COUPLING-BASED ANALYSIS OF OBJECT-ORIENTED SOFTWARE

by

Aynur Abdurazik

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of the
the Requirements for the Degree
of
Doctor of Philosophy
Information and Software Engineering

Committee:

_____	Jeff Offutt, Dissertation Director
_____	Hassan Gomaa
_____	Paul Ammann
_____	Elizabeth White
_____	Sanjeev Setia, Chairman, Department of Information and Software Engineering
_____	Lloyd J. Griffiths, Dean, School of Information Technology and Engineering
Date: _____	Spring Semester 2007 George Mason University Fairfax, VA

Coupling-based Analysis of Object-Oriented Software

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Aynur Abdurazik
Bachelor of Science
Beijing University of Posts and Telecommunications, 1992
Master of Science
George Mason University, 1999

Director: Jeff Offutt, Professor
Department of Information and Software Engineering

Spring Semester 2007
George Mason University
Fairfax, VA

Dedication

I dedicate this dissertation to my family.

Acknowledgments

I would like to thank my advisor, Dr. Jeff Offutt, for his support and guidance during this long journey. He has always believed in me and gave me encouragement when I needed it.

I would like to thank Dr. Hassan Gomaa, Dr. Paul Ammann and Dr. Elizabeth White for serving in my committee.

I thank Dr. Roger Alexander for his support and encouragement in my early days.

I thank Prof. Naomi Altman, my mentor through mentornet, for her understanding, support, and guidance.

I thank Lynne Rosenthal for her giving me an opportunity to work at NIST.

My research work was sponsored in part by the National Science Foundation under grant number CCR-0097056. I would like to thank Dr. Steve Schach, the co-investigator on this grant, for his collaboration.

I wish to thank Prof. Lionel Briand and his colleagues for sharing their experimental materials.

I thank my dear friend Muna Al-Razgan, for always being there for me.

Most of all, I thank my family for their love, support, and encouragement.

Table of Contents

Page

Abstract xiv
1 INTRODUCTION 1
1.1 The Problem 4
1.2 Thesis Statement 4
1.3 Introduction to the Three Focus Problems 4
1.3.1 Class Integration and Test Order Problem 5
1.3.2 Change Impact Analysis 6
1.3.3 Design Pattern Detection 8
2 LITERATURE REVIEW 10
2.1 Coupling Background 10
2.2 Class Integration and Test Order 15
2.3 Change Impact Analysis 19
2.4 Design Pattern Detection 26
3 COUPLING MODEL 32
3.1 Object-Oriented Approach 35
3.1.1 Object-Oriented Relationship Types for Classes and Objects . 35
3.1.2 Object-Oriented Connection Types Among Classes 39
3.1.3 Metamodel 39
3.2 Object-Oriented Coupling Measures 40
3.2.1 Derivation of Object-Oriented Coupling Measures 44
3.2.2 Formal Definition of Coupling Measures 50
3.2.3 A Unified Representation of Coupling Measures 56
3.2.4 A Simple Example 60
3.3 Java Source Code Patterns for Object-Oriented Couplings 62
3.3.1 Java Source Code Pattern for Association Coupling 63
3.3.2 Java Source Code Pattern for Aggregation Coupling 64
3.3.3 Java Source Code Pattern for Composition Coupling 65

3.3.4 Java Source Code Pattern for Usage Dependency Coupling . . 67
3.3.5 Java Source Code Pattern for Generalization (Inheritance) Coupling 70
3.3.6 Java Source Code Pattern for InterfaceRealization Coupling . 74
3.3.7 Java Source Code Pattern for Exception Coupling 74
4 PROOF OF CONCEPT TOOL: JCAT 78
4.1 JCAT User Interface 80
4.2 JCAT Design 82
4.3 Summary Tables 86
5 APPLICATIONS OF THE COUPLING MODEL 93
5.1 Applying Coupling Measures to Class Integration and Test Order . . 93
5.2 Applying Coupling Measures to Change Impact Analysis 95
5.3 Applying of Coupling Measures to Design Pattern Detection 96
6 COUPLING-BASED CLASS INTEGRATION AND TEST ORDER . . . 97
6.1 Summary of Existing Solutions 98
6.2 A New Model and Algorithms 99
6.2.1 Modeling Class Integration and Test Order 99
6.2.2 Measuring Stub Complexity 103
6.2.3 Heuristics Algorithm for Breaking Cycles Using Edge Weights 106
6.2.4 Applying Algorithm 3 to A Special Case 110
6.2.5 Heuristics Algorithm for Breaking Cycles Using Node Weights 112
6.2.6 Heuristics Algorithm for Breaking Cycles Using Node and Edge Weights 118
6.3 Algorithm for Ordering Classes for Integration Testing 123
6.4 Case Study 127
6.5 Summary of Existing Graph Algorithms for Cycle Elimination 130
6.6 Summary 131
7 COUPLING-BASED CHANGE IMPACT ANALYSIS 133
7.1 Class Change Impact and Change Sensitivity 137
7.2 Measuring Class Change Impact and Class Change Sensitivity 139
7.2.1 Usefulness of Measures 142
7.2.2 Object-Oriented Program Changes 143

7.2.3 Computing Impact Sets for Individual Changes 144
7.3 Case Study 150
7.3.1 Visualization of Metrics 152
7.4 Summary 155
8 COUPLING-BASED DESIGN PATTERN DETECTION 157
8.1 Representation of System and Patterns 158
8.1.1 Analysis and Representation of Design Patterns 159
8.1.2 Adapter Pattern 161
8.1.3 Composite Design Pattern 162
8.1.4 Observer Pattern 164
8.2 Similarity Scoring Algorithm 166
8.2.1 From Coupling Tables to Matrices 168
8.2.2 An Example 171
8.3 Methodology for Detecting Design Pattern Structures 172
8.4 Case Study - Couplings and Design Patterns in JUnit 176
8.4.1 Analysis of the Adapter Pattern 178
8.4.2 Analysis of the Composite Pattern 185
8.4.3 Analysis of the Observer Pattern 186
8.4.4 Lessons Learned - New Approach 191
8.4.5 Discussion of Difference from Other Approaches 192
8.5 Summary 194
9 COUPLING-BASED FAULT MODEL 196
9.1 Conceptual Model for Correlations Among Relationships, Couplings, and Faults in Object-Oriented Systems 198
9.2 Coupling Levels and Coupling-based Fault Classification of Object-Oriented Software 201
9.3 Fault Index Computation for Couplings 207
9.4 Summary 208
10 CONCLUSION AND FUTURE WORK 211
10.1 Contributions 211
10.2 Future Work 215
10.2.1 Application of Coupling Model to Web Applications 215
10.2.2 Coupling-based Fault Analysis 216

10.2.3 Comprehensive Empirical Validation of Three Specific Problems 216
10.2.4 Extension of Design Pattern Detection 217
10.2.5 JCAT Enhancement 217
10.2.6 Coupling-based Reverse Engineering 217
10.2.7 Coupling-based Component Ranking 218
10.2.8 Coupling-based Testing 218
10.2.9 Coupling-based Analysis of Concurrent Software 219
LIST OF REFERENCES 220
A Unified Framework for Coupling - Definitions of Terms 232
B Sample Implementation of Design Patterns 239
B.1 Adapter Pattern Structure Java Sample Implementation 239
B.2 Composite Pattern Structure Java Sample Implementation 241
B.3 Observer Pattern Structure Java Sample Implementation 244

List of Tables

Table Page

2.1 **Types of Changes (I)** . . . 23

3.1 **Types of Connections** . . . 40

3.2 **Options for Counting Connections at the Attribute and Method Level** . . . 49

3.3 **Mathematical Properties of Coupling Measures** . . . 56

3.4 **Couplings in Object Adapter Pattern Structure** . . . 62

6.1 **Cycle-weight Ratio for Edges in** SCC $\{E, A, C, H, B, D, F\}$ . . . 108

6.2 **Cycle-weight Ratio for Edges in** SCC $\{E, A, C, H, B, D, F\} -$ $\{A \rightarrow C\}$ . . . 109

6.3 **Cycle-weight Ratio for Edges in** $SCC\{E, A, C, H, B, D, F\}$ **-**$\{A \rightarrow C, H \rightarrow B\}$ . . . 110

6.4 **Cycle-weight Ratio for Edges in** SCC $\{E, A, C, H, B, D, F\}$ **- All Edges Have the Same Weight** . . . 111

6.5 **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\}$ **in Figure 6.4** . . . 114

6.6 **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\}$ **-{C} in Figure 6.4** . . . 115

6.7 **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\}$ **-{C, B} in Figure 6.4** . . . 116

6.8 **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\}$ **in Figure 6.5** . . . 117

6.9 **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\} - \{C\}$ 118

6.10 **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\} - \{C, B\}$ 118

6.11 **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\} - \{A \rightarrow C\}$ . . . 121

6.12 **Cycle-weight Ratio for Edges in** $SCC\{E, A, C, H, B, D, F\}$ **-**$\{A \rightarrow C, C\}$ . . . 121
6.13 **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\}-\{A \rightarrow C, C\}$ . . . 122
6.14 **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\}-\{A \rightarrow C\}$ . . . 124
6.15 **Cycle-weight Ratio for Edges in** $SCC\{E, A, C, H, B, D, F\}$ **-**$\{A \rightarrow C, B\}$ . . . 124
6.16 **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\}-\{A \rightarrow C, B\}$ . . . 125
6.17 **Coupling Measures for Edges in** SCC $\{8, 9, 10, 11, 12, 13, 14, 15\}$ 128
6.18 **Different Weights for Edges in** SCC $\{8, 9, 10, 11, 12, 13, 14, 15\}$. 129
7.1 **Descriptive Statistics of JRGrep Application** . . . 153
7.2 **JRGrep Application Coupling Matrix** . . . 153
7.3 **JRGrep Application Coupling Analysis** . . . 154
7.4 **JRGrep Application Class Ranking Based On Change Sensitivity** . . . 154
7.5 **JRGrep Application Class Ranking Based On Change Impact** 155
7.6 **Maximum and Minimum Values of JRGrep Application Metrics** 155
8.1 **Couplings in Object Adapter Pattern Structure** . . . 162
8.2 **Coupling Matrix for Composite Pattern Structure** . . . 164
8.3 **Coupling Matrix for Standard Observer Pattern Structure** . 165
8.4 **Coupling Matrix for the Observer Pattern Variation 1** . . . 166
8.5 **Coupling Matrix for the Observer Pattern Variation 2** . . . 167
8.6 Package Level Base Type Couplings . . . 178
8.7 **Aliases for Class Names in *framework* and *swingui* Packages** 179
8.8 **Coupling Matrix for the *framework* package in JUnit** . . . 180
8.9 **Coupling Matrix for the *swingui* package in JUnit** . . . 181
8.10 **Couplings from *swingui* to *framework*** . . . 182
8.11 **Coupling *Type* Matrix for the *framework* Package in JUnit** . 183
8.12 **Coupling Types in Adapter Pattern Structure** . . . 184
8.13 **Coupling Types in Composite Pattern Structure** . . . 184

8.14 **Coupling Types in Standard Observer Pattern Structure** . . 184
8.15 **Coupling Type Matrix for Observer Pattern Variation 1** . . . 185
8.16 **Coupling Type Matrix for the Observer Pattern Variation 2** 185
8.17 **Coupling Matrix for Classes in *framework* and *swingui* packages** 189
9.1 **OO Coupling Levels and OO Faults - Example** 210

List of Figures

Figure		Page
1.1	Applications of the Coupling Model	5
3.1	A Metamodel for Object-Oriented Systems	41
4.1	Context Diagram of JCAT	79
4.2	The Main User Interface of JCAT.	80
4.3	Class Diagram of Package *coupling*	82
4.4	AST Nodes for Coupling Computation	83
4.5	Class Diagram of Package *query*	85
4.6	Table Schema for Class Category	87
4.7	Method Level Tables	89
4.8	Variable Level Tables	90
4.9	Variable Usage Level Tables	92
4.10	Method Call Level Tables	92
5.1	Application of the Coupling Model	94
5.2	An Example of a Dependency Cycle	95
6.1	Example of Method Call Overlap	102
6.2	Example Weighted Object Relation Diagram (WORD)	107
6.3	WORD - Node Weight is the **Sum** of Incoming Edge Weights (Algorithm 2, case 1)	113
6.4	WORD - Node Weight is the **Maximum** of Incoming Edge Weights (Algorithm 2, case 2)	114
6.5	WORD - Node Weight is **between the Maximum and Sum** of Incoming Edge Weights (Algorithm 2, case 2)	117
6.6	Finding Overall Test Order in a WORD	126
6.7	Compressed WORD	127
7.1	Change Impact Sets	135
7.2	Change Impact Analysis	136

7.3 Change Categories 137
7.4 Variable Level Tables 146
7.5 Variable Usage Level Tables 149
7.6 Method Call Level Tables 151
7.7 Class Diagram for JRGrep Application 152
7.8 Kiviat Diagram for FileSearchListener Metrics 156
8.1 UML Class Diagram of the Object Adapter Pattern 162
8.2 UML Class Diagram of the Class Adapter Pattern 162
8.3 UML Class Diagram of Composite Pattern 163
8.4 UML Class Diagram of the Observer Pattern 165
8.5 UML Class Diagram of a Variation of the Observer Pattern 166
8.6 UML Class Diagram of another Variation of the Observer Pattern . . 166
8.7 JUnit Package Diagram 177
9.1 Conceptual Model for Correlation Among Relationships, Couplings and Faults in Object-Oriented Systems 199
9.2 Conceptual Model for Relationships Between Faults and Failures . . . 200
9.3 Coupling-based Object-Oriented Fault Classification 203
9.4 IISD: Example of Indirect Inconsistent State Definition. 207

Abstract

COUPLING-BASED ANALYSIS OF OBJECT-ORIENTED SOFTWARE
Aynur Abdurazik, PhD
George Mason University, 2007
Dissertation Director: Jeff Offutt

Testing and maintenance of Object-Oriented (OO) software is expensive and difficult. Previous research has shown that complex relationships among OO software components are among the key factors that make testing and maintenance costly and challenging. Thus, measuring the relationships has become a prerequisite to develop efficient techniques for testing and maintenance.

Coupling analysis is a powerful technique for assessing relationships among software components. In coupling analysis, two components are coupled if any kind of connection or relationship exists between them. The coupling nature is categorized into different levels or *types*. Coupling analysis tries, by defining a theoretical model, to capture all the attributes of the relationships among components of a given program. It also quantifies the coupling levels by defining a set of measures. The theoretical model and the measurement set serve as a foundation for exercising complexity analysis on various problems that are related to the interaction among components.

This research presents a theoretical model of OO coupling, quantitative analysis techniques to measure coupling, engineering techniques to apply coupling to three specific and well-known testing and maintenance problems, and empirical evaluations

based on a tool that was developed as part of this research.

The coupling measures are validated theoretically and empirically. Theoretically, coupling measures are validated using a published unified coupling framework. Empirically, the measures are applied to three well known problems and the results are compared with published work in these areas.

The result is a collection of coupling measures that quantify basic connections for different high level relationships. These measures are useful in finding solutions to the three specific problems posed in this research. For two of the three problems, Class Integration and Test Order (CITO) and Design Pattern Detection (DPD), this research developed a simpler technique than previous research has arrived upon. For the third problem, Change Impact Analysis (CIA), the resulting impact set from using the proposed coupling measures was more complete than previous research has computed.

The importance of this work is in defining couplings in a more comprehensive way. Previous research only considered inheritance relationships. Considering all kinds of relationships is important, because it allows reasoning at different levels of abstractions with coupling measures.

Chapter 1: INTRODUCTION

Software engineering defines the collection of techniques that apply an engineering approach to the construction and support of software products [FP97]. Engineering disciplines apply methods that are based on models and theories. The scientific method includes stating a hypothesis, designing and running an experiment to test its truth, and analyzing the results. Underpinning the scientific method is measurement: measuring the variables to differentiate cases, measuring the changes in behavior, and measuring the causes and effects. Once the scientific method suggests the validity of a model or the truth of a theory, measurement is continuously used to apply the theory to practice. Measurements can be effective in making characteristics and relationships more visible, in assessing the magnitude of problems, and in fashioning a solution to problems [FP97].

Object-Oriented (OO) software construction uses objects to design applications and computer programs. An object-oriented (OO) software consists of components (objects) that interact with each other to carry out specified functionalities of a software system. Principles of OO software development support reuse of software components and easier development and maintenance through better modularity and data encapsulation [CCHJ94]. However, dynamic binding, inheritance, and polymorphism increase the complexity of the relationships in the OO software. This increased complexity of relationships has brought new challenges to integration, testing, and maintenance of the OO software system. Software researchers and practitioners have

addressed these problems by continually developing new techniques and tools, however, many of the techniques and tools are not based on rigorous measurements. As Fenton and Pfleeger say, "methodological improvement alone does not make an engineering discipline" [FP97]. Modeling and measuring the relationships among components have become necessary and essential activities in finding solutions to the emerged problems [FP97, LH93].

Coupling analysis is one of several techniques that model and measure the relationships among components in a software system. In coupling analysis, two components are coupled if any kind of connection or relationship exists between them. The coupling nature is often categorized into different levels or types. Coupling analysis tries, by defining a theoretical model, to capture all the attributes of the relationships among components of a given software program. It also quantifies the coupling levels by defining a set of measures. The theoretical model and the measurement set serve as a foundation for exercising complexity analysis on various problems that are related to the interaction among components.

