rule). You must also submit the corresponding JUnit test class(es), and the completed Timesheet.xls file showing all time spent on the assignment up to this date. Of the validations listed on the last page of the assignment, the following are required for iteration one:

3. The same e-mail address may not appear in the addresses section of both the allowed and blocked lists.
4. The same domain name may not appear in the domains section of both the allowed and blocked lists.

The second iteration is due September 25th at 10:00 pm and should include all functionality. Submit a zip file named 3_2_SectionLeaderName_YourFirstName_YourLastName. You must submit your
Eclipse project with all source code files (including those previously submitted in iteration one\textsuperscript{16}, and all provided source files—whether you changed them or not), your JUnit test classes, and your timesheet (including all time from both the first and second iterations).

**Grading Criteria, Iteration 1, 50 points (Subject to change):**

+ 2 You turned in an Eclipse project with required files representing a valid attempt that compiles.
+ 4 Time sheet has been filled out (please record your time accurately).
+ 6 You did not utilize any unit testing (no unit tests or assertions are present).
+ 30 All methods in ‘RuleParser’ (except for getRules) work correctly.
  
  The 30 points will be curved so 30 minus the highest score in this group will be added to everyone's score in this group. If the second highest score differs substantially from the highest score, then 30 minus the second highest score will be added to everyone in this group, which means the highest score would get a bonus.
+ 8 Style, Design, and Unanticipated Error (see bullets below).

**Grading Criteria, Iteration 2, 50 points (Subject to change):**

+ 2 You turned in an Eclipse project with required files representing a valid attempt that compiles.
+ 4 Time sheet has been filled out (please record your time accurately).
+ 6 You did not utilize any unit testing (no unit tests or assertions are present).
+ 30 The getRules method of RuleParser and all methods of FilterRule work correctly.
  
  The 30 points will be curved so 30 minus the highest score in this group will be added to everyone's score in this group. If the second highest score differs substantially from the highest score, then 30 minus the second highest score will be added to everyone in this group, which means the highest score would get a bonus.
+ 8 Style, Design, and Unanticipated Error.
  
  * Each class has your name and a description of its single responsibility
  * Efficient (no inefficient code that could have been done with less runtime cost)
  * Maintainable (readable code)
  * You used intention-revealing names (meaningful identifiers)
  * Used consistent spacing/indentation (format all code with Source > Format).
  * Class names begin with an UpperCase letter
  * All method and variable names begin with a lowerCase letter
  * Placed one blank line between methods

\textsuperscript{16}You may make changes to classes submitted in iteration one before resubmitting them in iteration two.
• Used local variables when appropriate and instance variables when appropriate
• There are no unanticipated errors or breeches of good style or design
Programming Assignment
Assigned Method and Submission Instructions

Name: ________________________________________________________________

You have been assigned to use test-driven development during completion of the attached programming assignment. Please complete the programming assignment while following the instructions under Test-Driven Development below.

Test-Driven Development
You should create JUnit tests for all of the functionality you add to complete the assignment. JUnit tests should be created before the corresponding program logic whenever possible. Please create the JUnit tests as instructed in class and ensure that they test the full functionality of your code. You are not required or expected to create JUnit tests for any functionality that is provided for you. You will be graded on both your completion of the attached assignment and the completeness of your JUnit tests.

Recording of Time
You must record all of your time spent on the project in the ‘Timesheet.xls’ spreadsheet file which is available on the course web site. Instructions for completing the timesheet can be found at the top of the timesheet file. You will be graded on the completeness and accuracy of your timesheet but not on the amount of time spent.

Assignment Submission
Use the ‘turnin’ program to submit your work using a project named ‘SpamFilter’ with a turnin folder and zip file name of 3_1_SectionLeaderName_YourFirstName_YourLastName. The first iteration is due September 18th at 10:00 pm. On or before this date, you must submit your almost complete ‘RuleParser’ class. All methods should work except getRules(), which only need return a List of RuleFilter objects constructed with default values and the default constructor (none of the getters need represent the true state of the Rule). You must also submit the completed Timesheet.xls file showing all time spent on the assignment up to this date. Of the validations listed on the last page of the assignment, the following are required for iteration one:

5. The same e-mail address may not appear in the addresses section of both the allowed and blocked lists.
6. The same domain name may not appear in the domains section of both the allowed and blocked lists.
The second iteration is due September 25th at 10:00 pm and should include all functionality. Submit a zip file named 
3_2_SectionLeaderName_YourFirstName_YourLastName. You must submit your Eclipse project with all source code files (including those previously submitted in iteration one\(^{17}\), and all provided source files—whether you changed them or not), and your timesheet (including all time from both the first and second iterations).

**Grading Criteria, Iteration 1, 50 points (Subject to change):**

+ 2 You turned in an Eclipse project with required files representing a valid attempt that compiles.  
+ 4 Time sheet has been filled out (please record your time accurately).  
+ 6 You have a unit test to test all methods (except for getRules) of ‘RuleParser’.  
+ 30 All methods in ‘RuleParser’ (except for getRules) work correctly.  
  The 30 points will be curved so 30 minus the highest score in this group will be added to everyone's score in this group. If the second highest score differs substantially from the highest score, then 30 minus the second highest score will be added to everyone in this group, which means the highest score would get a bonus.  
+ 8 Style, Design, and Unanticipated Error (see bullets below).

**Grading Criteria, Iteration 2, 50 points (Subject to change):**

+ 2 You turned in an Eclipse project with required files representing a valid attempt that compiles.  
+ 4 Time sheet has been filled out (please record your time accurately).  
+ 6 You have a unit test to test all methods of ‘FilterRule’ and you added tests for the getRules method of ‘RuleParser’ (from iteration 1).  
+ 30 The getRules method of ‘RuleParser’ and all methods of ‘FilterRule’ work correctly.  
  The 30 points will be curved so 30 minus the highest score in this group will be added to everyone's score in this group. If the second highest score differs substantially from the highest score, then 30 minus the second highest score will be added to everyone in this group, which means the highest score would get a bonus.  
+ 8 Style, Design, and Unanticipated Error.  
  • Each class has your name and a description of its single responsibility  
  • Efficient (no inefficient code that could have been done with less runtime cost)  
  • Maintainable (readable code)  
  • You used intention-revealing names (meaningful identifiers)  
  • Used consistent spacing/indentation (format all code with Source > Format).  

\(^{17}\) You may make changes to classes submitted in iteration one before resubmitting them in iteration two.
• Class names begin with an UpperCase letter
• All method and variable names begin with a lowerCase letter
• Placed one blank line between methods
• Used local variables when appropriate and instance variables when appropriate
• There are no unanticipated errors or breeches of good style or design
You have been assigned to use both formal software inspection and test-driven development during completion of the attached programming assignment. After you complete and submit your assignment, your code will undergo a process of formal software inspection. You may be asked to participate in the inspection process. Approximately two weeks after the assignment submission deadline you will receive a report of issues in your code that need to be resolved. You will be given additional time to resolve these issues and will be required to resubmit your work with the issues resolved. You will be given resubmission instructions with your report of issues to be resolved. Please complete the programming assignment while following the instructions under Test-Driven Development below.

### Test-Driven Development

You should create JUnit tests for all of the functionality you add to complete the assignment. JUnit tests should be created before the corresponding program logic whenever possible. Please create the JUnit tests as instructed in class and ensure that they test the full functionality of your code. You are not required or expected to create JUnit tests for any functionality that is provided for you. You will be graded on both your completion of the attached assignment and the completeness of your JUnit tests.