Deciding the order in which components are integrated and tested, computing the impacts of changes to the system, and detecting design patterns in program source are well known problems that directly depend on the analysis of relationships among components in a system. Many studies have used coupling measurement to try to solve the class integration and test order problem [BLW01, KGH+95b, TD97, BLW03, BFL02] and change impact analysis [BWL99]. So far, there is no research that uses coupling measures to detect design patterns. However, the coupling measures in the these papers are not complete. A coupling analysis method should try to (1) capture as many characteristics of relationships as possible, (2) distinguish among

relationships not only according to the structure but also their level of abstraction, (3) identify associations among different levels of relationships, and(4) define a unit to measure the couplings. With these details, the coupling measures can be used to more accurately solve problems that emerged from relationships among OO components. If we do not precisely analyze and measure the relationships, the solutions will be problematic regardless of the method employed. Of the previous research, the most recent is by Briand et al. [BLW03]. They tried to address many of the problems in the earlier papers, however in their research on the graph-based class integration and test order problem, the number of distinct method calls from one class to another is not computed. Instead, only the existence of connection between two classes are considered. Moreover, research shows that the current set of coupling metrics do not fully capture all of the code-visible dependencies that are important for impact analysis [BWL99]. Researchers have tried find ways to detect patterns in source for their importance in program understanding and reusing design experiences. Design pattern detection is challenging for a number of reasons. A class can play multiple roles in a specific design pattern. Thus, when a system has large number of classes, a combinatorial explosion can occur in the detection process [TCSH06]. Furthermore, the list of design patterns is continuously expanding. Whether coupling measures be useful in finding improved solutions to design pattern detection has not been addressed.

1.1 The Problem

The current research on modeling and measuring the relationships among software components through coupling analysis is insufficient. Coupling measures are incomplete in their precision of definition and quantitative computation. In particular, current coupling measures do not reflect the differences in and the connections between design level relationships and implementation level connections. Hence, the way coupling is used to solve problems is not satisfactory.

1.2 Thesis Statement

Coupling measures that distinguish and connect design level relationships and implementation level connections can be used effectively to assess the magnitude of and to fashion a solution to testing and maintenance problems.

1.3 Introduction to the Three Focus Problems

The basic theoretical results in this research, coupling-based analysis, have been applied to three specific engineering problems. Using the theory to solve three well studied problems demonstrates the power of this theory. Subsequent chapters describe, in detail, how the theory is applied to the problems. The remainder of Chapter 1 introduces the concepts of the three sample problems: the class integration and test order problem, change impact analysis, and design pattern extraction using coupling measures. Figure 1.1 gives an overview of coupling-based analysis research and its applications. Coupling-based source code analysis (CBASCA) starts with parsing and analyzing the program source. Next, coupling measures are computed. Finally,

the coupling measures are applied to class integration and test order, change impact analysis, and design pattern detection. The measures can also be applied to other software engineering problems.

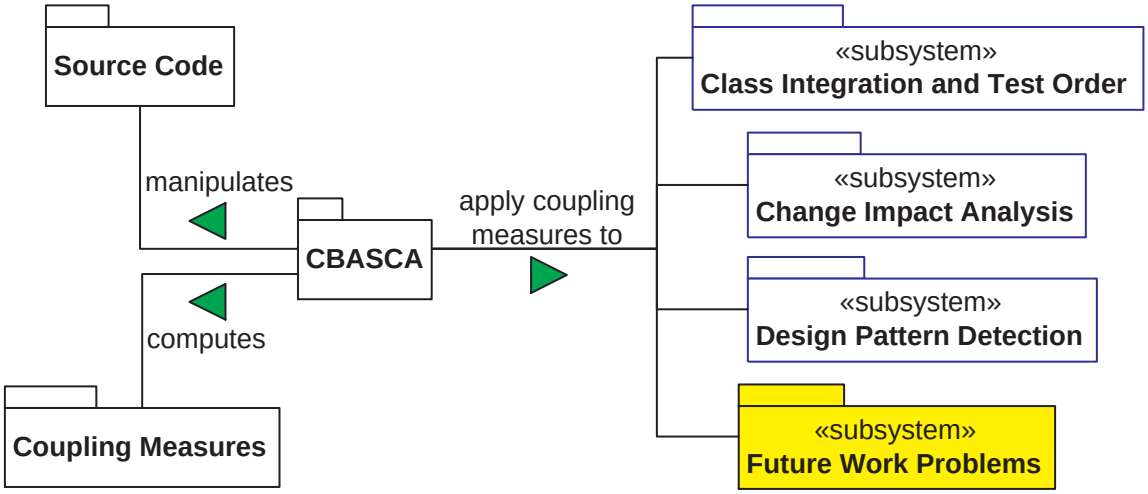

Figure 1.1: Applications of the Coupling Model

1.3.1 Class Integration and Test Order Problem

Object-oriented software development tries to achieve high quality by applying information hiding, abstraction, modularization, and reuse concepts. As a result, OO systems consist of classes that encapsulate concepts relative to some problem and domain [Mey97]. These classes are developed and integrated gradually to form the complete system.

One major problem in inter-class integration testing of object-oriented software is to determine the order in which classes are integrated and tested. The class integration and test order is important for several reasons. First, it determines the order

in which classes are integrated. Second, it impacts the use of test stubs and drivers for classes and the preparation of test cases. Third, it determines the order in which inter-class faults are detected.

The *class integration and test order (CITO)* problem is that of finding an optimal order to integrate and test individual classes can be . The CITO problem is important for several reasons [TD97]. First, it determines the order in which classes are integrated. Second, it impacts the use of test stubs[1]and drivers for classes and the preparation of test cases. Third, it determines the order in which inter-class faults are detected. A solution for the CITO problem should have two goals:

1. minimize the total number of test stubs
2. minimize the total complexity of test stubs

1.3.2 Change Impact Analysis

Change is an inherent and necessary part of a software. The importance of change is reflected in the distribution of software costs. Estimates show that 65-75% of total software costs are subsumed in maintenance activities [Som95]. Software systems change for two general reasons [War99]:

1. The environment in which a system operates is dynamic
2. Software development invariably introduces errors

As software systems become increasingly large and complex, it becomes more necessary to predict and control the effects of software changes. Experience over the

[1]Test stubs are pieces of software that have to be built in order to simulate parts of the software that are either not developed yet or have not yet been unit tested, but are needed to test classes that depend on them.

last three decades shows that making software changes without visibility into their effects can lead to poor effort estimates, delays in release schedules, degraded software design, unreliable software products, and premature retirement of software systems [BA96].

Change impact analysis (CIA) is the task of identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change. Change impact analysis addresses the problem by identifying the likely ripple effects of software changes and using this information to re-engineer the software system design [BA96]. A ripple effect is caused by making a small change to a system, which can affect many other parts [SMC74]. The purpose of impact analysis is to determine the scope of change requests as a basis for accurate resource planning and scheduling, and to confirm the cost/benefit justification.

There are two major technology areas for impact analysis: dependency analysis and traceability analysis [BA96]. These complementary areas approach impact analysis from quite different perspectives and each has advantages to enhancing the potential of identifying software impacts.

Dependency analysis involves examining detailed dependency relationships among entities (variables, logic, modules, etc.). It provides detailed evaluation of low-level dependencies in code but does little for software objects at other levels. Dependency analysis determines how different parts of a program interact, and how various parts require other parts in order to operate correctly. A *control dependency* governs how different routines or sets of instructions affect each other. A *data dependency* governs how different pieces of data affect each other.

Traceability analysis involves examining dependency relationships among different software objects. Although traceability covers many of the relationships among

artifacts that a software project library or repository might store, these relationships typically are not very detailed.

Coupling-based change impact analysis is in the category of *dependency analysis*.

1.3.3 Design Pattern Detection

A *design pattern* is a general repeatable solution to commonly occurring problems in software design. It is a description or template for how to solve a problem that can be used in many different scenarios. A design pattern is not a finished design that can be transformed directly into code. Object-oriented design patterns typically show relationships and interactions among classes or objects, without specifying the final application classes or objects that are involved. Algorithms are not regarded as design patterns, since they solve computational problems rather than design problems.

Design patterns can speed up the development process by allowing designers to use structures that have been successful in previous projects. Effective software design requires considering issues that may not become visible until later in the implementation after deployment, or when portions of the system are reused in other systems. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for programmers and design architects who are familiar with the patterns.

Often, software developers only understand how to apply certain software design techniques to certain problems. However, these techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that does not require specifics to be tied to a particular problem. In addition, patterns allow developers to communicate using well-known names for software interactions. Common design patterns can be improved over time, making them more robust than single-use designs.

Because most current software projects deal with evolving products consisting of a large number of components, their architectures can become complicated and cluttered. Design patterns can impose structure on the system through common abstractions. Consequently, identifying implemented design patterns could be useful for comprehending existing designs and provide information needed for refactoring [Vok06]. Thus, design pattern identification from source code can help improve software maintainability and reuse of designs.

Chapter 2: LITERATURE REVIEW

This section summarizes the related work. There is an increasing amount of research in coupling-based analysis and testing of software. The subsections that follow describe the contributions in detail. The first subsection describes the coupling types. The second subsection describes the coupling-based software metrics and precise measurement. The third subsection describes the coupling-based testing techniques. The fourth subsection describes the coupling-based software analysis techniques.

2.1 Coupling Background

Stevens et al. first introduced coupling in the context of structured development techniques, and defined coupling as "the measure of the strength of association established by a connection from one module to another" [SMC74]. Myers [Mye74] refined the concept of coupling by presenting well-defined, though informal, levels of coupling. Since his levels were neither precise nor prescriptive definitions, coupling could only be determined by hand, leaving room for subjective interpretations of the levels. Other researchers [TZ81,KH81,HB85,SB91] have used coupling levels or similar measures to evaluate the complexity of software design and relate this complexity to the number of software faults. El Amam et al. have established a similar correlation for predicting faulty classes in object-oriented software [EMM01]. Fenton and Melton [FM90] developed a measurement theory that provides a basis for defining software complexity and used hand-derived coupling measures to demonstrate their theory. They enhanced

previous work in coupling by incorporating the number of interconnections between modules into the measure and by considering the effects on coupling of return values and reference parameters as well as input parameters.

Historically, module coupling was used as an imprecise measure of software complexity. Jalote said that coupling is "an abstract concept and is as yet not quantifiable" [Jal91]. Offutt and Harrold [OHK93] extended previous work to reflect type abstraction, and quantified coupling by developing a general software metric system to automatically measure coupling. They offered precise definitions of the coupling levels so that they can be determined algorithmically, incorporated the notion of direction into the coupling levels, and accounted for different types of non-local variables as found in newer programming languages. To precisely define the coupling levels, they classified each call and return parameter by the way it is used in the module. They used the classification of uses as computation-uses (C-uses) and predicate-uses (P-uses) from data flow testing [FW88] and defined indirect-uses (I-uses). A C-use occurs whenever a variable (or parameter) is used in an assignment or output statement. A P-use occurs whenever a variable is used in a predicate statement. An I-use occurs whenever a variable is a C-use that affects some predicate in the module. They defined precise coupling levels between two modules A and B in the following list and indicated which of the coupling levels are bidirectional and which are commutative.

0. `Independent Coupling` (commutative) - `A` does not call `B` and `B` does not call `A`, and there are no common variable references or common references to external media between `A` and `B`.

1. `Call Coupling` (commutative) - `A` calls `B` or `B` calls `A` but there are no parameters, common variable references or common references to external media between `A` and `B`.

2. `Scalar Data Coupling` (bidirectional) - A scalar variable in `A` is passed as an actual parameter to `B` and it has a C-use but no P-use or I-use.

3. `Stamp Data Coupling` (bidirectional) - A record in `A` is passed as an actual parameter to `B` and it has a C-use but no P-use or I-use.

4. `Scalar Control Coupling` (bidirectional) - A scalar variable in `A` is passed as an actual parameter to `B` and it has a P-use.

5. `Stamp Control Coupling` (bidirectional) - A record in `A` is passed as an actual parameter to `B` and it has a P-use.

6. `Scalar Data/Control Coupling` (bidirectional) - A scalar variable in `A` is passed as an actual parameter to `B` and it has an I-use but no P-use.

7. `Stamp Data/Control Coupling` (bidirectional) - A record in `A` is passed as an actual parameter to `B` and it has an I-use but no P-use.

8. `External Coupling` (commutative) - `A` and `B` communicate through an external medium such as a file.

9. `Non-Local Coupling` (commutative) - `A` and `B` share references to the same non-local variable; a non-local variable is visible to a subset of the modules in the system.

10. `Global Coupling` (commutative) - `A` and `B` share reference to the same global variable; a global variable is visible to the entire system.

11. `Tramp Coupling` (bidirectional) - A formal parameter in `A` is passed to `B` as an actual parameter, `B` subsequently passes the corresponding formal parameter to another procedure without `B` having accessed or changed the variable.

Jin and Offutt later used couplings as a basis for integration testing [JO95, JO98]. They determined that the previous twelve-item ordered list contained more detail than was needed for integration testing, thus combined coupling into four unordered types:

- *Call coupling* is the same as in the previous levels.
- *Parameter coupling* refers to all parameter passing. This type combines scalar data coupling, stamp data coupling, scalar control coupling, stamp control, scalar data/control coupling, stamp data/control coupling and tramp coupling.
- *Shared data coupling* refers to procedures that both refer to the same data objects. This type combines nonlocal coupling and global coupling.
- *External device coupling* refers to procedures that both access the same external medium. This type is analogous to external coupling.

These were used to define formal integration testing criteria that required testing to proceed through couplings from data definitions to data uses.

Chidamber and Kemerer [CK92] developed six design metrics for OO systems and analytically evaluated the metrics against Weyuker's [Wey88] proposed set of measurement principles. They developed and implemented an automated data collection tool to collect an empirical sample of these metrics at two field sites in order to demonstrate their feasibility and suggested ways in which managers may use these metrics for process improvement.

Briand et al. [BDW99] provided a standardized terminology and formalism for expressing coupling measures in a consistent and operational manner. Based on their

review of existing frameworks and measures for coupling measurement, they provided a unified framework and classified all existing measures according to this framework. The proposed coupling framework has the following six criteria:

1. The *type of connection*, i.e., what constitutes coupling
2. The *locus of impact*, i.e., import or export coupling
3. *Granularity of the measure,* the domain of the measure and how to count coupling connections
4. *Stability of server*
5. *Direct or indirect coupling*
6. *Inheritance:* inheritance-based vs. noninheritance-based coupling, and how to account for polymorphism, and how to assign attributes and methods to classes

The framework by Briand et al. is useful for the comparison, evaluation, and definition of coupling measures in object-oriented systems. However, this framework is not complete. It did not differentiate noninheritance-based relationships. Different coupling measures represent different complexities of the relationships. Hence, the framework should reflect this criteria as well.

Arisholm [Ari02] proposed dynamic coupling measures quantifying the flow of messages between objects at runtime. His motivation for investigating dynamic coupling was that (1) static coupling is not up to the task of measuring the scope of a scenario; (2) static coupling analysis may include coupling that results from dead code; and (3) static coupling metrics cannot measure polymorphism. He defined 12 dynamic coupling measures and explored their relationship with change proneness of the classes. The result was that the dynamic coupling measures can indicate change

proneness in classes. The comparison between static coupling and dynamic coupling was left as future work.

2.2 Class Integration and Test Order

The class integration and test order problem has been addressed by several researchers and several solutions have been proposed. The solutions can be categorized into *graph-based* and *genetic algorithm-based* approaches. This section summarizes existing solutions and discusses their advantages and disadvantages.

In graph-based approaches, classes and their relationships in software are modeled as object relation diagrams (ORD) or test dependency graphs (TDG). An ORD or TDG is a directed graph $G(V, E)$ where V is a set of nodes representing classes and E is a set of edges representing the relationships among classes. The class integration and test order problem is to find an ordering of nodes in the graph so that the classes can be integrated and tested with minimum effort.

In most papers [BLW03, KGH+95a, TD97, TJJM00], the testing effort is estimated by counting the number of test stubs that need to be created during integration testing. This method assumes that all stubs are equally difficult to write. One recent paper tries to consider test stub complexity when estimating the testing effort [MCL03].

In the genetic algorithm-based approach [BFL02], inter-class coupling measurements and genetic algorithms are used in combination to assess the complexity of test stubs and to minimize complex cost functions.

Kung et al. [KGH+95a] were the first researchers to address the class test order problem and they showed that, when no dependency cycles are present among

classes, deriving an integration order is equivalent to performing a topological sorting of classes based on their dependency graph – a well known graph theory problem. In the presence of dependency cycles, they proposed a strategy of identifying strongly connected components (SCCs) and removing associations until no cycles remain. When there is more than one candidate for cycle breaking, Kung et al.'s approach chooses randomly. They mention that a possible alternative would involve the use of the complexity of the associations involved in cycles.

Tai and Daniels proposed a number of properties for inter-class test ordering [TD97]. They assumed that aggregation and inheritance relations do not form cycles, but association relations may. Tai and Daniels defined a *major* and a *minor* level for classes, and sorted classes according to these levels. First, classes are assigned major level numbers according to the inheritance and aggregation relationships only. There are no inheritance or aggregation edges between classes in the same major level. Because there are no cycles in the ORG when there are only inheritance and aggregation relationships, the classes can have a topological order, and major level numbers are assigned according to the reverse topological order of classes in the increasing order. Then, within each major level, minor-level numbers are assigned based on the association relationships only. At each major level, cycles may appear and must be broken in order to apply topological sorting. In this case, first, strongly connected components (SCCs) in a major level are identified, then each edge in a SCC is assigned a value, called *weight (e)*, which is defined as the sum of the number of incoming dependencies of the origin node of e and the number of outgoing dependencies of the target node of e. Edges with higher values are selected to break cycles. The hypothesis is that removing edges with higher values will break more cycles. However, Briand et al. [BLW03] showed this hypothesis is not always true. Another problem is that their

algorithm may break an association edge that crosses major levels but is not involved in any cycles [BLW03].