### Recording of Time

You must record all of your time spent on the project in the ‘Timesheet.xls’ spreadsheet file which is available on the course web site. Instructions for completing the timesheet can be found at the top of the timesheet file. You will be graded on the completeness and accuracy of your timesheet but not on the amount of time spent.

### Assignment Submission

Use the ‘turnin’ program to submit your work using a project named ‘SpamFilter’ with a turnin folder and zip file name of 3_1_SectionLeaderName_YourFirstName_YourLastName. The first iteration is due September 18th at 10:00 pm. On or before this date, you must submit your almost complete ‘RuleParser’ class. All methods should work except `getRules()`, which only need return a List of RuleFilter objects constructed with default values and the default constructor (none of the getters need represent the true state of the Rule). You must also submit the corresponding JUnit test class(es), and the completed Timesheet.xls file showing all time spent on the assignment up to this date. Of the validations listed on the last page of the assignment, the following are required for iteration one:
7. The same e-mail address may not appear in the addresses section of both the allowed and blocked lists.
8. The same domain name may not appear in the domains section of both the allowed and blocked lists.

The second iteration is due September 25\textsuperscript{th} at 10:00 pm and should include all functionality. Submit a zip file named 3_2_SectionLeaderName_YourFirstName_YourLastName. You must submit your Eclipse project with all source code files (including those previously submitted in iteration one\textsuperscript{18}, and all provided source files—whether you changed them or not), your JUnit test classes, and your timesheet (including all time from both the first and second iterations).

**Grading Criteria, Iteration 1, 50 points (Subject to change):**

+ 2 You turned in an Eclipse project with required files representing a valid attempt that compiles.
+ 4 Time sheet has been filled out (please record your time accurately).
+ 6 You have a unit test to test all methods (except for getRules) of ‘RuleParser’.
+ 30 All methods in ‘RuleParser’ (except for getRules) work correctly.

The 30 points will be curved so 30 minus the highest score in this group will be added to everyone's score in this group. If the second highest score differs substantially from the highest score, then 30 minus the second highest score will be added to everyone in this group, which means the highest score would get a bonus.

+ 8 Style, Design, and Unanticipated Error (see bullets below).

**Grading Criteria, Iteration 2, 50 points (Subject to change):**

+ 2 You turned in an Eclipse project with required files representing a valid attempt that compiles.
+ 4 Time sheet has been filled out (please record your time accurately).
+ 6 You have a unit test to test all methods of ‘FilterRule’ and you added tests for the getRules method of ‘RuleParser’ (from iteration 1).
+ 30 The getRules method of ‘RuleParser’ and all methods of ‘FilterRule’ work correctly.

The 30 points will be curved so 30 minus the highest score in this group will be added to everyone's score in this group. If the second highest score differs substantially from the highest score, then 30 minus the second highest score will be added to everyone in this group, which means the highest score would get a bonus.

+ 8 Style, Design, and Unanticipated Error (see bullets below).

\textsuperscript{18} You may make changes to classes submitted in iteration one before resubmitting them in iteration two.
highest score will be added to everyone in this group, which means the highest score would get a bonus.

+8 Style, Design, and Unanticipated Error.
- Each class has your name and a description of its single responsibility
- Efficient (no inefficient code that could have been done with less runtime cost)
- Maintainable (readable code)
- You used intention-revealing names (meaningful identifiers)
- Used consistent spacing/indentation (format all code with Source > Format).
- Class names begin with an UpperCase letter
- All method and variable names begin with a lowerCase letter
- Placed one blank line between methods
- Used local variables when appropriate and instance variables when appropriate
- There are no unanticipated errors or breeches of good style or design
APPENDIX E:

CODE PROVIDED TO PARTICIPANTS

This appendix contains listings of source code that was provided to participants in the quasi-experimental comparison of software inspection and TDD. The source code was used by participants as a template for completing the programming assignment. The purpose of providing the template was two-fold. First, it reduced the scope of the assignment, making completion of the assignment practical within the allotted time. Second, it specified an API the participants were required to provide to allow the automated tests created for defect counting to execute on all code submitted by participants.

The code was delivered in a zip file as part of a preconfigured Eclipse project containing source code, JavaDoc API documentation, the Xerces XML parser, and a template for the XML configuration file the participant's code was intended to parse. All pre-written code provided to participants is included here except for two trivial exception classes: ValidationException and FilterViolationException.
E.1 Code Listing for SpamFilter Class

```java
package edu.arizona.spamfilter;

import java.io.IOException;
import java.util.List;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.SAXException;
import edu.arizona.spamfilter.beans.Message;
import edu.arizona.spamfilter.util.ArgumentParser;
import edu.arizona.spamfilter.xml.AbstractRuleParser;

/**
 * Filters a message specified on the command line according to the
 * rules of the specified XML configuration file.
 */
public class SpamFilter {

    /**
     * The parser used to parse the XML configuration file.
     */
    private AbstractRuleParser parser = null;

    /**
     * Parses and validates the specified XML configuration file and
     * uses the information from the file to filter the message
     * specified on the command line. Calls
     * (parser.parseConfigFile(String, String)) to read and parse the
     * configuration file, and (parser.filterMessage(Message)) to
     * filter the message specified on the command line (after
     * creating a Message object from the command line parameters).
     *
     * @param args
     *          the command line arguments (see parser.usage() for details).
     * @throws ParserConfigurationException
     *          if the XML parser (i.e. Xerces) is configured
     *          incorrectly.
     * @throws SAXException
     *          if the configuration file is malformed, doesn't
     *          conform to the DTD, or some other error occurs
     *          while parsing the configuration file.
     * @throws IOException
     *          if an IO error occurs.
     * @throws ValidationException
     *          if the configuration file conforms to the DTD but
     *          is invalid for some reason not detected by the
     *          DTD.
     */
    public static void main(String[] args)
            throws ParserConfigurationException, SAXException,
            IOException, ValidationException {
```

ArgumentParser argParser = new ArgumentParser(args);

// Create a message from the command line parameters
Message message = new Message();
message.setTo(argParser.getArguments("to"));
message.setCc(argParser.getArguments("cc"));
message.setBcc(argParser.getArguments("bcc"));
message.setFrom(argParser.getArgument("from"));
message.setSubject(argParser.getArgument("subject"));
message.setBody(argParser.getArgument("body"));

if (!argParser.containsArgument("configFile")
    || !argParser.containsArgument("dtdFile")) {
    usage();
} else {
    SpamFilter filter = new SpamFilter();
    filter.parseConfigFile(argParser.getArgument("configFile"), argParser.getArgument("dtdFile"));

    try {
        filter.filterMessage(message);
        System.out.println("The message is not spam.");
    } catch (FilterViolationException ex) {
        System.out.println("The message is spam because: " + ex.getMessage());
    }
}

/**
 * Displays the usage string indicating how to invoke the
 * application.
 */
public static void usage() {
    StringBuffer buffer = new StringBuffer();

    buffer.append("Usage: SpamFilter -configFile "
        + "<configFileName>\n\t" + "-dtdFile <dtdFileName> [-to <toAddress>] "
        + "[-cc <ccAddress>]\n\t[-bcc <bccAddress>] "
        + "[-from <fromAddress>]\n\t" + "[-subject <subject text>] "
        + "[-body <body text>]\n\t"");

    buffer.append(System.getProperty("line.separator"));
    buffer.append(System.getProperty("line.separator"));
    buffer.append("\t-configFileName - The classpath relative "
        + "name of the xml\n\t configuration file "
        + "(required).\n\t");
    buffer.append(System.getProperty("line.separator"));
    buffer.append("\t-dtdFileName - The classpath relative "
        + "name of the dtd file\n\t (required).\n\t"};
buffer.append("\t-to - The 'to' address of the message. May occur multiple times.\n" + "message.");
buffer.append(System.getProperty("line.separator"));
buffer.append("\t-cc - The 'cc' address of the message. May occur multiple times.\n" + "message.");
buffer.append(System.getProperty("line.separator"));
buffer.append("\t-bcc - The 'bcc' address of the message. May occur multiple times.\n" + "message.");
buffer.append(System.getProperty("line.separator"));
buffer.append("\t-from - The 'from' address of the message.\n" + "message.");
buffer.append(System.getProperty("line.separator"));
buffer.append("\t-subject - The 'subject' of the message (enclose subject text in quotes).\n" + "message.");
buffer.append(System.getProperty("line.separator"));
buffer.append("\t-body - The 'body' of the message (enclose body text in quotes).\n" + "message.");

System.out.println(buffer.toString());
}

/**
 * Uses a student implemented subclass of
 * {@link edu.arizona.spamfilter.xml.AbstractRuleParser} (named RuleParser) to parse and validate the XML configuration file.
 *
 @param configFileName the CLASSPATH relative path name of the XML configuration file.
 @param dtdFileName the CLASSPATH relative path name of the DTD for the XML configuration file.
 @throws ParserConfigurationException if the XML parser (i.e. Xerces) is configured incorrectly.
 @throws SAXException if the configuration file is malformed, doesn't conform to the DTD, or some other error occurs while parsing the configuration file.
 @throws IOException if an IO error occurs.
 @throws ValidationException if the configuration file conforms to the DTD but is invalid for some reason not detected by the DTD.
 */
public void parseConfigFile(String configFileName, String dtdFileName) throws ParserConfigurationException, SAXException, IOException, ValidationException {
    // TODO To be implemented by the student
}

/**
* @param message
*            the message to be filtered.
* @throws FilterViolationException
*            if the message violates a rule or the sender is in
*            the blocked-list.
*/
public void filterMessage(Message message)
    throws FilterViolationException {
    // TODO To be implemented by the student
}

E.2 Code Listing for AbstractRuleParser Class

package edu.arizona.spamfilter.xml;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.net.URL;
import java.util.List;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.xml.sax.EntityResolver;
import org.xml.sax.ErrorHandler;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import edu.arizona.spamfilter.ValidationException;
import edu.arizona.spamfilter.beans.Rule;

/**
 * Abstract class that parses an XML configuration file and validates
 * it against a DTD. Resolves internal and external references
 * (including the configuration file and it's DTD) to the current
 * CLASSPATH. Handles parser errors and fatal errors by throwing a
 * SAXException. Handles parser warnings by writing a message to
 * System.err and continuing processing. The {@link #parse()} method
 * is one of several abstract methods to be implemented in
 * subclasses. It should parse the XML configuration file and create
 * the DOM tree by invoking {@link #getRootElement()} which is
 * already implemented.
 */
public abstract class AbstractRuleParser implements EntityResolver,
    ErrorHandler {

/**
 * The CLASSPATH relative path name of the configuration file

private String configFileName;

private String dtdFileName;

private Element rootElement;

public AbstractRuleParser(String configFileName, String dtdFileName) {
    // Validate the parameters
    if (configFileName == null || configFileName.length() == 0) {
        throw new IllegalArgumentException("'configFileName' not specified");
    }

    if (dtdFileName == null || dtdFileName.length() == 0) {
        throw new IllegalArgumentException("'dtdFileName' not specified");
    }

    this.configFileName = configFileName;
    this.dtdFileName = dtdFileName;
}

/**
 * Parses and validates the configuration file, making the
 * allowed-list, blocked-list, and rules available to the
 * corresponding accessor methods. Subclasses should start with
 * the root element returned by (@link #getElement()) to
 * extract the blocked-list, allowed-list, and rules from the DOM
 * tree.
 *
 * @throws ParserConfigurationException
 *     if the XML parser (i.e. Xerces) is configured
public abstract void parse()
    throws ParserConfigurationException, SAXException, IOException, ValidationException;

/**
 * Parses the configuration file, returning the root element of
 * the resulting DOM tree. Caches the tree so multiple calls do
 * not result in multiple parses of the file. Intended to be used
 * by the {@link #parse()} method.
 *
 * @throws ParserConfigurationException
 * if the XML parser (i.e. Xerces) is configured
 * incorrectly.
 *
 * @throws SAXException
 * if the configuration file is malformed, doesn't
 * conform to the DTD, or some other error occurs
 * while parsing the configuration file.
 *
 * @throws IOException
 * if an IO error occurs.
 *
 * @throws ValidationException
 * if the configuration file conforms to the DTD but
 * is invalid for some reason not detected by the
 * DTD.
 *
 * @see #getAllowedAddresses()
 * @see #getAllowedDomains()
 * @see #getBlockedAddresses()
 * @see #getBlockedDomains()
 * @see #getRules()
 */

protected final Element getRootElement()
    throws ParserConfigurationException, SAXException, IOException, ValidationException {
    if (this.rootElement == null) {
        // Create an InputSource for the configuration file
        // with the SystemId set to match the file name so
        // the entity resolver callback method can find the
        // file.
        InputSource inputSource = resolveEntity(null, configFileName);

        // Get the document builder factory
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        factory.setValidating(true);

        // Create the document builder factory
        DocumentBuilder documentBuilder = factory.newDocumentBuilder();

        // Parse the configuration file
        Document domDocument = documentBuilder.parse(inputSource);

        // Get the root element of the configuration file
        Element rootElement = domDocument.getDocumentElement();

        // Cache the root element
        this.rootElement = rootElement;
    }
    return this.rootElement;
}
// Get the document builder and parse the document
DocumentBuilder builder = factory.newDocumentBuilder();
builder.setEntityResolver(this);
builder.setErrorHandler(this);
Document doc = builder.parse(inputSource);

// Get the root element from the DOM tree
this.rootElement = doc.getDocumentElement();
return this.rootElement;

/**
 * Accessor method to return the list of allowed email addresses.
 * @return the allowed addresses.
 */
public abstract List<String> getAllowedAddresses();

/**
 * Accessor method to return the list of allowed domains.
 * @return the allowed domains.
 */
public abstract List<String> getAllowedDomains();

/**
 * Accessor method to return the list of blocked email addresses.
 * @return the blocked addresses.
 */
public abstract List<String> get BlockedAddresses();

/**
 * Accessor method to return the list of blocked domains.
 * @return the blocked domains.
 */
public abstract List<String> getBlockedDomains();

/**
 * Accessor method to return the list of spam filter rules.
 * @return the rules.
 */
public abstract List<Rule> getRules();

// Implementation of the EntityResolver interface

/**
 * Callback method that returns (to the DOM Builder) either the
 * configuration file or the DTD used to validate the
 * configuration file. Both are loaded from the CLASSPATH.
public InputSource resolveEntity(String publicId, String systemId) throws IOException {
    if (systemId == null || systemId.length() == 0
        || systemId.charAt(systemId.length() - 1) == '/') {
        throw new FileNotFoundException("Invalid System ID '" + systemId + '"");
    } else {
        // Get the last part of the System ID Config file
        // name and DTD File Name
        int index = systemId.lastIndexOf('/');
        String systemIdEnd = systemId.substring(index + 1);

        index = configFileName.lastIndexOf('/');
        String configFileNameEnd = configFileName
            .substring(index + 1);

        index = dtdFileName.lastIndexOf('/');
        String dtdFileNameEnd = dtdFileName.substring(index + 1);