Le Traon et al. assigned weights to each node in the ORD, then removed the incoming edges of the node with maximum weight [TJJM00]. This process is repeated until no cycle remains in the ORD. To assign weights, they first used Tarjan's algorithm to identify strongly connected components. In each SCC, edges are partitioned into four classes: (1) *tree edges* lead from a node to an unvisited node, (2) *forward edges* are non tree-edges that go from a node to a descendent, (3) *frond edges* go from a node to an ancestor, and (4) *cross edges* are the remaining edges. The weight of a node is the sum of the number of incoming and outgoing frond edges.

Le Traon et al.'s approach is non-deterministic in two ways. First, different sets of edges can be labeled as *frond edges* depending on the different starting node. Second, the approach arbitrarily chooses a node when two or more nodes have the same weight. Thus, different runs of the algorithm result in different outcomes.

Briand et al. [BLW01,BLW03] proposed a graph-based strategy for ordering classes for testing that combines Tai and Daniels and Le Traon et al.'s approaches. They first used Tarjan's algorithm to identify strongly connected components (SCCs). Next, weights are assigned to *association* edges in the SCCs. The weight of an edge is the *estimated* number of cycles that the edge may be involved in. Let $G_i(V_i, E_i)$ be a SCC of graph $G(V, E)$ and $v_1, v_2 \in V_i, e \in E_i$, and $e = v_1 \rightarrow v_2$. The estimated weight of edge e is $weight(e) = (v_1)_{in} \times (v_2)_{out}$, where $(v_1)_{in}$ is the number of incoming dependencies of node v_1 and $(v_2)_{out}$ is the number of outgoing dependencies of node v_2. Then, the edge with the highest weight value is removed. These steps are repeated until no SCC remains.

Briand et al.'s approach has the advantage over Le Traon's approach of not breaking inheritance and aggregation edges and also the weight computation for edges is more precise than Tai and Daniels' approach.

Subsequently, Briand et al. [BFL02] used a genetic algorithm and coupling metric to try to break cycles by removing edges that will reduce the complexity of stub construction. A *genetic algorithm* is a heuristic that mimics the evolution of natural species in searching for the optimal solution to a problem. It is a search algorithm that locates optimal binary strings by processing an initially random population of strings using artificial mutation, crossover and selection operators, in an analogy with the process of natural selection [Gol89]. Briand et al. conclude that composition and inheritance relationships should never be removed since, according to their heuristic, removal of these edges would likely lead to complex stubs. The complexity of stub construction for parent classes is induced by the likely construction of stubs for most of the inherited member functions [BLW01]; moreover, inherited member functions must be tested in the new context of the derived class rather than the context of the parent class [HM92]. Their experiment showed that genetic algorithms can be used to obtain optimal results by using more complex cost functions and perform as well as graph-based algorithms under similar conditions.

Malloy et al. developed a *Class Ordering System* that is driven by a parameterized cost model [MCL03]. They used a strategy similar to Briand et al.'s graph-based approach [BLW03]. They defined six types of edges, *association*, *composition*, *dependency*, *inheritance*, *owned element*, and *polymorphic*. These edges are assigned weights of (2, 2, 20, 5, 20, 20) based on their estimation of the cost of stub construction for untested classes based on heuristics. For an ORD $G = (V, E)$, where V is a set of nodes representing classes and E is a set edges representing relationships

among classes, their cost model $C = \langle W, f(e), w(m_{x,y})\rangle$ is a 3-tuple where W is a set of weight assignments and $f(e)$ and $w(m_{x,y})$ are weight functions defined as:

$$W = \{w_1, w_2, w_3, w_4, w_5, w_6\} \tag{2.1}$$

$$f : E \rightarrow W \tag{2.2}$$

$$for\ a\ given\ x, y \in V, m_{x,y} = \{(x, y) \in E\} \tag{2.3}$$

$$W = w(m_{x,y}) = \sigma_{e \in m_{x,y}} f(e) \tag{2.4}$$

This cost model assigns values to the relationships among classes. When there is a cycle, the edge with the smallest weight is removed from the strongly connected component. When there is no cycle, the reverse topological sort of the nodes in the ORD is the order for integration testing.

To summarize, the existing graph-based approaches use high level, course grained estimates of test stub complexity. The GA approach must be run many times, greatly complicating the process.

2.3 Change Impact Analysis

Logical ripple effect analysis [YCM78] is defined as identifying program areas that require additional maintenance activity to ensure their compatibility with the original change. Yau et al. developed a technique for analyzing ripple effects for functional programs from both logical and performance perspectives [YCM78]. Yau et al. used error flow analysis to compute logical ripple effects. Error sources propagate across

module boundaries and are used to measure potential error propagation in the subsequent modules. All program variable definitions involved in an original change represent primary error sources from which inconsistencies can propagate to other program areas. Secondary error resources represent variables or control definitions implicated through the use of a primary error source. Identification of affected program areas is made by initially tracking each primary error source and its secondary error sources within the changed module to a point of exit. At each point of exit, a determination would be made as to which error sources propagate across module boundaries. Propagated error sources then became primary error sources within the subsequent modules. Tracing continues until no new secondary error sources are created. The ripple effect computation is carried out in two functional stages: lexical analysis and application of an algorithm for ripple effect computation.

Performance ripple effect analysis identifies modules whose performance may change when software is modified. Yau et al. [YCM78] identified eight mechanisms that may exist in large-scale programs by which changes in performance as a consequence of a software modification are propagated throughout the program: *parallel execution*, *shared resources*, *interprocess communication*, *called modules*, *shared data structures*, *sensitivity to the rate of input*, *execution priorities*, and *abstractions*. These mechanisms are linked to 13 performance attributes, which are associated with performance requirements. The performance ripple effect is analyzed by tracing the changes to performance requirements.

Yau et al. also proposed an expression to estimate the complexity of program modification and to evaluate various modifications. A programmer's effort required to perform a modification n_p on module M_j taking into account all its ripple effects

is estimated by the following expression:

$$G(Q_j, B_{jp}) + \sum_{M_i \in \psi_{jp}} \{D(Q_i) + F(Q_i, E_{ip}) + G(Q_i, E_{ip})\}, \tag{2.5}$$

where Q_i is the complexity of module M_i, $D(Q_i)$ is the amount of programmer's effort to understand M_i that is a function of Q_i, $G(Q_j, B_{jp})$ is the programmer's effort for making the modification n_p, $F(Q_i, E_{ip})$ and $G(Q_i, E_{ip})$ is the programmer's effort for examining E_{ip} and making the necessary change due to n_p's ripple effect in M_i.

Their research did not establish quantitative measures for the terms in equation 2.5.

Kung et al. [KGH+94] defined change types and provided methods to identify changes and their impacts. They formally modeled the impacts of class relationship changes, but not the impacts of variable and method related changes. Since this research was done in the early 1990s, some features of Java are not included. In particular, adding or deleting "import" statements were omitted.

Lee et al. developed an analysis technique for object-oriented software [LOA00]. The technique includes definitions for object-oriented dependency graphs, a set of algorithms that evaluate proposed changes on object-oriented software, a set of object-oriented change impact metrics to quantitatively evaluate the change impacts, and a proof of concept tool that computes the impacts of changes.

Briand et al. [BWL99] investigated the use of coupling measurements to identify classes likely to contain ripple effects when another class is being changed. Their study showed that aggregation and invocation coupling measures are related to a higher probability of changes. This indicates that these coupling measures should be good indicators of ripple effects and can be used in a decision model for ranking

classes according to their probability to contain ripple effects. Experimental results showed that their coupling-based model indicates class pairs with higher ripple effect probability. However, a substantial number of ripple effects were not covered by the selected highly coupled classes. Their conclusion is that it is very likely that the current set of coupling measures, as defined in the literature, does not fully capture all the code-visible dependencies that are important for impact analysis, e.g., inherited aggregation relationships. They suggest expanding the coupling measure set and building models derived not only from code, but all sorts of requirement and design artifacts, thus providing additional information for coupling measurement.

Ryder and Tip [RT01] transformed source code edits into a set of *atomic changes* , as shown in Table 2.1, and proposed breaking source code edits into unique sets of atomic changes. **CM** captures changes to a method body, including (i) adding a body to a previously abstract method, (ii) removing the body of a non-abstract method and making it abstract, and (iii) making any number of statement-level changes inside a method body. The **LC** category "abstracts" any kind of source code change that affects dynamic dispatch behavior. Some source code changes correspond to more than one atomic change. For example, the addition of an empty method may imply several atomic changes, of types **AM** and **LC**. Here, the **AM** change denotes the added method as a node in the call graph of *P*', and the **LC** change(s) specifies the change(s) in dynamic dispatch behavior caused by this method addition. LC changes can be caused by adding or deleting methods, and by adding or deleting inheritance relations.

Ryder and Tip ignored source code level changes that have no direct semantic impact apart from controlling visibility, including changes to access rights of classes, methods, and fields, addition/deletion of comments, and addition/deletion of import

Table 2.1: **Types of Changes (I)**

AC	Add an empty class
DC	Delete an empty class
AM	Add an empty method
DM	Delete an empty method
CM	Change body of a method
LC	Change virtual method lookup
AF	Add a field
DF	Delete a field

statements.

They have also developed a tool, Chianti, that analyzes changes to Java program and how they impact test cases [RST+04]. Chianti analyzes two versions of an application and decomposes their difference into a set of atomic changes. Change impacts are reported in terms of affected (regression or unit) tests whose execution behavior may have been modified by the applied changes. For each affected test, Chianti also determines a set of affecting changes that were responsible for the test's modified behavior. Isolating changes that induce the failure of one specific test from those changes that only affect other tests can be used as a debugging technique in situations where a test fails unexpectedly after a long editing session.

Tsantalis et al. [TCS05] proposed a probabilistic approach to estimate how prone an object-oriented design is to being changed by evaluating the probability that each class of the system will be affected when new functionality is added or when existing functionality is modified. When a system exhibits a high sensitivity to changes, the corresponding design quality is questionable. The extracted probabilities of change can be used to assist maintenance and to observe the evolution of stability through successive generations and identify a possible "saturation level" beyond which any

attempt to improve the design without major re-factoring is impossible. The proposed model has been evaluated on two multi-version open source projects. The process has been fully automated by a Java program, and statistical analysis shows improved correlation between the extracted probabilities and actual changes in each of the classes in comparison to a prediction model that relies simply on past data.

Wilkie and Kitchenham [WK00] empirically investigated the effects of class couplings on changes made to a commercial C++ application over a period of two and half years. They used the Chidamber and Kemerer CBO metric [CK92] and investigated whether classes with high CBO are more likely to be affected by ripple changes. This hypothesis was not proven true, but CBO was found to be an indicator of change-proneness in general. They also investigated whether classes that are affected by the same ripple change are coupled to at least one another. The conclusion was that CBO cannot account for all changes and other dependencies are needed to to be considered to explain the remaining ripple effects.

Arisholm [Ari01] proposed and validated a measurement framework for assessing the changeability of object-oriented software. Arisholm viewed changeability as a two-dimensional quality characteristic, related both to the *effort* to implement changes and to the resulting *quality* of the software after the changes. Arisholm defined *changeability* and proposed three alternative approaches for measuring changeability: *Structural Attribute Measurement (SAM)*, *Change Profile Measurement*, and *Benchmarking*. In his definitions, changeability can only be compared between two systems, and changeability decay can be compared between two successive versions of a software.

Chaumun et al. [CKKL02, CKK+00] computed change impacts among classes to assess the *changeability* of object-oriented software. They define changeability as

a program's ability to absorb a change. A system easily absorbs a change if the number of impacted components is low. They defined a *change impact model* at the *conceptual level* and mapped it onto the C++ language. Then, the change impact model is used to assess the changeability of software system. This approach was validated empirically by making one change to a telecommunications system. They defined and used these terms in their research: *changeability*, *change impact*, *the ability to absorb a change*, *using design metrics as indicators of changeability.*

They defined a change to a system as a modification to any component in the system. A component refers to either a class, a method, or a variable. A change in a class may affect other classes if other classes are *connected* to the changed class through some *links*. They defined four types of links: association(**S**), aggregation (**G**), inheritance (**H**), and invocation (**I**). The absence of an operator between two links, a special notation used in Boolean algebra, is used to mean an intersection. The "+" and "∼" operators are used to represent an union and a negation. Aggregation is defined as "a form of association that specifies a whole-part relationship between two classes." When methods defined in one class are being invoked by methods in other another class, this is referred to as invocation. For association, they used the definition in [BRJ98]. For the local impact of changes, they introduced a link called "local" (**L**).

They defined *impact* separately for each type of change. For example, the impact of a *method signature change* is the average number of impacted classes by a change to each method's signature. They indicate that this definition cannot be used for other changes. In their experiment, the software under experiment was parsed each time to analyze a change.

2.4 Design Pattern Detection

Before design patterns appeared in the literature, design pattern notions were described using clichés. Rich and Waters called "commonly used combinations of elements with familiar names" as clichés [RW88]. This project developed an intelligent assistant for building reusable and well structured software. This project included a tool called Recognizer, which analyzed source code in various languages and derived a representation in a form that could be compared to the clichés stored in a knowledge base. The Recognizer part of the Programmer's Apprentice was similar to today's automated design pattern detection techniques.

The first attempt to automatically detect design patterns was performed by Brown [Bro96], in which Smalltalk code was reverse-engineered in order to detect four well-known patterns from the catalog by Gamma et al. [GHJV01]. The algorithm was based on information retrieved from class hierarchies, association and aggregation relationships, as well as the messages exchanged between classes of the system.

Prechelt and Krämer [PK98] developed a system that could identify a number of design patterns that are present in C++ source code. OMT class diagrams representing the patterns were inspected to build Prolog rules aiding their recognition. As a result, such an approach required the definition of new Prolog rules when a novel design pattern had to be detected.

According to Wendehals [Wen03], a combination of static and dynamic analysis is desirable to efficiently detect design patterns. In terms of UML notations, this requires the analysis of class diagrams in order to recover the static information and the examination of sequence or collaboration diagrams for the dynamic information. Heuzeroth et al. [HHHL03] first applied static analysis to obtain a candidate set of

pattern instances and then performed dynamic analysis of this set. However, this approach heavily depends on the characteristics of each pattern: For every new pattern, one has to come up with a specific algorithm for computing the static candidates and then set up the rules that will enable the dynamic analysis. Consequently, this is prohibitive for the development of an extensible automated design pattern detection methodology.

To examine how useful a design pattern recovery tool could be in program understanding and maintenance, Antoniol et al. [ACPF01] developed a technique to identify structural patterns in a system. In the first stage metrics are used to identify possible pattern candidates. In the second stage, shortest path constraints are generated from the shortest paths between roles in the patterns. Finally, for some patterns where method calls are important, delegation constraints are generated. The above three-stage pattern recovery approach aims to reduce the exploration space. The final pattern instances are extracted based on structural information. Their technique has been tested on small to medium size public domain systems. As the authors also note, the main disadvantage of the approach is low precision (many false positives).

Balanyi and Ferenc [BF03] use the Columbus [Fro07] reverse engineering framework to extract an abstract semantic graph and DPML (Design Pattern Markup Language) to describe the characteristics of pattern roles. The pattern mining algorithm tries to match roles present in the DPML files with classes in the abstract semantic graphs. In this approach, the search space is reduced by filtering based on structural information. The technique has been tested on four medium to large size public domain projects. Their study revealed that the more the description of the patterns is simplified, the more false positives appear. Since the algorithm performs exact matching, this approach may not be able to identify modified pattern versions.

A different solution proposed by Costagliola et al. [CLD+05], uses a graphical format as an intermediate representation. Design patterns are expressed in terms of visual grammars and a design pattern library is built. A visual language parsing technique is used to detect patters in the system under study by simultaneously comparing the results of parsing with the existing library. The main advantage of this approach is that the process can be directly visualized; however, the approach has not been evaluated on real systems since the tool does not integrate with existing source-code to class-diagram extractors.

The methods described above are not able to detect modified versions of patterns that deviate from their standard representation. This poses a serious limitation on the applicability of these techniques to real software systems.

Bergenti and Poggi [BP00] developed a method in which UML diagrams are examined to propose modifications to the software architecture that would lead to design patterns. Automated detection of design patterns in the system is part of this process. The input to their tool is the UML design (class and collaboration diagrams) of the software system in XMI (XML Metadata Interchange) format. Both static and dynamic analysis is performed by exploiting a knowledge base consisting of Prolog rules that describe the main characteristics of the patterns to obtain the final set of pattern instances. New Prolog rules have to be composed to introduce new design patterns to the tool. Furthermore, no evaluation results for real software systems are presented in this study.

More recently, a method for detecting design patterns through so-called "fingerprinting" has been proposed by Gueheneuc et al. [GSZ04]. This approach reduces the search space by identifying classes playing certain roles in design motifs using metrics based on their external attributes. Actual pattern realizations are found with

structural matching in the next phase. The efficiency of such an algorithm depends strongly on the learning samples that compose the repository of design motif roles.

Albin-Amiot et al. [AACGJ01] developed a technique that claims to identify modified versions of design patterns. Using their pattern detection subsystem "PTIDEJ" the authors examine the problem as a constraint satisfaction problem. This problem is formulated by examining the pattern's abstract model and the source code under consideration. The set of the variables and the constraints for the variables are derived from the pattern's abstract model while the domain for the problem are the entities present in the source code of the examined system. The source code microarchitectures that are identical or similar to the microarchitecture defined by the design pattern are identified by a tool called PALM. The main drawback of the approach is that a new abstract model (for the constraint satisfaction problem) has to be embedded in the tool in order to achieve the detection of a new pattern.