        // Lookup the file on the classpath if the filename
        // part of the file's path name matches the filename
        // part of the system ID
        if (configFileName != null
            && configFileNameEnd.equals(systemIdEnd)) {
            // Resolves the configuration file from the
            // CLASSPATH
            URL configFileURL = Thread.currentThread()
                .getContextClassLoader().getResource(configFileName);

            if (configFileURL == null) {
                throw new FileNotFoundException("" + configFileName
                    + "' not found on CLASSPATH");
            }

            return (new InputSource(configFileURL.openStream()));
        } else if (dtdFileName != null
            && dtdFileNameEnd.equals(systemIdEnd)) {
            // Resolves the DTD file from the
            // CLASSPATH
            URL dtdFileURL = Thread.currentThread()
                .getContextClassLoader().getResource(dtdFileName);

            if (dtdFileURL == null) {
                throw new FileNotFoundException("" + dtdFileName
                    + "' not found on CLASSPATH");
            }

            return (new InputSource(dtdFileURL.openStream()));
        }
    }
}
&dtdFileNameEnd.equals(systemIdEnd)) {
// Resolves the DTD from the CLASSPATH
URL dtdURL = Thread.currentThread()
    .getContextClassLoader().getResource(
        dtdFileName);

if (dtdURL == null) {
    throw new FileNotFoundException("'
    + dtdFileName
    + "' not found on CLASSPATH");
}

return (new InputSource(dtdURL.openStream()));
} else {
    return null;
}
}

// Implementation of the ErrorHandler Interface

/**
 * Callback method invoked by the XML parser when it encounters a
 * warning while parsing the document. Writes the exception
 * message to System.err and continues processing.
 *
 * @param ex
 * the exception that caused this method to be
 * invoked.
 */
public void warning(SAXParseException ex) {
    System.err.println(ex);
}

/**
 * Callback method invoked by the XML parser when it encounters
 * an error while parsing the document. Re-throws the exception,
 * resulting in the parse halting with an error.
 *
 * @param ex
 * the exception that caused this method to be
 * invoked.
 * @throws SAXException
 * the exception that caused this method to be
 * invoked.
 */
public void error(SAXParseException ex) throws SAXException {
    throw ex;
}

/**
 * Callback method invoked by the XML parser when it encounters a
 * fatal error while parsing the document. Re-throws the
 * exception, resulting in the parse halting with an error.

package edu.arizona.spamfilter.beans;

import edu.arizona.spamfilter.FilterViolationException;
import edu.arizona.spamfilter.ValidationException;

/**
 * Represents a rule from the configuration file. Use
 * {@link #applyRule(Message)} to determine if the rule is violated
 * by a specific message. Rule clauses may be specified as text or
 * regular expressions depending on the rule's type. If a rule
 * contains multiple rule clauses, the conjunction determines how the
 * clauses are applied. If a clause's 'exists' attribute is set to
 * true, the clause is violated if a match is found in the message.
 * If the exists attribute is set to 'false', the clause is violated
 * if a match is NOT found in the message.
 */
public interface Rule {

/**
 * An enumeration type that represents the valid rule types from
 * the configuration file.
 */
public enum Type {

/**
 * Indicates that the rule clauses should be interpreted as
 * regular expressions.
 */
Regex,

/**
 * Indicates that the rule clauses should be interpreted as
 * standard text.
 */
Text
};

/**
 * An enumeration type that represents the valid values for the
* conjunction attribute of a rule.
 */
public enum Conjunction {
 /**
 * Indicates that the rule is only violated if <b>all</b> of its clauses are violated.
 */
    And,

 /**
 * Indicates that the rule is violated if <b>any</b> of its clauses are violated.
 */
    Or
};

/**
 * Returns the rule's id attribute. Id's are unique within a configuration file.
 * @return the id.
 */
public int getId();

/**
 * Sets the rule's id attribute.
 * @param id the id.
 */
public void setId(int id);

/**
 * Returns the rule's type.
 * @return the type.
 */
public Type getType();

/**
 * Sets the rule's type.
 * @param type the type.
 */
public void setType(Type type);

/**
 * Returns the rule's conjunction. The conjunction determines the meaning of the rule when multiple rule clauses are specified.
 * For example, if the rule's conjunction is 'And', the rule is only violated if all of the rule's clauses are violated by the message. If the rule's conjunction is 'Or' the rule is violated if any of the rule's clauses are violated.
 */
public Conjunction getConjunction();

/**
 * Sets the rule's conjunction.
 * @param conjunction the conjunction.
 * @see #getConjunction()
 */
public void setConjunction(Conjunction conjunction);

/**
 * Returns the rule's 'to' clause.
 * @return the 'to' clause or null if no 'to' clause was specified for the rule.
 */
public String getTo();

/**
 * Indicates whether the rule is violated by messages with or without a matching 'to' recipient. If true, the rule is violated by messages that have a matching 'to' recipient, otherwise, the rule is violated by messages that do not have a matching 'to' recipient.
 * @return the 'exists' value for the rule's 'to' clause.
 */
public boolean isToExists();

/**
 * Sets the rule's 'to' clause.
 * @param to the 'to' clause.
 * @param exists indicates whether the rule is violated by messages with or without a matching 'to' recipient.
 * @see #isToExists()
 */
public void setTo(String to, boolean exists);

/**
 * Returns the rule's 'cc' clause.
 * @return the 'cc' clause or null if no 'cc' clause was specified for the rule.
 */
public String getCc();
* Indicates whether the rule is violated by messages with or without a matching 'cc' recipient. If true, the rule is violated by messages that have a matching 'cc' recipient, otherwise, the rule is violated by messages that do not have a matching 'cc' recipient.

* @return the 'exists' value for the rule's 'cc' clause.
*/
public boolean isCcExists();

/**
 * Sets the rule's 'cc' clause.
 *
 * @param cc
 *   the 'cc' clause.
 * @param exists
 *   indicates whether the rule is violated by messages with or without a matching 'cc' recipient.
 * @see #isCcExists()
 */
public void setCc(String cc, boolean exists);

/**
 * Returns the rule's 'bcc' clause.
 *
 * @return the 'bcc' clause or null if no 'bcc' clause was specified for the rule.
 */
public String getBcc();

/**
 * Indicates whether the rule is violated by messages with or without a matching 'bcc' recipient. If true, the rule is violated by messages that have a matching 'bcc' recipient, otherwise, the rule is violated by messages that do not have a matching 'bcc' recipient.
 *
 * @return the 'exists' value for the rule's 'bcc' clause.
 */
public boolean isBccExists();

/**
 * Sets the rule's 'bcc' clause.
 *
 * @param bcc
 *   the 'bcc' clause.
 * @param exists
 *   indicates whether the rule is violated by messages with or without a matching 'bcc' recipient.
 * @see #isBccExists()
 */
public void setBcc(String bcc, boolean exists);
* Returns the rule's 'from' clause.
* @return the 'from' clause or null if no 'from' clause was
  specified for the rule.
*/
public String getFrom();

/**
 * Indicates whether the rule is violated by messages that are or
 * are not from a matching sender. If true, the rule is violated
 * by messages that are sent by a matching sender, otherwise, the
 * rule is violated by messages that are not sent by a matching
 * sender.
 * @return the 'exists' value for the rule's 'from' clause.
 */
public boolean isFromExists();

/**
 * Sets the rule's 'from' clause.
 * @param from      the 'from' clause.
 * @param exists    indicates whether the rule is violated by messages
 *                  sent by a matching sender.
 * @see #isFromExists()
 */
public void setFrom(String from, boolean exists);

/**
 * Returns the rule's 'subject' clause.
 * @return the 'subject' clause or null if no 'subject' clause
 *         was specified for the rule.
 */
public String getSubject();

/**
 * Sets the rule's 'subject' clause.
 * @param subject  the 'subject' clause.
 */
public void setSubject(String subject);

 /**
 * Returns the rule's 'body' clause.
 * @return the 'body' clause or null if no 'body' clause was
 *         specified for the rule.
 */
public String getBody();
/**
 * Sets the rule's 'body' clause.
 * @param body            the 'body' clause.
 */
public void setBody(String body);

/**
 * Performs validations of the rule beyond what the XML parser
 * can do by comparing the configuration file to the DTD.
 * @throws ValidationException
 *          if the rule is invalid.
 */
public void validate() throws ValidationException;

/**
 * Applies the rule to the specified message, throwing an
 * exception if the rule is violated.
 * @param message            the message to be tested against the rule.
 * @throws FilterViolationException
 *                      if the message violates the rule.
 */
public void applyRule(Message message)
    throws FilterViolationException;
}

E.4 Code Listing for Message Class

package edu.arizona.spamfilter.beans;

import java.util.ArrayList;
import java.util.List;

/**
 * Represents an email message.
 */
public class Message {

    /**
     * The list of 'to' recipients of the message.
     */
    private List<String> to;

    /**
     * The list of 'cc' recipients of the message.
     */
    private List<String> cc;

    /**
     * The list of 'bcc' recipients of the message.
     */
    private List<String> bcc;

    /**
     * The list of 'subject' of the message.
     */
    private String subject;

    /**
     * The list of 'from' of the message.
     */
    private String from;

    /**
     * The list of 'body' of the message.
     */
    private String body;

    /**
     * The list of 'attachment' of the message.
     */
    private List<String> attachments;

    //...
* The list of 'bcc' recipients of the message.
* /
private List<String> bcc;

/**
* The sender of the message.
* /
private String from;

/**
* The subject of the message.
* /
private String subject;

/**
* The body of the message.
* /
private String body;

/**
* Creates a new empty message.
* /
public Message() {
  super();
}

/**
* Creates a new message.
* @param to
*   the recipient of the message.
* @param from
*   the sender of the message.
* @param subject
*   the subject of the message.
* @param body
*   the body of the message.
* /
public Message(String to, String from, String subject,
String body) {
  addTo(to);
  setFrom(from);
  setSubject(subject);
  setBody(body);
}

/**
* Clears the message by setting all values to null.
* /
public void clear() {
  to = null;
  cc = null;
  bcc = null;
from = null;
subject = null;
body = null;
}

/**
 * Returns the list of 'to' recipients.
 * @return the 'to' recipients.
 */
public List<String> getTo() {
    return to;
}

/**
 * Sets the list of 'to' recipients.
 * @param toList the 'to' recipients.
 */
public void setTo(List<String> toList) {
    this.to = toList;
}

/**
 * Adds a 'to' recipient to the message.
 * @param to the recipient to be added.
 */
public void addTo(String to) {
    if (to != null && to.trim().length() > 0) {
        if (this.to == null) {
            this.to = new ArrayList<String>();
        }
        this.to.add(to);
    }
}

/**
 * Returns the list of 'cc' recipients.
 * @return the 'cc' recipients.
 */
public List<String> getCc() {
    return cc;
}

/**
 * Sets the list of 'cc' recipients.
 * @param ccList the 'cc' recipients.
 */
/*
  public void setCc(List<String> ccList) {
    this.cc = ccList;
  }

  /**
   * Adds a 'cc' recipient to the message.
   * @param cc
   * the recipient to be added.
   */
  public void addCc(String cc) {
    if (cc != null && cc.trim().length() > 0) {
      if (this.cc == null) {
        this.cc = new ArrayList<String>();
      }
      this.cc.add(cc);
    }
  }

  /**
   * Returns the list of 'bcc' recipients.
   * @return the 'bcc' recipients.
   */
  public List<String> getBcc() {
    return bcc;
  }

  /**
   * Sets the list of 'bcc' recipients.
   * @param bccList
   * the 'bcc' recipients.
   */
  public void setBcc(List<String> bccList) {
    this.bcc = bccList;
  }

  /**
   * Adds a 'bcc' recipient to the message.
   * @param bcc
   * the recipient to be added.
   */
  public void addBcc(String bcc) {
    if (bcc != null && bcc.trim().length() > 0) {
      if (this.bcc == null) {
        this.bcc = new ArrayList<String>();
      }
      this.bcc.add(bcc);
    }
  }
/**
 * Returns the sender of the message.
 * @return the sender.
 */
public String getFrom() {
    return from;
}

/**
 * Sets the sender of the message.
 * @param from
 *     the sender.
 */
public void setFrom(String from) {
    this.from = from;
}

/**
 * Returns the subject of the message.
 * @return the subject.
 */
public String getSubject() {
    return subject;
}

/**
 * Sets the subject of the message.
 * @param subject
 *     the subject.
 */
public void setSubject(String subject) {
    this.subject = subject;
}

/**
 * Returns the message body.
 * @return the message body.
 */
public String getBody() {
    return body;
}

/**
 * Sets the message body.
 * @param body
 *     the message body.
 */
public void setBody(String body) {
    this.body = body;
}
}

E.5 Code Listing for ArgumentParser Class

package edu.arizona.spamfilter.util;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
/**
 * Parses command line arguments from a command line argument array.
 * Arguments must be in the form: -argument or -argument
 * &lt;value&gt; (without the angle brackets around the values).
 * /
 * public class ArgumentParser {
 **
 * Contains the command line arguments that are parsed out of the
 * argument array.
 * /
 * private HashMap<String, List<String>> arguments = new HashMap<String,
 * List<String>>();

 /**
 * Creates a new ArgumentParser and initializes it by parsing the
 * specified args array to make it ready to return any parameters
 * found in the args array.
 * *
 * @param args
 * the array containing the command line arguments to
 * be parsed.
 * /
 * public ArgumentParser(String[] args) {
 * if (args != null) {
 *     parseArgs(args);
 * };
 * }

 /**
 * Called by the constructor to parse any command line arguments
 * out of 'args'. Makes this ArgumentParser ready to return any
 * parameters found in the 'args' array.
 * *
 * @param args
 * the array containing the command line arguments to
 * be parsed.
 * /
 * private void parseArgs(String[] args) {
for (int i = 0; i < args.length; i++) {
    if (args[i].startsWith("-")) {
        String argumentName = args[i].substring(1);

        // If there is at least one parameter left it will
        // either be the value or the value will be null and
        // the next parameter will be another argument name.
        // Get the next parameter and find out what it is.
        // If there isn't at least one parameter left, the
        // value of the current argument is null.
        String argumentValue = null;
        if (args.length > (i + 1)) {
            argumentValue = args[i + 1];