Tonella and Antoniol [ACPF01] used concept analysis based on class relationships. No knowledge base of design pattern representations are used in their application. The design patterns present in a system are inferred directly from the system under study by finding recurrent groups of classes. The advantage of this approach is that it is easily extensible since new patterns can be easily discovered. One disadvantage of this approach is computational complexity, which is reduced by considering up to order 3 class-context. That means that class sequences of length up to 3 are considered to build a concept.

Smith and Stotts [SS] present a different approach to automated design pattern detection based on the notion of elemental design patterns. Elemental design patterns [SS02] are base concepts on which more complex design patterns are built. The main power of an approach based on the notion of elemental design patterns is the ability

to detect a design pattern after "refactorings" [FBB+99] have been applied to it. Such elemental design patterns are identified at a first level and then these findings are composed to identify actual design patterns. In order to represent relationships between objects directly, methods, and fields, a formal language called rho-calculus is used. The same language is used to formalize both the design patterns as well as the system under consideration. Next, an automated theorem prover is used to detect instances of patterns in the system. However, it is not clear which heuristic is used to combine the existing predicates in order to achieve this result. Obviously, the computational complexity of examining all the possible combinations, i.e., when no heuristic is applied, is prohibitive. The applicability of this technique is presented with an illustration of the steps required to detect the Decorator pattern in a small author-made system.

Vokác [Vok] tried to find a relation between the presence of specific design patterns in software and the number of defects. A reverse engineering tool called "Understand for C++" parses the source code and produces structural metadata, which is stored in a database. Patterns are then recovered through database queries [Vok06] that correspond to the structural signature of each pattern. Both the recall (few false negatives) and precision (few false positives) are quite good. A large commercial system is used to perform the validation of the technique. Recall has been evaluated on a random sample of classes using statistical analysis.

Tsantalis et al. [TCSH06] proposed a graph vertices similarity scoring based methodology to detect design patterns. To detect design patterns in source, they reduced the search space by constructing subsystems according to the inheritance hierarchy. Furthermore, they identified nine characteristics for patterns and used unweighted directed graphs to represent each characteristic. As a result, subsystems

and patterns can have a number of graph representations depending on how many characteristics they have. Finally, they used a graph similarity algorithm for each pair of subsystem and pattern graphs to detect patterns. They evaluated their method on three open-source projects, and several patterns were missed. They explain that those missed patterns lack certain pattern requirements to be considered as patterns, although the documents claim that they are patterns.

Chapter 3: COUPLING MODEL

"When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science."

Lord Kelvin

"You cannot control what you cannot measure."

De Marco, 1982

"You cannot control who you do not understand."

Mao

Software testing and maintenance are generally recognized to consume the majority of resources in many software organizations [BBC+]. Testing and maintenance of Object-Oriented (OO) software are costly and expensive [Bei90, AN91, HC90]. Previous research has shown that complex relationships among OO software components are among the critical factors that make testing and maintenance difficult and costly [KGH+95c]. Therefore, analyzing and measuring software component relationships has gained increasing importance [FP97, LH93].

Coupling analysis is a powerful tool for analyzing interactions among components in software. In coupling analysis, two components are coupled if a connection or relationship exists between them. The coupling nature is categorized into different levels. Coupling analysis tries to capture all the attributes of a software that are related to relationships among components by defining a theoretical model, and to quantify the coupling levels by defining a set of measures. The theoretical model and the measurement set serve as a foundation for exercising complexity analysis on various problems that are related to the interaction among components.

A large number of coupling measures have been defined for object-oriented (OO) systems. Previous research has revealed limitations of existing OO coupling measures in terms of their capability to assist certain testing and maintenance tasks [BDW99, BFL02, BWL99]. As these coupling measures can be considered to be defined mostly using low level connections, we investigate how high level relationship information can be used to enhance the coupling measures in the context of testing and maintenance activities. Such coupling measures could be used to further refine the solutions to testing and maintenance problems. This research presents a comprehensive methodology to perform coupling analysis of OO software. In particular, we explore different relationships and connections between OO components and define coupling measures accordingly. Moreover, we separate coupling measures for different relationships so that our measures are more comprehensive with respect to interaction characteristics of components.

This research is based on several concepts including UML relationships, high level relationship, low level connection, and message passing. UML relationships appear in the Unified Modeling Language (UML) class diagrams. High level relationships refer to the design level abstract relationships among OO classes. This research considers

UML relationships as high level relationships. Low level connections are at the source code level and they are reducible to mathematical formal notation. Message passing is a low level connection in which classes or objects send messages to each other to request a service.

Our approach relies on the following intuition. First, message passing can occur for different UML relationships. Second, separate coupling measures for different relationship makes the measures useful in more situations.

This chapter makes the following contributions:

- It presents both high level and low level connection analysis to identify the couplings that need to be measured for testing and maintenance activities; in particular, for class integration and test order, change impact analysis, and design pattern detection activities.

- It presents a set of coupling measures based on UML and other relationships. These measures include the measurement of return types and parameters of method invocations.

The rest of the chapter is organized as follows. Section 3.1 analyzes the object-oriented approach, including UML relationships and OO program coupling mechanisms. Section 3.2 formally defines coupling measures for this research, justifies the measures by identifying their mathematical properties, and presents a format that associates coupling measures to their base relationship types. Section 3.3 provides Java source code patterns for identified couplings.

3.1 Object-Oriented Approach

Understanding the object-oriented approach is the first step towards defining measures for that approach. The study of the object-oriented approach results in object-oriented concepts such as class, object, attributes, method, message passing, inheritance, and other relationships.

Terminology varies among object-oriented programming languages, however, all object-oriented languages share some common concepts. The characteristics of the object-oriented approach considered in this research include the concepts of *message passing* and *relationships*. Message passing is a common *communication mechanism* among objects. Whenever an object requests a service that another object provides, it sends a message to the other object. A relationship is a general term covering the specific types of *logical connections* among classes and objects. Message passing and relationships are different, yet they are associated with each other. Relationships are realized through message passing. This research uses UML concepts for relationships and Java language mechanisms for message passing. We study their characteristics in Sections 3.1.1 and 3.1.2.

3.1.1 Object-Oriented Relationship Types for Classes and Objects

In the context of object-oriented (OO) development, the Unified Modeling Language (UML) [Obj05] has become the de-facto standard language for analyzing and designing software systems. UML relationships are connections between model elements that add semantics to a model. UML relationships are used to define the structure between model elements. Examples of relationships include associations, dependencies, generalizations, realizations, and transitions. Associations indicate that instances of

one model element are connected to instances of another model element. Dependencies indicate that a change to one model element can affect another model element. Generalizations indicate that one model element is a specialization of another model element. Realizations indicate that one model element provides a specification that another model element implements. Transitions trigger state change and flow between activities.

Variations of these categories of relationships can be created by setting *properties* and using *keywords*. A *property* is a typed element that represents an attribute of a class. We have identified the following four types of variations of relationships for classes and objects that are relevant to our research [Gom00, RJB04].

1. *Association.* An association specifies a semantic relationship that can occur between typed instances. An association relationship is a structural relationship between two model elements that shows that objects of one classifier (actor, use case, class, interface, node, or component) connect and can navigate to objects of another classifier. An association relationship has three variations: association, aggregation, and composition.

    - An *association* is a relationship between two classifiers, such as classes or use cases, that describes the reasons for the relationship and the rules that govern the relationship.
    - An *aggregation* relationship depicts a classifier as a part of, or as subordinate to, another classifier. Aggregation represents "is part of" relationships. An engine is part of a plane, a package is part of a shipment, and an employee is part of a team. Aggregation is a specialization of association, specifying a whole-part relationship between two objects. In aggregation, the whole does not manage the life cycle of its parts.

- A *composition* relationship represents wholepart relationships and is a form of aggregation. A composition relationship specifies that the lifetime of the part classifier depends on the lifetime of the whole classifier. Composition is a stronger form of aggregation where the whole and parts have coincident lifetimes, and it is common for the whole to manage the life cycle of its parts.

2. A *dependency* relationship is a relationship in which changes to one model element (the supplier) impact another model element (the client). A dependency implies the semantics of the client are not complete without the supplier. There are several types of dependencies: *abstraction, substitution,* and *usage.*

    - An *abstraction* relationship is a dependency between model elements that represents the same concept at different levels of abstraction or from different viewpoints. Abstraction relationships can be added to a model in several diagrams, including use-case, class, and component diagrams. If an abstraction element has more than one client element, the supplier element maps into the set of client elements as a group. For example, an analysis-level class might be split into several design-level classes. The situation is similar if there is more than one supplier element. In summary, an *abstraction* relationship is between model elements at two different development stages. Therefore, we do not consider abstraction relationships in source code level coupling analysis.

    - A *usage* relationship is a type of dependency relationship in which one model element (the client) requires another model element (the supplier) for full implementation or operation. The model element that requires the presence of another model element is the client, and the model element

whose presence is required is the supplier.

- A *substitute* dependency declares that the source classifier may be substituted in a place where the target classifier has been declared as a type [Obj05]. The substitute dependency has similarities with "implements" but a substitute is not formally a specialization. This research does not consider the substitution relationship.

3. A *generalization* relationship is a relationship in which one model element (the child) is based on another model element (the parent). Generalization relationships are used in class, component, deployment, and use case diagrams. A generalization is a taxonomic relationship between classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier inherits the features of the more general classifier. A generalization relates to a specific classifier to a more general classifier, and is owned by the specific classifier.

4. A *realization* relationship is a relationship between two model elements, in which one model element (the client) realizes the behavior that the other model element (the supplier) specifies. Several clients can realize the behavior of a single supplier. Realization relationships can be used in class diagrams and component diagrams.

    - An *InterfaceRealization* or *Implementation* is a specialized realization relationship between a Classifier and an Interface. This relationship signifies that the realizing classifier conforms to the contract specified by the Interface. A classifier that implements an interface specifies instances that conform to the interface and to any of its ancestors. A classifier may

implement a number of interfaces.

3.1.2 Object-Oriented Connection Types Among Classes

The UML relationships described in Section 3.1.1 are abstract. They define the structure between model elements. They are visible at the design level but not directly visible at the implementation level. In other word, relationships are elements of a design model, but except for inheritance and realization, they are not part of a programming language. Table 3.1, which is adapted from Briand et al. [BDW99], gives a summary of connections that will occur among components using program constituents.

3.1.3 Metamodel

We briefly summarize the object-oriented approach by means of the metamodel in Figure 3.1. A metamodel is a precise definition of the constructs and rules needed for creating semantic models [Met07]. A model is an abstraction of phenomena in the real world, and a metamodel is yet another abstraction, highlighting properties of the model itself. A model is said to conform to its metamodel like a program conforms to the grammar of the programming language in which it is written.

We define any relationship that occurs because of exceptions to be a separate category because of the importance of exception handling. Exception handling deals with abnormal situation. The goal of exception handling mechanisms is to make programs robust and reliable. Incompletely or incorrectly handling of some abnormal situations causes failures in systems [PRT00]. If exceptions are not used correctly, they can slow down a program, as it takes memory and CPU time to create, throw, and catch exceptions. If they are overused, they make the code difficult to read and

Table 3.1: **Types of Connections**

#	Client item	Server item	Description
1	attribute a of a class c_i	class c_j, $c_j \neq c_i$	class c_j is the type of a
2	method m of a class c_i	class c_j, $c_j \neq c_i$	class c_j is the type of a parameter of m, or the return type of m
3	method m of a class c_i	class c_j, $c_j \neq c_i$	class c_j is the type of a local variable of m, or the return type of m
4	method m of a class c_i	class c_j, $c_j \neq c_i$	class c_j is the type of a parameter of a method invoked by m
5	method m of a class c_i	attribute a of a class c_j, $c_j \neq c_i$	m references a
5.1	class c_i	attribute a of a class c_j, $c_j \neq c_i$	c_i references a
6	method m of a class c_i	method m' of a class c_j, $c_j \neq c_i$	m invokes m'
6.1	class c_i	method m' of a class c_j, $c_j \neq c_i$	c_i invokes m'
7	class c_i	class c_j, $c_j \neq c_i$	high level relationships between classes, such as "uses" and "consists-of"
8	class c_i	class c_j, $c_j \neq c_i$	c_i extends c_j
9	class c_i	interface c_j, $c_j \neq c_i$	c_i implements c_j
10	class c_i	exception handler c_j, $c_j \neq c_i$	c_j handles an exception thrown by c_i

frustrating for the programmers using the API [Dos03].

3.2 Object-Oriented Coupling Measures

A large number of Object-Oriented coupling measures exist in the literature. Informal definitions of terminologies and metrics in coupling analysis brings about ambiguities

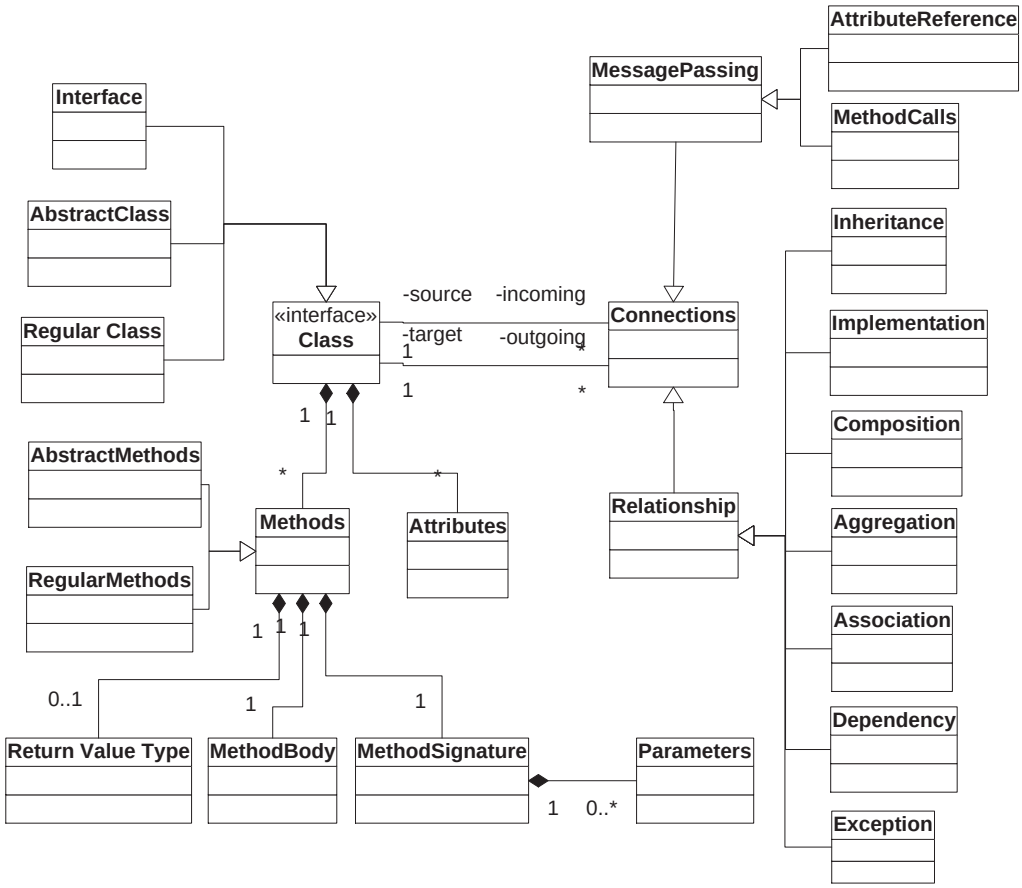

Figure 3.1: A Metamodel for Object-Oriented Systems

in interpreting their meaning, thus makes the coupling computation difficult. This also makes it difficult to understand how different coupling measures relate to one another. By using standardized terminologies and formalism, we can express coupling measures in a consistent and unambiguous manner. Considering how hard it is to determine how such measures relate to one another and for which application they can be used, Briand et al. provided a unified framework for OO coupling measurement. This framework comes with a standardized terminology and formalism so that measures can be expressed in a consistent and operational manner [BDW99].

Briand et al. investigated the properties of couplings and proposed five mathematical properties. The motivation behind defining mathematical properties is that a measure must be supported by some underlying theory of the internal quality attribute it measures. The five proposed coupling properties are defined as follows. Let $Coupling$ be a candidate measure for coupling of a class or an object-oriented system. Relationships capture the connections between classes the respective coupling measure is focused on. As the coupling measure can measure import or export coupling (or both), $OuterR(c)$ will denote the relevant set of relationships from or to class c (or both). Let $InterR(C) = \cup_{c \in C} OuterR(c)$ be the set of interclass relationships in system C. The five coupling properties are:

Coupling.1: Nonnegativity. The coupling [of a class c | of an object-oriented system C] is nonnegative:

$$[Coupling(c) \geq 0 \mid Coupling(C) \geq 0]$$

Coupling.2: Null value. The coupling [of a class c | of an object-oriented system C] is null if $[OuterR(c) \mid InterR(C)]$ is empty:

$$[OuterR(c) = \emptyset \Rightarrow Coupling(c) = 0$$

$$\mid InterR(C) = \emptyset \Rightarrow Coupling(C) = 0]$$

Coupling.3: Monotonicity. Let C be an object-oriented system and $c \in C$ be a class in C. Class c is modified to form a new class c' which is identical to c except that $OuterR(c) \subseteq OuterR(c')$, i.e., some relationships are added to c. Let C' be the object-oriented system which is identical to C except that class c

is replaced by class c'. Then

$$[Coupling(c) \leq Coupling(c') \mid Coupling(C) \leq Coupling(C')]$$

Coupling.4: Merging of classes. Let C be an object-oriented system, and $c_1, c_2 \in C$ two classes in C. Let c' be the class which is the union of c_1 and c_2. Let C' be the object-oriented system which is identical to C except that classes c_1 and c_2 are replaced by c'. Then

$$[Coupling(c_1) + Coupling(c_2) \geq Coupling(c') | \ Coupling(C) \geq Coupling(C')]$$

Coupling.5: Merging of unconnected classes. Let C be an object-oriented system, and $c_1, c_2 \in C$ two classes in C. Let c' be the class which is the union of c_1 and c_2. Let C' be the object-oriented system which is identical to C except that classes c_1 and c_2 are replaced by c'. If no relationships exist between classes c_1 and c_2 in C, then

$$[Coupling(c_1) + Coupling(c_2) = Coupling(c') | Coupling(C) = Coupling(C')]$$

The unified coupling framework proposed by Briand et al. [BDW99] has the following six criteria:

1. The *type of connection*, i.e., what constitutes coupling. Choosing a type of connection implies choosing the mechanism that constitutes coupling between two classes.