            // Now see if argumentValue is really another
            // argument name
            if (argumentValue.startsWith("-")) {
                // The next parameter is another argument
                // name, not a value, so set the value to
                // null.
                argumentValue = null;
            } else {
                // The next parameter was the argument value
                // for the argument we were processing, skip
                // the argument value so we can get
                // the next argument name on the next pass.
                i++;
            }
        }
    }
}

List<String> argumentList = getArguments(argumentName);
if (argumentList == null) {
    argumentList = new ArrayList<String>();
    arguments.put(argumentName.toLowerCase(),
                  argumentList);
}

argumentList.add(argumentValue);
}

/**
 * Performs a case insensitive search for a command line argument
 * named 'argName'. If multiple arguments exist with that name,
 * it returns one of them. It is undefined which of multiple
 * arguments with the same name is returned.
 *
 * @param argName
 * the name of the command line argument.
 * @return the value of the command line argument or null if the
 * argument does not exist.
 */
public String getArgument(String argName) {
List<String> argumentList = arguments.get(argName.toLowerCase());
return argumentList == null ? null : argumentList.get(0);
}
/**
* Performs a case insensitive search for command line arguments named 'argName'.
* @param argName the name of the command line argument(s).
* @return the value of the command line argument(s) with the specified name or null if none exist with that name.
*/
public List<String> getArguments(String argName) {
return arguments.get(argName.toLowerCase());
}
/**
* Performs a case insensitive search for a command line argument named 'argName'. Returns the argument as an int. If multiple arguments exists with that name, it returns one of them.
* @param argName the name of the command line argument.
* @return the value of the command line argument or -1 if the argument does not exist.
* @throws NumberFormatException if the value cannot be converted to an int.
*/
public int getIntArgument(String argName) throws NumberFormatException {
return (getArgument(argName) == null) ? -1 : Integer.parseInt(getArgument(argName));
}
/**
* Performs a case insensitive search for a command line argument named 'argName'.
* @param argName the name of the command line argument.
* @return true if at least one argument with that name exists, otherwise, false.
*/
public boolean containsArgument(String argName) {
return arguments.containsKey(argName.toLowerCase()) ? true : false;
}
APPENDIX F:

CHA-AS ALGORITHM PSEUDOCODE

This appendix contains Pseudocode describing the logic of the CHA-AS algorithm and supporting data structures. The Pseudocode was derived from the completed algorithm and is based on Java syntax with the following modifications intended to simplify and compress the logic:

- Return statements can return multiple comma separated variables
- Method return types are a comma separated list of return types
- Arrays are dynamically resizable
- Contents of arrays can be modified using Java operators (such as += for appending an item to the end of the array)
- Arrays have the same methods as the java.util.Collection Interface
- The syntax /*< ... >*/ denotes code that has been replaced by a textual comment to simplify the description of the logic.

The structure of the Pseudocode has been simplified to aid readability by eliminating inheritance, by replacing member variables with local variables, by making methods from different classes appear as stand-alone functions, and by introducing redundant code in place of method calls in some cases.
F.1 CHA-AS Algorithm Pseudocode

/**
 * Compares two projects and identifies and returns the complete set
 * of methods and fields potentially impacted by the changes between
 * the two projects.
 *
 * @param originalProjectName the name of the project to be used as
 *        the baseline for comparison.
 * @param editedProjectName the name of the project to be compared
 *        to the original project.
 */
FieldInfo[], MethodInfo[] getSourceChangeImpact(
    String originalProjectName,
    String editedProjectName) {

    // Parse the original and edited projects to find all field and
    // method references
    ChaReferenceParser originalProjectParser =
        parse(originalProjectName);
    ChaReferenceParser editedProjectParser = parse(editedProjectName);

    Document atomicChanges = /*<Invoke Chianti with original and
    edited project names to get XML file describing atomic
    changes>*;*/

    // Get the fields and methods to be used as a starting point for
    // calculating the change impact
    FieldInfo[] startingFields = getStartingFields();
    MethodInfo[] startingMethods = getStartingMethods();

    return getChangeImpact(startingFields, startingMethods, editedProjectParser);
}

/**
 * Parses the specified project and caches all method and field
 * references, along with their class, the method or field
 * containing the reference, the position of the reference and the
 * top-most enclosing loop position of the reference. For references
 * enclosed in a loop or set of nested loops, the enclosing loop
 * position is the position of the first character of the top-most,
 * or outer-most, enclosing loop. Enclosing loop positions are used
 * to ensure that all code in loop containing impacted code is also
 * marked as impacted.
 *
 * @param projectName the name of the project to be parsed.
 */
void parse(String projectName) {

Project project = /*<Lookup 'projectName' from the Eclipse workspace>*/;
Type[] types = /* <Find all source files in 'project' > */;

// Pre-process the source files to support CHA-based reference expansion
for (Type type : types) {
   /*<Extract and cache all constructors, methods, and fields>*/ 
}

TypeHierarchy typeHierarchy = /*<Use Eclipse to create the datatype inheritance hierarchy of all types in the project>*/;

for (Type type : types) {
    /*<Extract and cache all method and field references in 'type'>*/

    // Perform CHA expansion of all method references to cache methods invokable by dynamic method invocation
    for (MethodFieldReference or MethodMethodReference ref : /*<Get each reference to a method from the cache>*/) {
        Type[] subtypes = typeHierarchy.getAllSubtypes(type);

        for (Type subtype : subtypes) {
            if (/*<'subtype' contains method matching 'ref.referencingMethod'*/) {
                /*<Cache subtype's matching method with 'ref.referencingMethod'>*/;
            }
        }
    }
}

/**
 * Gets the fields to be used as a starting point for determining the change impact. These are the fields reported by Chianti as having been added or changed.
 * @param atomicChanges the XML document returned by Chianti containing complete set of atomic changes.
 */
FieldInfo[] getStartingFields(Document atomicChanges) {

    FieldInfo[] startingFields = atomicChanges.extractAddedFields();
    startingFields += atomicChanges.extractChangedFields();
    return startingFields;
}
/**
 * Gets the methods to be used as a starting point for determining
 * the change impact. These are the methods reported by Chianti as
 * having been added, changed, or impacted by lookup changes
 * (resulting from dynamic method invocation), plus the methods
 * potentially impacted by deleted methods.
 *
 * @param atomicChanges the XML document returned by Chianti
 *          containing complete set of atomic changes.
 * @param originalProjectParser the parser containing cached
 *          CHA-expanded reference information obtained from the
 *          source files of the original project.
 * @param editedProjectParser the parser containing cached
 *          CHA-expanded reference information obtained from the
 *          source files of the edited project.
 */
MethodInfo[] getStartingMethods(
    Document atomicChanges,
    ChaReferenceParser originalProjectParser,
    ChaReferenceParser editedProjectParser) {

    MethodInfo[] startingMethods = atomicChanges.extractAddedMethods();
    startingMethods += atomicChanges.extractChangedMethods();
    startingMethods += atomicChanges.extractLookupChangedMethods();
    startingMethods += getDeletedMethodImpact(atomicChanges,
                                              originalProjectParser, editedProjectParser);

    return startingMethods;
}

/**
 * Gets the methods potentially impacted by deleted methods. These
 * include methods referencing fields that had been set by the
 * deleted method, methods that had been invoked by the deleted
 * method, and methods that had been invoking the deleted method.
 * Only methods still existing in the edited project are reported.
 *
 * @param atomicChanges the XML document returned by Chianti
 *          containing complete set of atomic changes.
 * @param originalProjectParser the parser containing cached
 *          CHA-expanded reference information obtained from the
 *          source files of the original project.
 * @param editedProjectParser the parser containing cached
 *          CHA-expanded reference information obtained from the
 *          source files of the edited project.
 */
MethodInfo[] getDeletedMethodImpact(
    Document atomicChanges,
    ChaReferenceParser originalProjectParser,
    ChaReferenceParser editedProjectParser) {

    MethodInfo[] deletedMethods =
        atomicChanges.extractDeletedMethods();

for (MethodInfo deletedMethod : deletedMethods) {
    // Include methods referencing the fields that had been modified
    // by the deleted method (except for fields that were also
    // deleted).
    MethodFieldReference[] modifiedFields =
        originalProjectParser.getFieldsSetByMethod(deletedMethod);
    modifiedFields += originalProjectParser.
        getFieldsInitializedByMethod(deletedMethod);

    for (MethodFieldReference fieldRef : modifiedFields) {
        // If the edited project still contains the field
        if (editedProjectParser.parsedField(
            fieldRef.referencedField)) {
            startingMethods += editedProjectParser.
                getMethodsReferencingField(fieldRef.referencedField);
        }
    }

    // Include methods that had been invoked by the deleted
    // method (except for invoked methods that were also
    // deleted).
    MethodMethodReference[] invokedMethods =
        originalProjectParser.getMethodsInvokedByMethod(deletedMethod);
    for (MethodMethodReference invokedMethodRef : invokedMethods) {
        // If the edited project still contains the invoked method
        if (editedProjectParser.parseMethod(
            invokedMethodRef.invokedMethod)) {
            startingMethods += invokedMethodRef.invokedMethod;
        }
    }

    // Include methods that had been invoking the deleted
    // method (except for invoking methods that were also
    // deleted).
    MethodMethodReference[] invokingMethods =
        originalProjectParser.getMethodsInvokingMethod(deletedMethod);
    for (MethodMethodReference invokingMethodRef : invokingMethods) {
        // If the edited project still contains the invoking method
        if (editedProjectParser.parseMethod(
            invokingMethodRef.invokingMethod)) {
            startingMethods += invokingMethodRef.invokingMethod;
        }
    }
}
/**
 * Returns the complete set of methods and fields potentially
 * impacted by the specified starting methods and fields. Starts by
 * recursively expanding the set of starting fields to include all
 * fields initialized by a starting field. Enqueues all methods that
 * access a starting field and all starting methods. Then processes
 * the queue until empty by invoking processMethod(...) for each item
 * in the queue.
 *
 * @param startingFields the fields to be used as a starting point
 *          for assessing the change impact.
 * @param startingMethods the methods to be used as a starting
 *          point for assessing the change impact.
 * @param editedProjectParser the parser containing cached
 *          CHA-expanded reference information obtained from the
 *          source files of the edited project.
 */
FieldInfo[], MethodInfo[] getChangeImpact(
    FieldInfo[] startingFields,
    MethodInfo[] startingMethods,
    ChaReferenceParser editedProjectParser) {

FieldInfo[] impactedFields += startingFields;
impactedFields += /*<Recursively get fields initialized by
starting fields>*/;

Queue methodQueue = new Queue();

// Enqueue all methods that directly access an impacted field
for (FieldInfo impactedField : impactedFields) {
    MethodFieldReference[] methods = editedProjectParser.
        getMethodsReferencingField(impactedField);

    for (MethodFieldReference methodRef : methods) {
        Range range = /*<a subrange of the referencing method starting
        with the enclosing position of the field reference and
        extending to the end of the method>*/;
        MethodRangeInfo methodRangeInfo = new MethodRangeInfo();
        methodRangeInfo.method = methodRef.referencingMethod;
        methodRangeInfo.range = range;

        if(/*<methodQueue does not contain a MethodRangeInfo with
        method = 'methodRangeInfo.method'>*/) {
            methodQueue.enqueue(methodRangeInfo);
        } else if(/*<The range does not contain all characters in
        'methodRangeInfo.range'>*/) {
            /*<Expand the range of the contained method to include
            'methodRangeInfo.range'>*/
        }
    }
}

// Enqueue the starting methods
if (startingMethods != null) {
for (MethodInfo method : startingMethods) {
    Range range = /*<A range containing all characters in the
    method>*/

    if(/*<methodQueue contains a MethodRangeInfo matching
    'method'>*/) {
        /*<Replace the range in the matching method with 'range'>*/
    } else {
        MethodRangeInfo methodRangeInfo = new MethodRangeInfo();
        methodRangeInfo.method = method;
        methodRangeInfo.range = range;

        methodQueue.enqueue(methodRangeInfo);
    }
}

MethodRangeInfo[] impactedMethodRanges;

// Process the method queue to identify all impacted methods
while (!methodQueue.isEmpty()) {
    MethodRangeInfo method = methodQueue.remove();
    processMethod(method, methodQueue, impactedFields,
                  impactedMethodRanges, editedProjectParser);
}

// Extract MethodInfo objects from impactedMethodRanges
MethodInfo[] impactedMethods;
for (MethodInfo methodRange : impactedMethodRanges) {
    impactedMethods += methodRange.method;
}

return impactedFields, impactedMethods;

/**
 * Processes a single method previously identified as having been
 * impacted by adding its processing range to
 * 'impactedMethodRanges', and invoking processFieldsSet(...),
 * processInvokingMethods(...), and processInvokedMethods(...).
 *
 * @param method the method whose invoking methods are to be
 *              processed.
 * @param methodQueue the queue of methods waiting to be processed.
 * @param impactedFields the current set of fields determined by the
 *              algorithm to be impacted.
 * @param impactedMethodRanges the current set of method subranges
 *              determined by the algorithm to be impacted.
 * @param editedProjectParser the parser containing cached
 *              CHA-expanded reference information obtained from the
 *              source files of the edited project.
 * /
 * void processMethod(}
MethodRangeInfo method,
Queue methodQueue,
FieldInfo[] impactedFields,
MethodRangeInfo[] impactedMethodRanges,
ChaReferenceParser editedProjectParser) {

    processFieldsSet(method, methodQueue, impactedFields,
                      impactedMethodRanges, editedProjectParser);

    processInvokingMethods(method, methodQueue, impactedFields,
                           impactedMethodRanges, editedProjectParser);

    processInvokedMethods(method, methodQueue, impactedFields,
                          impactedMethodRanges, editedProjectParser);

    if (/*<impactedMethodRanges contains a MethodRangeInfo with a
        method matching 'method'>*/) {
        /*<Replace the range in the matching MethodRangeInfo with
           'method.range'>*/
    } else {
        impactedMethodRanges += method;
    }
}

/**
* Finds the CHA-expanded set of fields potentially impacted by the
* method currently being processed, then finds all methods that
* contain a reference to any of these fields and processes them by
* calling processReference(...). All fields initialized by the
* current method and all fields set from within the current methods
* processing range are considered impacted by it.
* *
* @param method the method whose invoking methods are to be
*          processed.
* @param methodQueue the queue of methods waiting to be processed.
* @param impactedFields the current set of fields determined by the
*          algorithm to be impacted.
* @param impactedMethodRanges the current set of method subranges
*          determined by the algorithm to be impacted.
* @param editedProjectParser the parser containing cached
*          CHA-expanded reference information obtained from the
*          source files of the edited project.
*/
void processFieldsSet(
    MethodRangeInfo method,
    Queue methodQueue,
    FieldInfo[] impactedFields,
    MethodRangeInfo[] impactedMethodRanges,
    ChaReferenceParser editedProjectParser) {