2. The *locus of impact*, i.e., import or export coupling

3. *Granularity of the measure:* the domain of the measure and how to count coupling connections

4. *Stability of server*

5. *Direct or indirect coupling*

6. *Inheritance:* inheritance-based vs. noninheritance-based coupling, and how to account for polymorphism, and how to assign attributes and methods to classes

Each of the above criterion is provided with a range of options [BDW99].

Many different decisions have to be made when defining a coupling measure – these decisions have to be made with respect to the goal of the measure and by defining an empirical model based on clearly stated hypotheses. The framework is applied to select existing measures or to derive new measures for a given measurement goal. The six criteria of the framework are necessary to consider when specifying a coupling measure, but they are not sufficient. In addition to the criteria, a coupling measure should be theoretically validated using the five mathematical properties and empirically validated using experiments [BDW99].

3.2.1 Derivation of Object-Oriented Coupling Measures

This section applies the unified OO coupling framework to derive new measures for testing and maintenance activities. The application is performed in the following two steps:

1. For each criterion of the framework, choose one or more of the available options basing each decision on the objective of measurement. The criteria must be dealt with in the introduced order because a decision made for one criterion can restrict the available options for subsequent criteria.

2. Choose the existing measures accordingly or, if none exists to match the decisions made, construct new coupling measures. Then use coupling properties to

guide the definition and theoretical validation of new measures.

In the context of applying this framework, the measurement goal must at least specify the development phase at which measurement is to take place and the underlying hypothesis which drives measurement.

The specific goals are to compute an optimal class integration and test order, to carry out change impact analysis, and to detect design patterns in the source code. The measurement takes place in the testing and maintenance phases of software development, and it is driven by the following hypotheses:

Hypothesis 1: OO relationships among classes in an OO system can be quantified through coupling measures.

Hypothesis 2: An impact set of a change can be computed by performing coupling analysis.

Hypothesis 3: Coupling measurement help optimize class integration and test order.

Hypothesis 4: Coupling measurement can be used in design pattern detection.

Hypothesis 5: Coupling measures can be automatically computed.

Hence, for each criterion of the framework, the following six decisions are made.

1. *Type of connection* (criterion 1): all types of connections (options 1 through 10 of Table 3.1).

   Justification: Hypothesis 1 alone needs all types of connections.

2. Locus of impact (criterion 2): Count both import and export coupling.

   Justification: High import coupling of a class indicates that the class strongly

depends on other classes and their methods and attributes [BDW99]. High export coupling of a class means that the class is used heavily by other classes and their methods and attributes. Both import and export couplings are therefore relevant in conjunction with the following activities:

- Class integration and test order computation: in order to compute the test stub complexity for each class, we must know about the clients that use the class as a server.
- Change impact analysis: to understand how likely a class or a method is to be changed, we must know about the services the class uses. Likewise, to understand how a change in a class or in a method affects others, we must know about the clients that use the class as a server.
- Design pattern detection: each pattern consists of several collaborating participants. A participant of a pattern is identified through how it collaborates with other participants. Therefore, it is essential to know both its import and export couplings.

3. Granularity (criterion 3):

   a) Required domain is "class".

      Justification: All hypotheses are tested at the class level.

   b) Count both individual and distinct connections (options C, D and F of Table 3.2).

      Justification: We need to measure distinct connections for CITO and individual connections for CIA. Any connection to a class should also be considered, because this is relavant in testing and maintenance. The more

often a method is invoked, the more effort is likely to be required to modify the invoking method when modification of the invoked method takes place. However, frequency of invocation a method do not make a difference in creating test stub for that method.

4. Stability of server (criterion 4): only count connections to unstable classes (option 1).

   Justification: In general, stable classes are not included in testing and maintenance. Hypothesis 1, 4, and 5 can be tested on stable classes. However, we do not consider them in this research.

5. Indirect or direct connections (criterion 5): Count both types of connections.

   Justification: there is no clear rationale for choosing one particular type of connection. All the available options should be investigated. In particular, considering indirect connections is important for connections through inheritance hierarchies. If class A inherits from B and B inherits from C, then A is coupled with C through inheritance. Not considering this can result in incorrect analysis.

6. *Inheritance (criterion 6):*
   a) Count both inheritance-based and noninheritance-based coupling and distinguish between these types of couplings (option III). Furthermore, we classify noninheritance-based couplings into the following categories:

      (2) Interface implementation-based

      (3) Abstract class implementation-based

      (4) Composition-based

      (5) Aggregation-based

(6) Association-based

(7) Dependency-based

(8) Exception-based

Justification: We do not know whether inheritance-based or noninheritance-based coupling is more important. Therefore, we need to measure all relationship types of coupling separately to investigate their relative importance.

b) Account for polymorphism.

Justification: For Hypothesis 2, any method that is used by a class may give rise to a modification to the class. For Hypothesis 3, any method that is used by a class should be included in the test stub. Therefore, we must include all methods that can be possibly invoked through polymorphism and dynamic binding.

c) Only methods implemented in a class contribute to the coupling of the class. Justification: We must choose this option because we count inheritance-based coupling separately.

We have chosen all connection types. However, some connection types reflect the relationships. For example, connection type 1 could implement an aggregation or composition relationship. Connection types 2, 3, and 4 are in fact usage dependency relationships. Connection types 7 through 10 also reflect higher level relationships. Therefore, connection types 1-4 and 7-10 can be combined with criterion 6. This means that we measure connection types 5 and 6 for eight different relationships. Connection types 5 and 5.1 measures the number of attributes of class c_j referenced in c_i, either in a method of c_i or at class level. We added connection type 5.1 to

Table 3.2: **Options for Counting Connections at the Attribute and Method Level**

#	Description	Import Coupling Example	Output Coupling Example
A)	count individual connections	for each method, the number of references to attributes	for each attribute the number of references to the attribute
B)	count number of distinct items at the other end of the connections	for each method, the number of attributes referenced	for each attribute the number of methods that reference the attribute
C)	add up the number of connections counted as in A) for each method or attribute of the class	the total number of attribute references by methods in the class	the total number of references to attributes of the class
D)	add up the number of connections counted as in B) for each method or attribute of the class	add up the number of attributes referenced by each methods of the class	add up or for each attribute of the class: the number of methods that reference the attribute
E)	count number of distinct items at the end of connections starting from or ending in methods or attributes of the class	the number of attributes referenced by the methods of the class	the number of methods referencing attributes of the class
F)	for each class c, count the number of other classes to which there is at least one connection	the number of classes which have an attribute that is referenced by a method of class c	the number of classes which have a method that reference an attribute of class c

the original table because Java allows programmers to reference an attribute in the scope of a class. Connection types 6 and 6.1 measure the number of methods of class c_j invoked in c_i, either in a method of c_i or at class level. Connection type 6.1 was added because Java allows programmers to invoke a method in the scope

of a class. Method invocation needs further discussion. There are four different types of method calls: (1) method calls with parameters and a return value, (2) method calls with parameters and without a return value, (3)method calls without parameters and with a return value, (4) method calls without parameters and without a return value. Each have different impacts in computing testing effort and change impact. To differentiate different method invocations, we measure the following two items in addition to counting number of method invocations: (1) the total number of return values that class c_i receives from class c_j (through method invocation), and (2) the total number of parameters that are sent from class c_i to c_j (through method invocation).

Section 3.2.2 formally defines these coupling measures.

3.2.2 Formal Definition of Coupling Measures

This section introduces the concept of *coupling base type (CBT)* to represent the abstract relationships in coupling measures.

DEFINITION 1 (Coupling Base Types).

> *Let CBT be a set that includes coupling base types. CBT =* {*inheritance, abstract class implementation, interface implementation, composition, aggregation, association, dependency, exception*}.

Each coupling base type is formally defined using the definitions in Appendix A, and assigned values to be differentiated in computation. The values do not indicate a quantitative judgement on the base types.

1. *Inheritance Coupling Base (InhrCB)* for a coupling measure between classes c_i and c_j is defined as follows:

Let $c_i, c_j \in UC$.

$$InhrCB(c_i, c_j) = \left\{ \begin{array}{cc} 256 & (c_j \in Parents(c_i)) \wedge (c_j \in RC) \\ 0 & c_j \notin Parents(c_i) \end{array} \right\}$$

2. *Abstract Class Implementation Coupling Base (AbsCB)* for a coupling measure between classes c_i and c_j is defined as follows:

   Let $c_i, c_j \in UC$.

$$AbsCB(c_i, c_j) = \left\{ \begin{array}{cc} 128 & (c_j \in Parents(c_i)) \wedge (c_j \in AC) \\ 0 & c_j \notin Parents(c_i) \end{array} \right\}$$

3. *Interface Implementation Coupling Base (IfimCB)* for a coupling measure between classes c_i and c_j is defined as follows:

   Let $c_i, c_j \in UC$.

$$IfimCB(c_i, c_j) = \left\{ \begin{array}{cc} 64 & (c_j \in Parents(c_i)) \wedge (c_j \in IC) \\ 0 & c_j \notin Parents(c_i) \end{array} \right\}$$

4. *Composition Coupling Base (CompCB)* for a coupling measure between classes c_i and c_j is defined as follows:

   Let $c_i, c_j \in UC$.

$$CompCB(c_i, c_j) = \left\{ \begin{array}{cc} 32 & c_j \in Compositions(c_i) \\ 0 & c_j \notin Compositions(c_i) \end{array} \right\}$$

5. *Aggregation Coupling Base (AggrCB)* for a coupling measure between classes c_i and c_j is defined as follows:

   Let $c_i, c_j \in UC$.

$$AggrCB(c_i, c_j) = \left\{ \begin{array}{ll} 16 & c_j \in Aggregations(c_i) \\ 0 & c_j \notin Aggregations(c_i) \end{array} \right\}$$

6. *Exception Coupling Base (ExcpCB)* for a coupling measure between an exception throwing class c_i and an exception handling class c_j is defined as follows:

   Let $c_i, c_j \in UC$.

$$ExcpCB(c_i, c_j) = \left\{ \begin{array}{ll} 8 & c_j \in Exceptions(c_i) \\ 0 & c_j \notin Exceptions(c_i) \end{array} \right\}$$

7. *Association Coupling Base (AssoCB)* for a coupling measure between classes c_i and c_j is defined as follows:

   Let $c_i, c_j \in UC$.

$$AssoCB(c_i, c_j) = \left\{ \begin{array}{ll} 4 & c_j \in Associations(c_i) \\ 0 & c_j \notin Associations(c_i) \end{array} \right\}$$

8. *Dependency Coupling Base (DpdnCB)* for a coupling measure between classes c_i and c_j is defined as follows:

   Let $c_i, c_j \in UC$.

$$DpdnCB(c_i, c_j) = \left\{ \begin{array}{ll} 2 & c_j \in Dependencies(c_i) \\ 0 & c_j \notin Dependencies(c_i) \end{array} \right\}$$

For each coupling base type, CBT_i, we define sets, multisets, and coupling measures. A multiset can be formally defined as a pair (A, m) where A is some set and $m : A \rightarrow N$ is a function from A to the set $N = \{1, 2, 3, ...\}$ of positive natural numbers. The set A is called the underlying set of elements. For each a in A the

multiplicity (that is, number of occurrences) of a is the number $m(a)$. The concept of a multiset is a generalization of the concept of a set. A multiset is a set if the multiplicity of every element is one [Sta97]. Many systems have been designed to support multisets in their data model (whether relational or object-oriented). The use of this type of data structure is motivated by its ability to manage quantities [Roc]. In the context of this research, sets are used for the distinct counting option and multisets are used for the individual counting option. The following definitions differentiate sets and multisets by using subscripts $_{\mathcal{D}}$ and $_{\mathcal{T}}$. $_{\mathcal{D}}$ stands for *distinct* and is used with sets, and $_{\mathcal{T}}$ stands for *total* and is used with multisets.

DEFINITION 2 (Attribute Reference Coupling).

Let $c_i, c_j \in C$. The set of attributes of c_j that are referenced by c_i is denoted by $\mathcal{A}_{\mathcal{D}}$ and is formally defined as

$$\mathcal{A}_{\mathcal{D}} = \{a|a \in A_I(c_j) \wedge a \in AR(m) \wedge m \in M_I(c_i)\} \cup \{a|a \in A_I(c_j) \wedge a \in AR(c_i)\} \tag{3.1}$$

The multiset of attributes of c_j that are referenced by c_i is denoted by $\mathcal{A}_{\mathcal{T}}$ and is formally defined as

$$\mathcal{A}_{\mathcal{T}} = [[a|a \in A_I(c_j) \wedge a \in AR(m) \wedge m \in M_I(c_i)]] \uplus [[a|a \in A_I(c_j) \wedge a \in AR(c_i)]] \tag{3.2}$$

Distinct attribute reference coupling between classes c_i and c_j is denoted by V_d,

$$V_d = |\mathcal{A}_{\mathcal{D}}|$$

and *total attribute reference coupling* between classes c_i and c_j is denoted

by V_t,

$$V_t = |\mathcal{A}_\mathcal{T}|$$

DEFINITION 3 (Method Invocation Coupling).

The set of methods of c_j that are invoked by c_i is denoted by $\mathcal{M}_\mathcal{D}$ and is formally defined as

$$\mathcal{M}_\mathcal{D} = \{m | m \in (SIM(c_i, c_j) \cup PIM(c_i, c_j)\} \tag{3.3}$$

The multiset of methods of c_j that are invoked by c_i is denoted by $\mathcal{M}_\mathcal{T}$ and is formally defined as

$$\mathcal{M}_\mathcal{T} = \{m | m \in (SIM(c_i, c_j) \cup PIM(c_i, c_j)\} \tag{3.4}$$

Distinct method invocation coupling between classes c_i and c_j is denoted by M_d.

$$M_d = |\mathcal{M}_\mathcal{D}|$$

Total method invocation coupling between classes c_i and c_j is denoted by M_t.

$$M_t = |\mathcal{M}_\mathcal{T}|$$

DEFINITION 4 (Method Return Value Coupling).

The set of returned values by methods in $\mathcal{M}_\mathcal{D}$ is denoted by $\mathcal{RV}_\mathcal{D}$ and is formally defined as

$$\mathcal{RV}_\mathcal{D} = \{rv | rv \in RV(m) \wedge m \in \mathcal{M}_\mathcal{D}\} \tag{3.5}$$

The multiset of returned values by methods in $\mathcal{M}_\mathcal{T}$ is denoted by $\mathcal{RV}_\mathcal{T}$

and is formally defined as

$$\mathcal{RV}_{\mathcal{T}} = \{rv | rv \in RV(m) \wedge m \in \mathcal{M}_{\mathcal{T}}\} \tag{3.6}$$

Distinct return value coupling between classes c_i and c_j is denoted by R_d.

$$R_d = |\mathcal{RV}_{\mathcal{D}}|$$

Total return value coupling between classes c_i and c_j is denoted by R_t.

$$R_t = |\mathcal{RV}_{\mathcal{T}}|$$

DEFINITION 5 (Method Parameter Coupling).

The set of parameters of methods in $\mathcal{M}$ is denoted by $\mathcal{P}_{\mathcal{D}}$ and is formally defined as

$$\mathcal{P}_{\mathcal{D}} = \{p | p \in Par(m)) \wedge m \in \mathcal{M}_{\mathcal{D}}\} \tag{3.7}$$

The multiset of parameters of methods in $\mathcal{M}$ is denoted by $\mathcal{P}_{\mathcal{T}}$ and is formally defined as

$$\mathcal{P}_{\mathcal{T}} = \{p | p \in Par(m)) \wedge m \in \mathcal{M}_{\mathcal{T}}\} \tag{3.8}$$

Distinct method parameter coupling between classes c_i and c_j is denoted by P_d.

$$P_d = |\mathcal{P}_{\mathcal{D}}|$$

Total method parameter coupling between classes c_i and c_j is denoted by P_t.

$$P_t = |\mathcal{P}_{\mathcal{T}}|$$

Analysis of the measures with respect to Briand's mathematical properties of couplings [BDW99] shows that measures with individual counts do not violate any properties, but measures with distinct counts violate property 5, merging of unconnected classes. This is summarized in Table 3.3. Measures with distinct counts are necessary in CITO problem analysis because only distinct elements used by other classes are considered in constructing a stub for a class. Therefore, we keep these measures and will consider the validity of those mathematical properties in the future.

Table 3.3: **Mathematical Properties of Coupling Measures**

Measure	**Nonnegativity**	**Null value**	**Monotonicity**	**Merging of Classes**	**Merging of Unconnected Classes**
V_d	✓	✓	✓	✓	
V_t	✓	✓	✓	✓	✓
M_d	✓	✓	✓	✓	
M_t	✓	✓	✓	✓	✓
R_d	✓	✓	✓	✓	
R_t	✓	✓	✓	✓	✓
P_d	✓	✓	✓	✓	
P_t	✓	✓	✓	✓	✓

3.2.3 A Unified Representation of Coupling Measures

We have now defined coupling base types and measures for each base type. The formal definition of a coupling between two classes c_i and c_j in a system is a tuple:

$$CP(c_i, c_j) = \langle CBT, V, M, R, P \rangle$$

where CBT is a finite set of coupling base types, V represents the number of

public variables declared in c_j that are directly used by c_i, M represents the number of public methods of c_j that are called by c_i, R represents the number of return types that appear in M, and P represents the number of parameters that appear in M.

To keep the coupling base type and the corresponding measures connected, we use matrix and a dot notation in our coupling representation. If a system has n unstable classes, we construct an $n \times n$ adjacency coupling matrix, where n is the number of unstable classes. A cell entry of the matrix, cp_{ij}, represents the couplings between classes c_i and c_j. This coupling is an import coupling for c_i and an export coupling for c_j.

The following equation uses a "dot" notation to represent a *coupling measure (CM)* for couplings between two classes c_i and c_j:

$$CM(c_i, c_j) = CBT.V.M.R.P \tag{3.9}$$

where c_i and c_j represent two classes that are coupled together, c_i being a client and c_j a server. CBT is a coupling base type indicator with the values

$$CBT = \begin{cases} 256 & \text{when coupling is based on inheritance} \\ 128 & \text{when coupling is based on abstract class implementation} \\ 64 & \text{when coupling is based on interface (implementation)} \\ 32 & \text{when coupling is based on composition} \\ 16 & \text{when coupling is based on aggregation} \\ 8 & \text{when coupling is based on exception} \\ 4 & \text{when coupling is based on association} \\ 2 & \text{when coupling is based on dependency} \end{cases} \tag{3.10}$$

Equation 3.10 assigns different values to CBT to distinguish different coupling types in our measure so that they can be analyzed and used methodically when needed. The dot notation is used to indicate that the five measures are independent

but related. For example, if CBT is an aggregation, the V and M measures indicate the number of attribute references and number of method calls under the aggregation relationship. Furthermore, the R and P measures indicate the number of return types and the number of parameters under the method calls in this relationship. When two classes are connected through more than one base coupling types, the overall coupling measure is the sum of the individual coupling measures.

After all couplings are measured in the form of equation 3.10, the total incoming coupling of a class c from k client classes $(d_1...d_k)$ is denoted as $cm_{c_{in}}$, and is computed as one measure as follows:

$$\begin{aligned} cm_{c_{in}} &= \{\sum_{i=1}^{k} CM(d_i, c) \mid d_i, c \in C\} \\ &= \sum_{i=1}^{k} CBT_{d_i,c}. \sum_{i=1}^{k} V_{d_i,c}. \sum_{i=1}^{k} M_{d_i,c}. \sum_{i=1}^{k} R_{d_i,c}. \sum_{i=1}^{k} P_{d_i,c} \\ &= CBT_{in}.V_{in}.M_{in}.R_{in}.P_{in} \end{aligned} \tag{3.11}$$

and the set contains all incoming coupling base types is denoted as $R_{c_{in}}$, and is computed as follows:

$$R_{c_{in}} = \{CBT_{d_i,c} \mid d_i, c \in C, i = 1...k\} \tag{3.12}$$

The total outgoing coupling of a class c to k server classes $(d_1...d_k)$ is denoted as

$cm_{c_{out}}$, and is computed as one measure as follows:

$$\begin{aligned} cm_{c_{out}} &= \{\sum_{i=1}^{k} CM(c, d_i) \mid c, d_i \in C\} \\ &= \sum_{i=1}^{k} CBT_{c,d_i}. \sum_{i=1}^{k} V_{c,d_i}. \sum_{i=1}^{k} M_{c,d_i}. \sum_{i=1}^{k} R_{c,d_i}. \sum_{i=1}^{k} P_{c,d_i} \\ &= CBT_{out}.V_{out}.M_{out}.R_{out}.P_{out} \end{aligned} \tag{3.13}$$

and the set contains all outgoing coupling base types is denoted as $R_{c_{out}}$, and is computed as follows:

$$R_{c_{out}} = \{CBT_{c,d_i} \mid c, d_i \in C, i = 1...k\} \tag{3.14}$$

The total coupling base type set for class c is denoted as R_c,

$$R_c = R_{c_{in}} \cup R_{c_{out}} \tag{3.15}$$

Finally, the set of total incoming coupling base types, R_{in}, of the system is equal to the set of total outgoing coupling base types, R_{out}, and denoted as total coupling base type set R.

$$R_{in} = \bigcup_{j=1}^{n} \{CBT_{d_i,c_j} \mid c_j, d_i \in C, i = 1...k\} \tag{3.16}$$

$$R_{out} = \bigcup_{j=1}^{n} \{CBT_{c_j,d_i} \mid c_j, d_i \in C, i = 1...k\} \tag{3.17}$$

$$R = R_{in} = R_{out} \tag{3.18}$$

Next, we will explain how we use this form to quantitatively represent each coupling type through examples.

3.2.4 A Simple Example

The following is a sample implementation of the Adapter pattern from the "Gang of Four" book [GHJV01].

```
// Adapter pattern -- Structural example

  class Client
  {
    public static void main(String args[])
    {
      // Create adapter and place a request
      Target target = new Adapter();
      target.request();
    }
  }

  // "Target"

  public interface Target
  {
    public void request();
  }

  // "Adapter"

  class Adapter implements Target
```

```
{
  private Adaptee adaptee = new Adaptee();

  public void request()
  {
    // Possibly do some other work
    // and then call specificRequest()
    adaptee.specificRequest();
  }
}

// "Adaptee"

class Adaptee
{
  public void specificRequest()
  {
    System.out.println("Called specificRequest().");
  }
}
```

```
Output from running Client class:
Called specificRequest().
```

Table 3.4 shows the couplings for each pair of classes. For example, this table indicates that there is an interface implementation based coupling from Adapter to Target, i.e., Adapter implements the Target interface. Furthermore, it indicates that

Table 3.4: **Couplings in Object Adapter Pattern Structure**

Classes	Client	Target	Adapter	Adaptee	$R_{c_{out}}$
Client		4.0.1.0.0	4.0.1.1.0		$\{4\}$
Target					$\{\}$
Adapter		64.0.1.0.0		32.0.2.1.0	$\{64, 32\}$
Adaptee					$\{\}$
$R_{c_{in}}$	$\{\}$	$\{4, 64\}$	$\{4\}$	$\{32\}$	$\{4, 32, 64\}$

within this relationship, there is a method invocation with no return values and parameters.

3.3 Java Source Code Patterns for Object-Oriented Couplings

We chose Java programs for our source code coupling analysis. Guéhéneuc and Albin-Amiot noticed that there is a discontinuity between object-oriented modeling and programming languages, and provided definitions and Java code patterns for *association, aggregation*, and *composition* relationships [GAA04]. We used the code patterns from their paper to identify association, aggregation, and composition coupling base types. The code patterns for other coupling base types are based on our programming knowledge and experience. The following subsections give details of code patterns for each coupling base types.

3.3.1 Java Source Code Pattern for Association Coupling

Association Coupling refers to couplings that occur between two classes through message passing. The following sample code contains the patterns of *Association Coupling*. There are two classes A and B. Class A has two methods, method1() and method3(); class B has one method, method2(). The first parameter coupling between class A and B occurs in A.method1(B b). Here a reference to a variable of type B is passed to method1(B b) and the method2() of class B is invoked. A second parameter coupling occurs in A.method3(). Here a local variable of type B is defined in method3(), then a method of class B is invoked through this local variable.

```
public class A {
   public void method1( B b ) {
      b.method2();
   }
   public void method3() {
      B anotherB = new B():
      anotherB.method2();
   }
}

public class B {
   public void method2() {
      ...
   }
}
```

In summary, association coupling occurs in two situations. The first is when a parameter is passed to an instance of a class, and then the receiver class invokes a method of the parameter. The second is when a class is instantiated and used inside the methods of a class.

3.3.2 Java Source Code Pattern for Aggregation Coupling

Aggregation coupling occurs between classes A and B when the definition of class A contains instances of class B, and A does not manage the lifecycle of the instances of B.

```
public class A {
   private B b; //aggregation
   public A( B b ) {
      this.b = b;
   }
   public void method1() {
      this.b.method2();
   }
}

public class B {
   public void method2() {
      ...
   }
}
```

The following classes also represent a pattern of aggregation coupling:

```
public class A {
   private List listOfBs;
   public void method3() {
      ((B) listOfBs.get(0)).method2();
   }
}

public class B {
   public void method2() {
      ...
   }
}
```

Aggregation coupling happens in two situations. First, when a class is the type of a class variable of another class. Second, when a container variable of a class has elements that have the type of another class.

3.3.3 Java Source Code Pattern for Composition Coupling

Composition coupling occurs between classes A and B when the definition of class A contains instances of class B, and A manages the lifecycle of the instances of B.

```
public class A {
   private B b = new B();
   public void method1(){
      this.b.method2();
   }
}
```

```
public class B {
   public void method2() {
      ...
   }
}
```

Composition coupling happens in two situations. The first is when a class is the type of a class variable of another class, the class variable is instantiated through the *new* operator, and the methods of the class used as the type are called. The second is when a container variable of a class has elements that are the type of another class, and the methods of the class used as the type are called. The following is an example of the second case.

```
public class A {
   private List listOfBs;
   private void init() {
      for (int i = 1; i < 10; i++)
        listOfBs.add(new B());
   }
   public void method3() {
      ((B) listOfBs.get(0)).method2();
   }
}
public class B {
   public void method2() {
      ...
   }
}
```

3.3.4 Java Source Code Pattern for Usage Dependency Coupling

In UML class diagrams, dependency relationships in a Java application connect two classes to indicate that there is a connection between the two classes, and that the connection is more temporary than an association relationship. A dependency relationship indicates that the consumer class does one of the following: (1) temporarily uses a supplier class that has global scope, (2) temporarily uses a supplier class as a parameter for one of its operations, (3) temporarily uses a supplier class as a local variable for one of its operations, or (4) sends a message to a supplier class.

Java has three types of *use*. One is that class A uses a variable of type B, as in examples 1.1 through 1.3; second is that class A directly uses a variable that is defined in class B, as in example 2; third is that class A directly invokes a static method of class B, as in example 3. The examples are taken from the implementation of the ATM system, which is provided by Briand and his colleagues [BFL02].

Example 1.1 shows that class ATMApplet uses an object of type Money by passing it in a method call. Example 1.2 shows that class ATM uses a Money type object as a return value. Example 1.3 line 7 shows that the parameter *amount*, which is an object of *Money* type, is used as a parameter in a method call. Example 1.3 line 10 shows that the parameter *amount* which is an object of *Money* type is assigned to a local variable, *_currentTransactionAmount*. Here the whole object is used. If classes A and B are coupled through this kind of relationship, the structure of class B is not used at all. Only an object of class B type is used in its entirety. Hence, any change to the internal structure and content of B, for example, a variable change, or a method change, will not impact A. However, a change to the class name will have an impact, because it is used in A. Also, class B must exist to test A.

Example code 1.1

```
public class ATMApplet extends Applet implements Runnable {
  ...
 public void run() {
    while (true) {
       Money initialCash = _theATM.startupOperation();
       _theATM.serviceCustomers(initialCash);
    }
 }
 ...
}
```

Example code 1.2

```
public class ATM {
  ...
 public synchronized Money startupOperation() {
    ...
    return _operatorPanel.getInitialCash();
 }
}
```

Example code 1.3

```
public class Bank {
  ...
 public int initiateWithdrawl
           ( int cardNumber, int PIN, int ATMnumber,
```

```
            int serialNumber, int from,
            Money amount,
            Money newBalance /* return value */,
            Money availableBalance /* returnValue */) {
    ...
 if (_availableBalance[_currentTransactionAccount ].less(amount))
        return Status.INSUFFICIENT_AVAILABLE_BALANCE;
     ...
     _currentTransactionAmount = amount;
  }
 }
```

Example 2 shows that class ATM directly uses the NO_CARD variable of class CardReader. In this case, the NO_CARD variable is a *public static final* variable, which serves as a constant. Only *public static final* variables can be used in this manner. This is similar to the traditional *global coupling*, and it is measured with the attribute reference coupling measures V_d and V_t.

Example code 2

```
public class ATM {
  ...
 public void serviceCustomers( Money initialCash ) {
    ...
    while (_state == RUNNING ) {
       int readerStatus = CardReader.NO_CARD;
    }
 }
```

```
 ...
}
```

Example 3 shows that class Session directly calls the *chooseTransaction()* method of class Transaction. The *chooseTransaction()* method is a *public static* method. Only *public static* methods can be used in this manner.

Example code 3

```
public class Session {
  ...
 public void doSessionUseCase() {
    ...
    _currentTransaction =
        Transaction.chooseTransaction(this, _atm, _bank);
 }
 ...
}
```

3.3.5 Java Source Code Pattern for Generalization (Inheritance) Coupling

Generalization is also called *Inheritance*. It occurs when one class inherits from another. In Java, we identify *generalization/inheritance* relationships through the "extends" keyword.

```
public class A {
   public void method1(){
   }
```

```
}

public class B extends A {
   public void method2() {
      ...
   }
}
```

In this example, class A is a *generalization* of class B. In other words, class B *extends* from class A. Class A can be an abstract class or a regular class. When class A is an abstract class, this relationship forms an abstract class implementation type coupling, and then class A is a regular class. Thus, the inheritance relationship forms an inheritance coupling.

Analyzing inheritance requires us to consider several issues:

- How much of the super class is inherited by the subclass? It seems that we cannot treat all inheritance as the same. The amount of content that a subclass inherits from its superclass should affect how we find optimal test orders and carry out our change impact analysis.

- If a super class has an aggregation, composition, or association relationship with other classes, do subclasses have the same relationship with those classes? The following code is an example:

```
public class WithdrawalTransaction extends Transaction {
   public int getTransactionSpecificsFromCustomer() {
      _fromAccount =
```

```
            _bank.chooseAccountType( "withdraw from", _atm );
        ...
    }
}
```

The *_bank* and *_atm* variables are defined in the superclass *Transaction*, and they form aggregation relationships in class pairs $\{Transaction, ATM\}$ and $\{Transaction, Bank\}$.

- It seems that the super() method makes a difference in this issue. For example, in the following sample code, the *WithdrawalTransaction* class calls the super() method of the *Transaction* class. The things to consider are whether WithdrawalTransaction has its own objects of Session, ATM, and Bank. When an object is created, the Java virtual machine allocates enough space for all the object's instance variables, which include all fields defined in the object's class and in all its superclasses. A subclass inherits only accessible members of its superclasses – and only if the subclass does not override or hide those accessible members. A class's members are the fields and methods actually declared in the class, plus any fields and methods it inherits from superclasses. A subclass does not inherit fields with *private* access specifier. As a result, the methods declared in a subclass can not directly access those private fields. Despite this, those fields are still part of the instance data of a subclass object. A superclass's constructor can explicitly be invoked using the super() statement.

```
public class Transaction  {
    public Transaction(Session session, ATM atm, Bank bank) {
```

```
        _session = session;
        _atm = atm;
        _bank = bank;
        _serialNumber = ++_lastSerialNumberAssigned;
        _newBalance = new Money();
        _availableBalance = new Money();
    }
}

public class WithdrawalTransaction extends Transaction {
    public WithdrawalTransaction(Session session,
                                 ATM atm, Bank bank) {
        super( session, atm, bank );
    }
}
```

This research counts the not-overridden methods and attributes in inheritance coupling. In abstract class implementation, the implemented abstract methods are counted in addition to the not-overridden methods and attributes. When a super class has an aggregation, composition, or association relationship with other classes, if a subclass can access and does not redefine those members of superclass that define these relationships, then subclasses have the same relationships as their superclasses.

3.3.6 Java Source Code Pattern for InterfaceRealization Coupling

InterfaceRealization Coupling between a class and an interface occurs when the class implements the interface. InterfaceRealization is identified through the "implements" keyword.

```
public interface A {
   public void method1();
}

public class B implements A {
   public void method1()
   {
      ...
   }
}
```

In this example, class A is an Interface, and class B implements the methods that are specified in class A. The interface based coupling measures measure the number of implemented methods and their return types and parameters.

3.3.7 Java Source Code Pattern for Exception Coupling

We define *exception coupling* between two classes to be when one class throws an exception and another class handles it. Exceptions occur in three situations: programming errors, client code errors, and resource failures. Java defines *Checked* and *Unchecked* exceptions to handle abnormal situations. *Checked exceptions* inherit from

the `Exception` class and has to be handled by the client code either in a `catch` clause or by forwarding it outward with the `throws` clause. *Unchecked exceptions* are `RuntimeExceptions` that also inherit from the `Exception` class. However, they get special treatment and are not required to be handled by the client code.

A checked exception thrown by a lower layer is a forced contract on the invoking layer to catch or throw it. Checked exceptions can also break encapsulation. For example, in the following code

```
public List getAllAccounts() throws
      FileNotFoundException, SQLException {
   ...
}
```

the method `getAllAccounts()` throws two checked exceptions. The client of this method has to explicitly deal with the implementation-specific exceptions, even if it has no idea what file or database call has failed within `getAllAccounts()`, or has no business providing filesystem or database logic. As a result, exception handling forces an inappropriately tight coupling between the method and its callers.

There are three ways for a class to be coupled with an Exception handling class. One is through the "throws" keyword, another is through the "throw" keyword, and the last is through the "catch" statement.

```
public class A {
   public void method1() throws UserException1 {
      ...
   }

   public void method2( boolean a, boolean b) {
```

```
        if ( a <> b )
         throw UserException2;
    }
 }

public class B {
   public void foo() {
      A a = new a();
      boolean x,y;
      try {
         ...
         a.method1( x, y );
      }
      catch (UserException1 ue) {
         ...
      }
   }
}
 public class UserException1 extends Exception {
   ...
 }

  public class UserException2 extends Exception {
   ...
 }
```

The "throws" and "throw" keywords can be used together in one method as in the following example:

```
public class Foo {

    public void execute() throws BuildException {

        super.execute();

        if (path == null) {

            throw new BuildException

                ("Must specify 'path' attribute");

        }

        execute("/sessions?path=" + URLEncoder.encode(this.path));

    }

}
```

Chapter 4: PROOF OF CONCEPT TOOL: JCAT

The purpose of the Java Coupling Analysis Tool (JCAT) is to analyze the structure and components of software packages written in Java, so that the individual object-oriented couplings among classes in the package can be identified automatically. This will allow the user to repair, debug, and modify a piece of software as needed.

JCAT was developed in Java using the JBuilder software application development tool. JCAT collaborates with a few other software applications to compute couplings. Figure 4.1 shows the *system context diagram* of JCAT. A *system context diagram* shows data flows between the main application and the other entities and abstractions with which it communicates. *System context diagrams* were developed to help understand the boundaries of systems [Gom00].

As shown in Figure 4.1, JCAT takes the absolute pathname of a *Java Code* package as an input argument, and asks the *JavaParser* to generate abstract syntax trees (ASTs) for each class file in the package. *JavaParser* is generated by the ANother Tool for Language Recognition system (ANTLR) [Par] from the Java grammar. ANTLR is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, or C++ actions. ANTLR provides support for tree construction, tree walking, and translation. ANTLR helps to build abstract syntax trees (ASTs) by providing grammar annotations that indicate what tokens are to be treated as subtree roots, which are to be leaves, and which are to be ignored with respect to tree construction. The

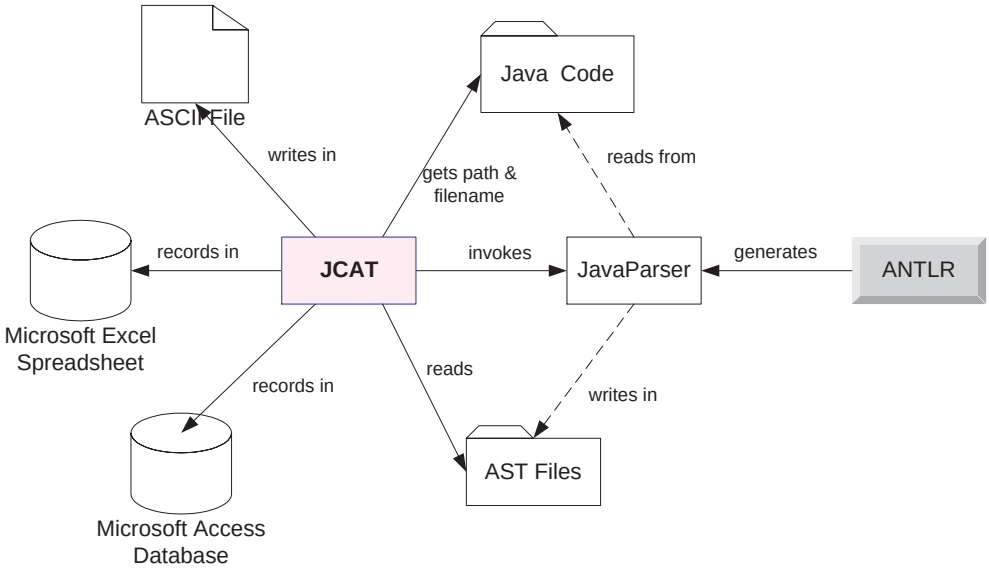

Figure 4.1: Context Diagram of JCAT

ASTs are saved as ASCII text files in the *AST Files* folder. JCAT reads the AST file of a Java class, then extracts information about inheritance, variables, method definitions, method calls, and variable uses for each class. The extracted information is stored in the *Microsoft Access database tables*. The database schemas are described in Section 4.3. According to the definitions of the couplings, we formulated queries for each coupling level. JCAT sends these queries to the database to compute the couplings, and reformats the query results into tabular forms. Finally, JCAT writes the coupling tables either in ASCII text files in the *ASCII Text Files* folder or in *Microsoft EXCEL Spreadsheets*, at the user's direction.

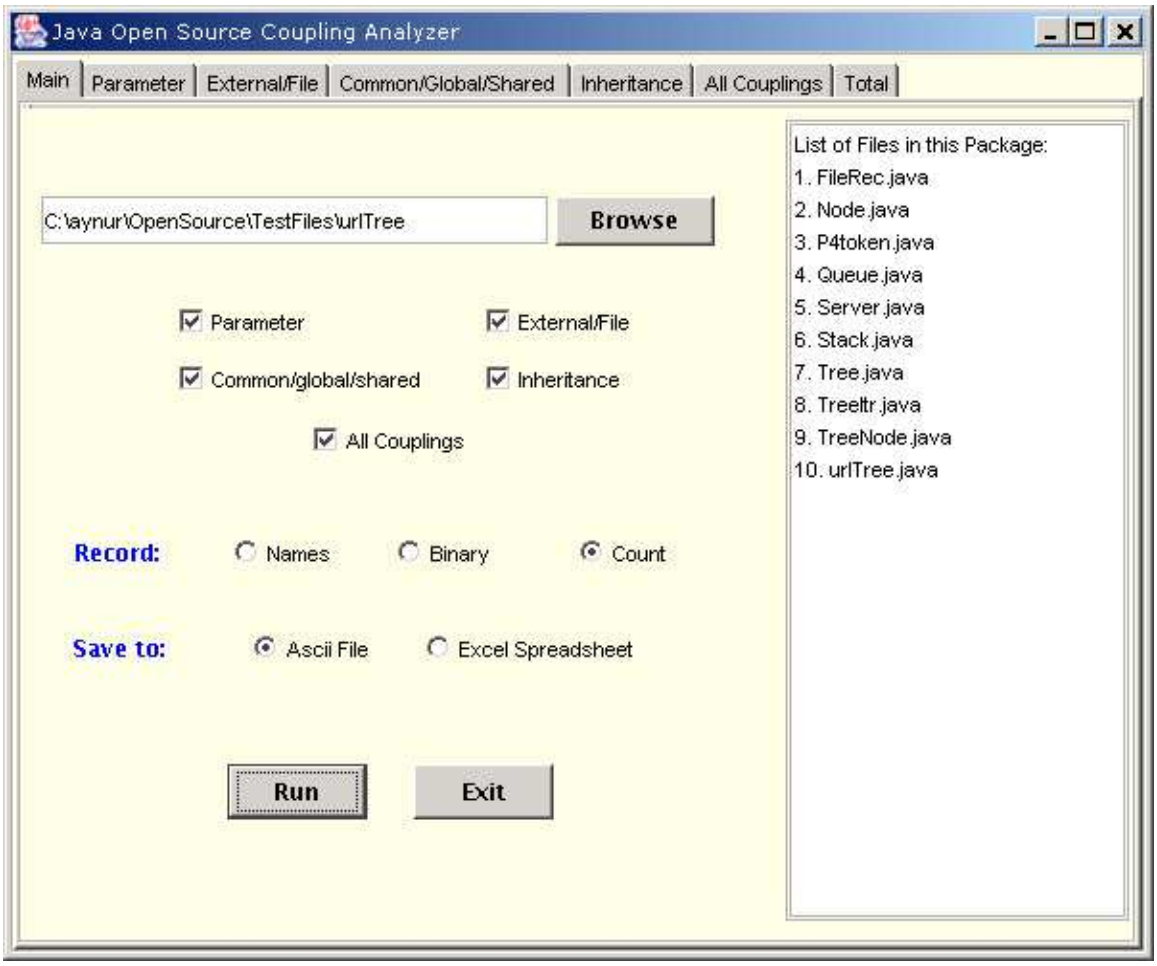

Figure 4.2: The Main User Interface of JCAT.

4.1 JCAT User Interface

Figure 4.2 shows JCAT's main screen. There are seven tabs in JCAT's user interface: **Main**, **Parameter**, **External/File**, **Common/Global/Shared**, **Inheritance**, **All Couplings**, and **Total**. The **Main** tab is shown in Figure 4.2; it lets the user enter a package to be analyzed, select the desired coupling level(s), choose the presentation format of the computed coupling(s), and determine where to save the

computation results. The user can either enter the pathname of the target Java source code package or browse to the target. JCAT can compute four levels of couplings. The user can select which couplings to compute by choosing from the checkboxes; any subset or all four can be chosen. The **Record** choice gives the user the option of either finding the existence of, or computing the number of occurances of a coupling between two classes. These choices are called "Names," "Binary," and "Count." "Names" and "Binary" choices will lead to computation of the existence of a coupling. The difference between "Names" and "Binary" options is that with the "Names" option, the existence of a coupling between two classes will be shown with the name of the coupling for easy vision, where as with the "Binary" option, the existence of a coupling between two classes will be shown with the number "1". With the "Count" option, JCAT computes all the instance couplings for each coupling level. The **Save to** option allows the user to save the coupling results to either ASCII text files or MS EXCEL spreadsheets. In either case, the coupling data is saved in a tabular format. The table rows and columns both represent file names. Each table entry has the coupling information between the two files. Finally, the "Run" button starts the computation, and the "Exit" button terminates JCAT.

After the target Java source code package is analyzed, the **Parameter** tab shows the parameter coupling result among the classes. The **External/File**, **Common/ Global/Shared**, and **Inheritance** tabs show the external, common, and inheritance couplings. The **All Couplings** tab shows all couplings together in one table. The **Total** tab shows the total number of each coupling for each class and for the whole package.

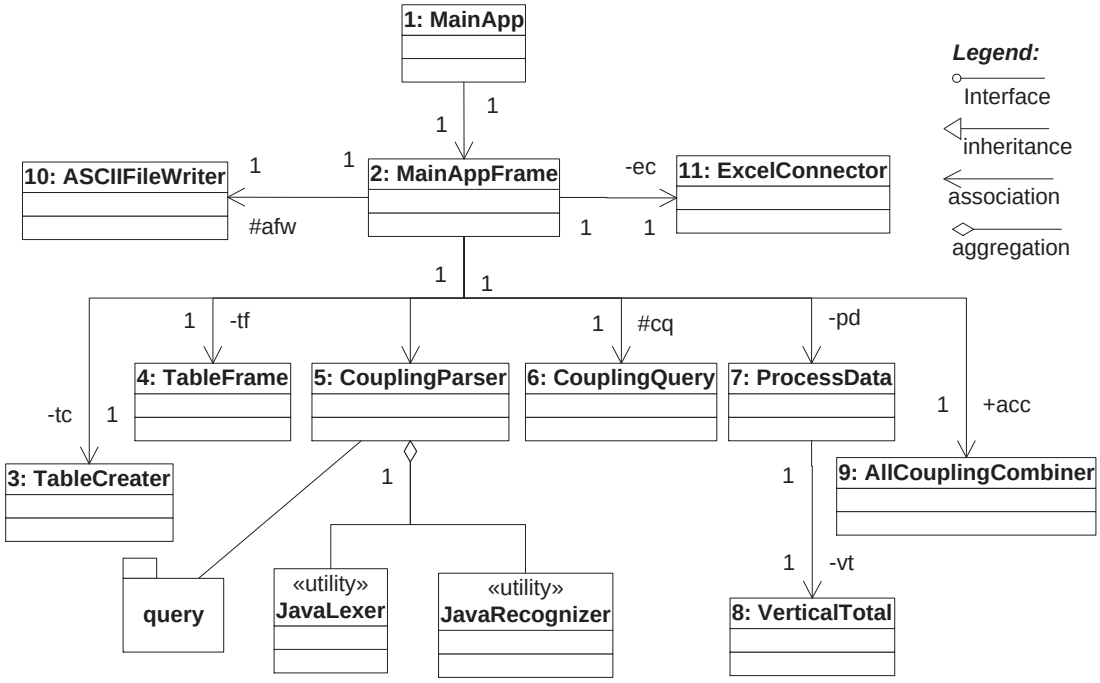

Figure 4.3: Class Diagram of Package *coupling*

4.2 JCAT Design

There are two packages in JCAT: *coupling*, the main package, and *query*, a subpackage of *coupling*. Figure 4.3 shows the class diagram for *coupling*. The *coupling* package is responsible for accepting the input, parsing the Java source code, invoking the methods of classes in the *query* package to extract coupling information, formatting the coupling results for presentation, and exporting the results to an output file and user interface. The following paragraphs describe each class in *coupling*.

The *query* package is responsible for computing the summary tables that are described in Section 4.3. Before explaining the responsibilities of each class, we present the abstract syntax tree (AST) nodes of Java classes that are used by JCAT in

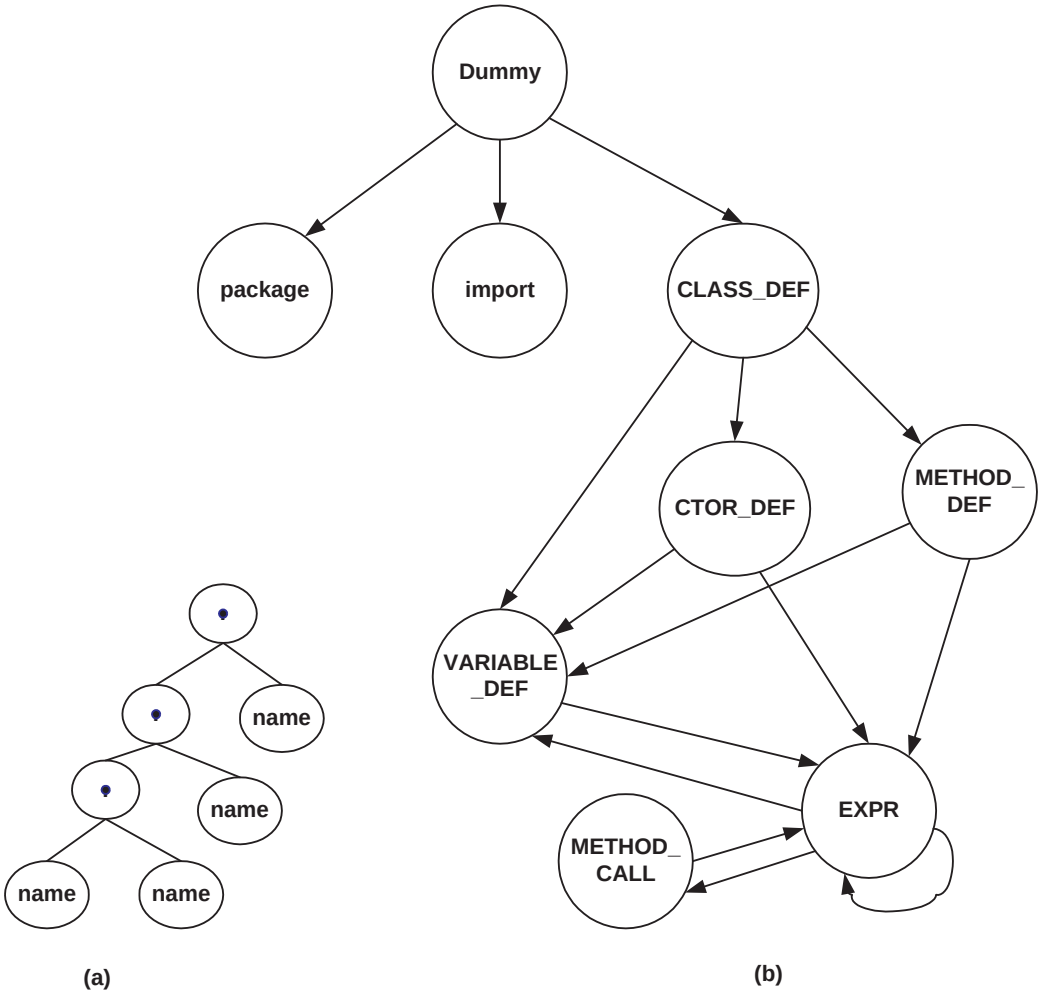

Figure 4.4: AST Nodes for Coupling Computation

computation.

Figure 4.4 (b) shows the AST nodes that JCAT pays particular attention to. Figure 4.4 (a) shows the tree structure of a name space. Since every node in Figure 4.4 (b) can have Figure 4.4 (a) as a subtree in a detailed description, we single out this structure and present it to explain some of the functions of the classes.

The *Dummy* node in Figure 4.4 is used to connect all the nodes. The *package*

node represents the package name of a class. A class may or may not have this node. The *import* node represents one imported package or class. There can be zero, one, or more of these nodes. The *CLASS_DEF* node contains two categories of information: (1) the name, type (class or interface), and pathname of a class, and (2) the name of any super classes this class inherits from. *CLASS_DEF* consists of zero or more *VARIABLE_DEF*, *CTOR_DEF*, and *METHOD_DEF* nodes. The *VARIABLE_DEF* node has information about the name, type, modifiers, and value of a variable. All *VARIABLE_DEF* nodes, whether they are for the class variables or for the method variables, have the same tree structure. For the sake of simplicity of the figure, the *CLASS_DEF* and *METHOD_DEF* nodes point to the same *VARIABLE_DEF* node in Figure 4.4. The *VARIABLE_DEF* node has zero or more *EXPR* nodes as children when this node assigns a value to the defined variable.

The *CTOR_DEF* node has information about a constructor of the class. The *METHOD_DEF* node has information about the modifier, return type, name, and parameters of a method. Both *CTOR_DEF* and *METHOD_DEF* nodes may consist of many child nodes such as *VARIABLE_DEF*, *TRY*, *EXPR*, *CATCH*, and *FINAL*, etc. JCAT is mainly concerned with the *VARIABLE_DEF* and *EXPR* nodes among the children of *METHOD_DEF*. The *EXPR* node contains information about an expression. *EXPR* is a complicated node and it has many children, including *VARIABLE_DEF*, *METHOD_CALL*, and itself. This is because an expression may contain variable definitions, method calls, and other expressions. JCAT extracts information about variable definitions, variable usages, and method calls from the *EXPR* node and its children.

The *METHOD_CALL* node has information about a method call, which consists of a callee method name and a list of parameters. The *METHOD_CALL* node has

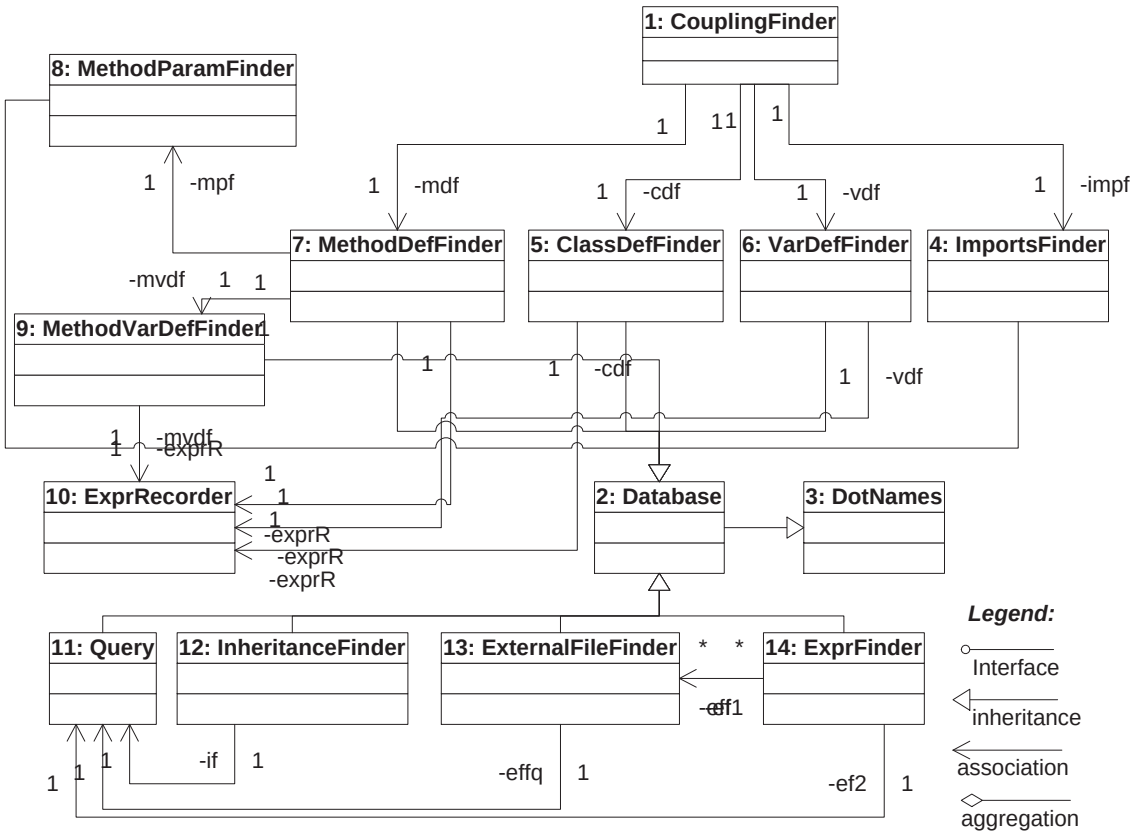

Figure 4.5: Class Diagram of Package *query*

zero or more *EXPR* nodes as children. We can see that the *EXPR* node is both a child and a parent node. This is because the list of parameters may contain variable definitions and expressions.

Figure 4.5 shows the class diagram of the *query* package. The classes in the *query* package extracts and processes the information at the AST nodes that are shown in Figure 4.4. The following paragraphs describe each class.

4.3 Summary Tables

JCAT computes the summary information for a class by using a combination of textual inspection, static structure analysis, and a restricted form of forward program slicing. First, the classes in the *JavaParser* package generates an Abstract Syntax Tree (AST) file for a Java source code file. Next, the classes in the *query* package make one pass over the text of the AST file, identify the class information, the definitions and uses of all variables, constructors, and methods, inheritance between classes, and the definitions and uses of external files, and saves them in MS Access database tables.

We have created 20 database tables to put the information that is related to the couplings in question. The information can be divided into six categories: *class*, *method*, *variable*, *variable usage*, *method calls*, and *external device*. The *class* category information is stored in class_def, class_modifiers, imported_classes, imported_packages, and inheritance tables. Figure 4.6 shows the tables and their structure in this category. The acronym "PK"in the figure stands for "Primary Key" and "FK" stands for "Foreign Key".

The table class_def is designed to store the *class_name*, *pathname*, *class_type* (i.e., class or interface), and *package_name* of each class. Each class has a unique *class_id*. The imported classes and packages in a class are saved in separate tables. Sometimes one physical file may contain several classes and all classes contained in this file would share the same imported classes and packages. The $file\_id$ field, which is used in three tables, is used to associate the class_def table with the imported_classes and the imported_packages tables. The class_modifiers table stores the modifier information of class definitions. This table uses the *class_id* field of the table class_def as a foreign key to associate the table entries in tables class_def and class_modifiers.

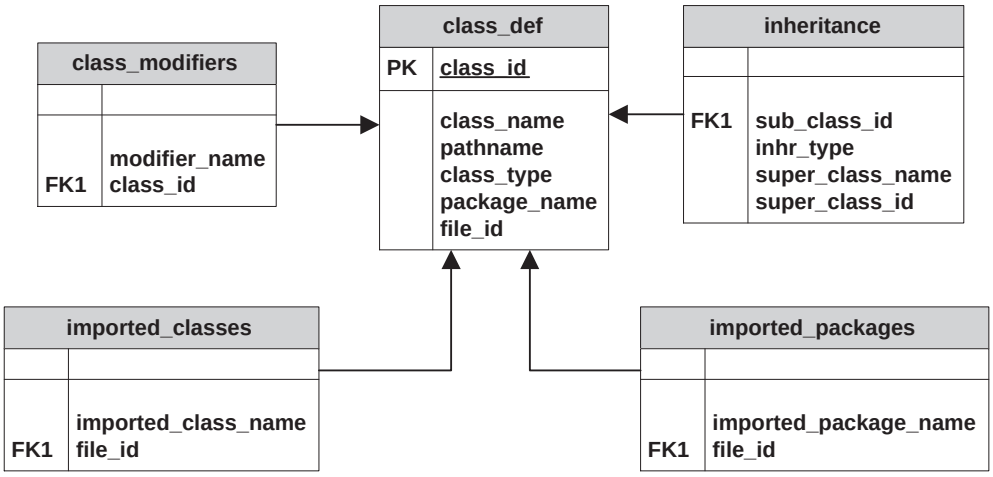

Figure 4.6: Table Schema for Class Category

Filling the inheritance table requires computation beyond simple text scanning. We first obtain the *sub_class_id*, *inhr_type* (e.g., extends or implements), and *super_class_name* when scanning the AST file. The *sub_class_id* is computed later. Algorithm 1 shows the steps in getting the *sub_class_id*.

The second of the six categories of information, *method* information, is stored in the method_def and method_modifiers tables, as shown in Figure 4.7. Because these two tables are related to the table class_def, they are shown together, but the class_def table is shaded to make it distinct. The *method_name* and *return_type* of each method are saved in the method_def table. Each method is distinguished by a pair of *method_id* and *class_id*. The modifiers of each method are saved in the method_modifiers table. The *method_id* key connects the two tables.

The variable level information is stored in the var_def, var_modifiers, method_var_def, method_var_modifiers, method_parameters, and param_modifiers tables, as

Algorithm 1 Get `super_class_id`

Require: *class_def, imported_classes, imported_packages* tables have been filled out.

Ensure: The *class_id* of a super class is identified.

```
1: super_class_id ⇐ −1
2: // Check imported classes
3: Iterator itr ⇐ imported class names
4: String superClassName, subClassName
5: while itr.next() ≠ null do
6:    superClassName ⇐ next imported class name
7:    if superClassName = subClassName then
8:       super_class_id ⇐ class_id of superClassName, break
9:    end if
10: end while
11: if super_class_id = −1 then
12:    // Check the classes in the imported packages
13:    Iterator itr ⇐ class names in imported packages
14:    String superClassName, subClassName
15:    while itr.next() ≠ null do
16:       superClassName ⇐ next class name
17:       if superClassName = subClassName then
18:          super_class_id ⇐ class_id of superClassName, break
19:       end if
20:    end while
21: end if
22: if super_class_id = −1 then
23:    // Check the classes in the local container package
24:    Iterator itr ⇐ class names in the local package
25:    String superClassName, subClassName
26:    while itr.next() ≠ null do
27:       superClassName ⇐ next class name
28:       if superClassName = subClassName then
29:          super_class_id ⇐ class_id of superClassName, break
30:       end if
31:    end while
32: end if
```

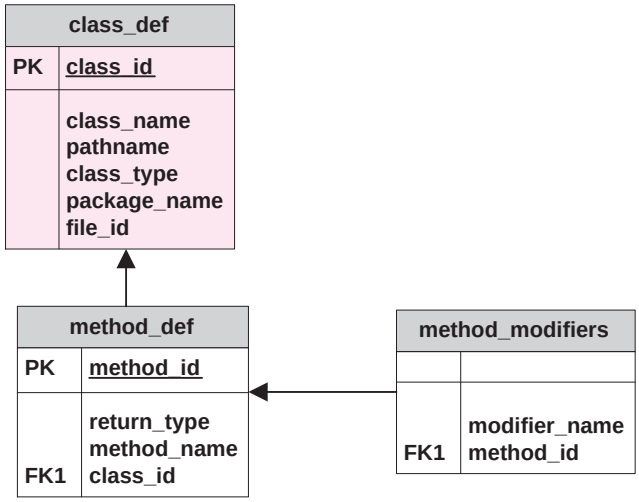

Figure 4.7: Method Level Tables

shown in Figure 4.8. There are three types of variables: class, method, and parameter. A *class variable* is a variable that is defined at the class level. A *method variable* is defined in a method. The parameters to a method are saved separately in the method_parameters table to make it convenient for later analysis. The related class_def and method_def tables are presented as well to show their relationship with the variable tables.

The *variable usage* level information is stored in the var_use_method and var_use_class tables. Figure 4.9 shows these two tables and other related tables. Since only the class level variables can be shared among classes, the var_def table is shown from the variable level tables. In the first step of the computation, *class_id* or *method_id*, *var_name*, and *var_actual_type* information are obtained by scanning the AST file. The *var_id* is obtained through the computation in Algorithm 2.

The *method call* level information is stored in the constructor_call_method, constructor_call_class, method_call_class, and method_call_method tables. Figure

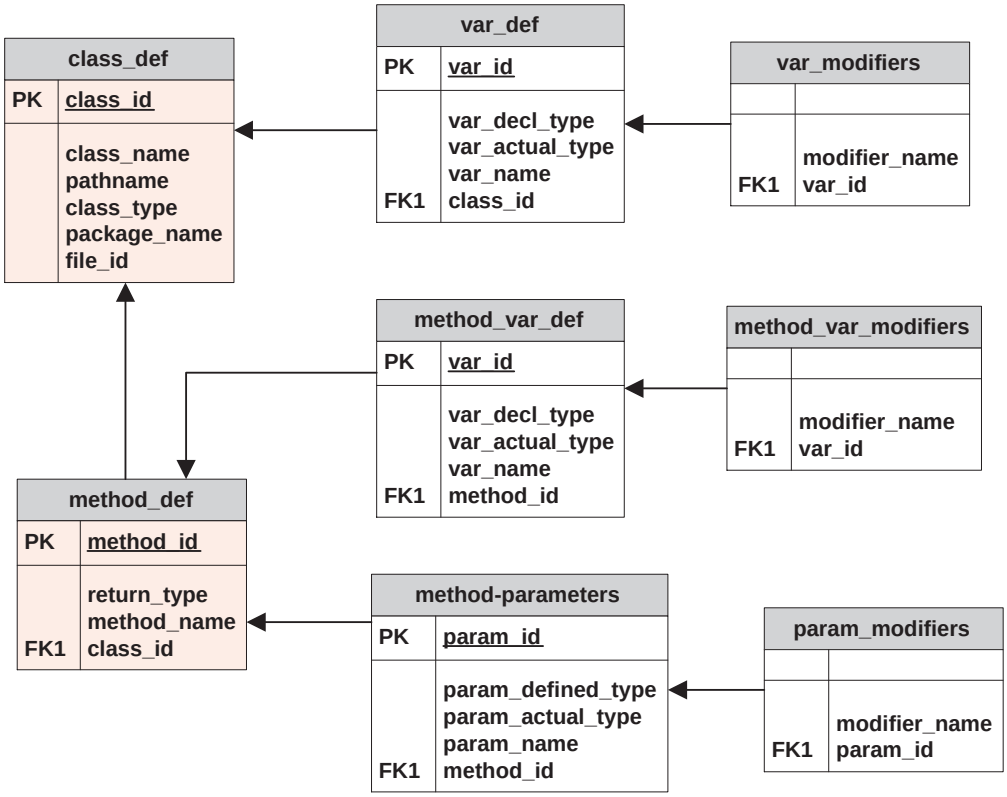

Figure 4.8: Variable Level Tables

4.10 shows the tables and their structure. The shaded tables are the parent tables and the arrows show the relationship among the tables. The method calls are saved in separate tables according to their scope. For example, if a method call occurs inside a method, this event and the related information are saved in the method_call_method table. Similarly, if a method call occurs at the class level, the information is saved in the method_call_class table. The invocation of a constructor is processed in the same way.

The caller's *id* and the callee's *name* can be obtained at the scan of the AST file. However, obtaining the callee's *id* requires some computation. The process of getting *callee_class_id* is the same as in Algorithm 1.

Algorithm 2 Get `var_id`

Require: *class_def, method_def, var-def* tables have been filled out.

Ensure: The *var_id* of a given *var_name* is identified.

1: $var\_id \Leftarrow -1$
2: $var\_name \Leftarrow objectName.varName$
3: // separate object name and variable name
4: $objectName \Leftarrow obj$
5: separate object name and var name.
6: // For example, varname = a.b, then a is object, b is variable
7: get class id of a
8: query var_def table with class_id and varname b
9: if var_id = -1 then get super class ids of class a
10: WHILE not empty
11: look into super class i for varname
12: if var_id = 0 then look next
13: String $qry1 \Leftarrow$ "SELECT *class_id* FROM *class_def* WHERE *class_name* = "
14: String $qry2 \Leftarrow$ "SELECT *class_id* FROM *external-device-usage* WHERE *device_id* = "

15: Database *db*
16: ResultSet $rs1,\ rs2$
17: Hashtable *externalCouplings*
18: $rs1 \Leftarrow db.execute(qry1)$
19: **while** $rs1.next() \neq null$ **do**
20: $did = rs1.getDeviceID()$
21: $rs2 \Leftarrow db.execute(qry2 + did)$
22: **if** $rs2.size() \geq 2$ **then**
23: record external coupling among classes in $rs2$ into *externalCouplings*
24: **end if**
25: **end while**

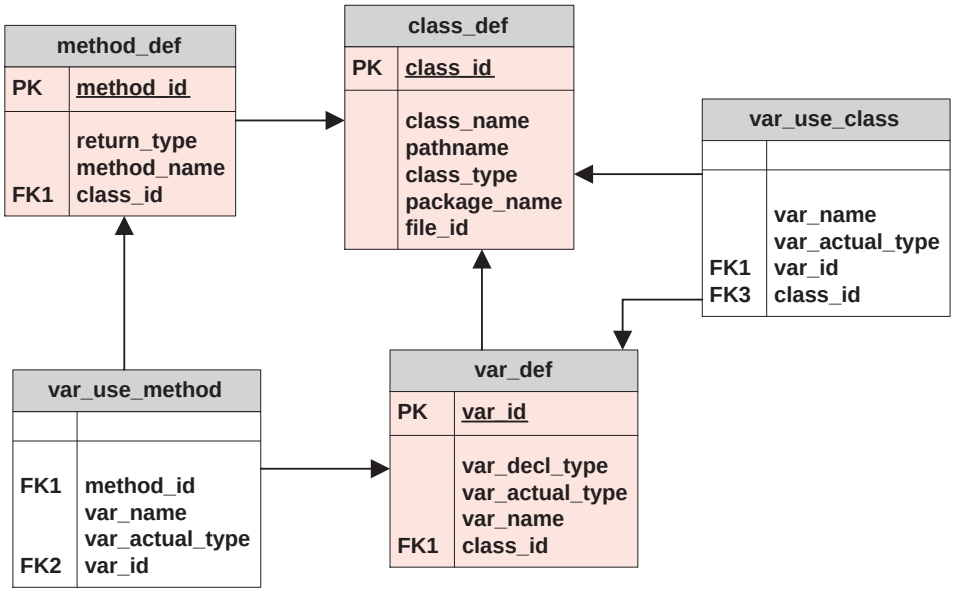

Figure 4.9: Variable Usage Level