    MethodFieldReference[] modifiedFields = /*<Get fields set
        within 'method.range' of 'method' in edited project>*/;
    modifiedFields += editedProjectParser.
getFieldsInitializedByMethod(method);

for (MethodFieldReference fieldRef : modifiedFields) {
    FieldInfo referencedField = fieldRef.referencedField;

    if (!impactedFields.contains(referencedField)) {
        for (MethodFieldReference methodRef :
            editedProjectParser.getMethodsReferencingField(
                referencedField)) {
            processReference(method, methodRef.referencingMethod,
                methodRef.enclosingPosition, methodQueue, impactedFields,
                impactedMethodRanges, editedProjectParser);
        }
        impactedFields.add(referencedField);
    }
}

/**
 * Finds the CHA-expanded set of methods potentially invoking the
 * method currently being processed and processes them by calling
 * processReference(...).
 *
 * @param method the method whose invoking methods are to be
 * processed.
 * @param methodQueue the queue of methods waiting to be processed.
 * @param impactedFields the current set of fields determined by the
 * algorithm to be impacted.
 * @param impactedMethodRanges the current set of method subranges
 * determined by the algorithm to be impacted.
 * @param editedProjectParser the parser containing cached
 * CHA-expanded reference information obtained from the
 * source files of the edited project.
 */
void processInvokingMethods(
    MethodRangeInfo method,
    Queue methodQueue,
    FieldInfo[] impactedFields,
    MethodRangeInfo[] impactedMethodRanges,
    ChaReferenceParser editedProjectParser) {

    for (MethodMethodReference invokingMethodRef :
        editedProjectParser.getMethodsInvokingMethod(
            method.methodInfo)) {
        MethodInfo invokingMethod = invokingMethodRef.invokingMethod;

        processReference(method, invokingMethod,
            invokingMethodRef.enclosingPosition, methodQueue,
            impactedFields, impactedMethodRanges,
            editedProjectParser);
    }
/**
 * Finds the CHA-expanded set of methods potentially invoked from
 * within the processing range of the method currently being
 * processed and processes them by calling processReference(...).
 * @param method the method whose invoked methods are to be
 * processed.
 * @param methodQueue the queue of methods waiting to be processed.
 * @param impactedFields the current set of fields determined by the
 * algorithm to be impacted.
 * @param impactedMethodRanges the current set of method subranges
 * determined by the algorithm to be impacted.
 * @param editedProjectParser the parser containing cached
 * CHA-expanded reference information obtained from the
 * source files of the edited project.
 */

void processInvokedMethods(
    MethodRangeInfo method,
    Queue methodQueue,
    FieldInfo[] impactedFields,
    MethodRangeInfo[] impactedMethodRanges,
    ChaReferenceParser editedProjectParser) {

    MethodMethodReference[] invokedMethods = /*<Get methods
    invoked within 'method.range' of 'method' in edited
    project>*/;

    for (MethodMethodReference invokedMethodRef : invokedMethods) {
        processReference(method, invokedMethodRef.invokedMethod, 0,
                         methodQueue, impactedFields, impactedMethodRanges,
                         editedProjectParser);
    }
}
/**
 * Processes a single method reference found in 'currentMethod' by
 * enqueueing the subrange of the referenced or referencing method
 * from the point of the reference (or its top-most enclosing loop),
 * to the end of the method. If the method is already enqueued the
 * processing range is expanded as necessary. Also handles the
 * special case of recursive references to the method currently being
 * processed by expanding the range and re-processing the method if
 * necessary.
 *
 * @param currentMethod the method containing the method reference
 *          to be processed.
 * @param referenceMethod the method either referencing or
 *          referenced by 'method'
 * @param referenceEnclosingLoopPosition the top-most enclosing
 *          loop position of the reference being processed.
 * @param methodQueue the queue of methods waiting to be processed.
 * @param impactedFields the current set of fields determined by the
 *          algorithm to be impacted.
 * @param impactedMethodRanges the current set of method subranges
 *          determined by the algorithm to be impacted.
 * @param editedProjectParser the parser containing cached
 *          CHA-expanded reference information obtained from the
 *          source files of the edited project.
 */

void processReference(
    MethodRangeInfo currentMethod,
    MethodInfo referenceMethod,
    int referenceEnclosingLoopPosition,
    Queue methodQueue,
    FieldInfo[] impactedFields,
    MethodRangeInfo impactedMethodRanges,
    ChaReferenceParser editedProjectParser) {

    Range referenceMethodRange = /*<Get the range of characters that
    need to be processed for the reference method, starting at the
    enclosing loop position and adjusting for any part of the method
    already processed.>*/;

    if (referenceMethodRange != null) {
        if (referenceMethod.equals(currentMethod)) {
            // Found a recursive reference to the method currently being
            // processed.
            if (/*<currentMethod.range' includes the entire range of
                referencingMethodRange>*/) {
                // Range in question is already covered.
                // currentMethod can be ignored
            } else {
                // Need to expand the range to be processed of
                // 'currentMethod'. Start over with the expanded
                // range (recursive call).
                currentMethod.range = referenceMethodRange;
                processMethod(currentMethod, methodQueue,
                              impactedFields, impactedMethodRanges,

```
editedProjectParser);
}
} else {
    if(/*<methodQueue does not contain a MethodRangeInfo with
        method = 'referenceMethod'>*/) {
        MethodRangeInfo methodRangeInfo = new MethodRangeInfo();
        methodRangeInfo.method = referenceMethod;
        methodRangeInfo.range = referenceMethodRange;
        methodQueue.enqueue(methodRangeInfo);
    } else if;/*<The range does not contain all characters in
                  'referenceMethodRange'>*/ {
        /*<Expand the range of the enqueued method to include
           'methodRangeInfo.range'>*/
    }
}
}

F.2 CHA-AS Algorithm Data Structures

public class FieldInfo {
    /**
     * The fully qualified name of the class containing the field.
     */
    public String binaryClassName;
    public String fieldName;
}

public class MethodInfo {
    /**
     * The fully qualified name of the class containing the method.
     */
    public String binaryClassName;
    public String methodSignature;
}
public class MethodRangeInfo {
    public MethodInfo methodInfo;

    /**
     * The range of the method that either was or needs to be
     * processed.
     */
    public Range range;
}

public class Range {
    public int startPosition;
    public int endPosition;
}

public class MethodFieldReference {

    /**
     * The field that was referenced.
     */
    public FieldInfo referencedField;

    /**
     * The method containing the reference.
     */
    public MethodInfo referencingMethod;

    /**
     * The character position of the reference.
     */
    public int position;

    /**
     * The position of the first character of the top-most enclosing
     * loop containing the reference, or the position of the reference
     * if not contained in a loop.
     */
    public int enclosingPosition;
}
public class MethodMethodReference {

/**
 * The method containing the reference.
 */
public MethodInfo invokingMethod;

/**
 * The method that was referenced (invoked).
 */
public MethodInfo invokedMethod;

/**
 * The character position of the reference.
 */
public int position;

/**
 * The position of the first character of the top-most enclosing loop containing the reference, or the position of the reference if not contained in a loop.
 */
public int enclosingPosition;
}
REFERENCES


*Curve Fitting Toolbox: For Use with MatLab.* (2006).: The MathWorks.


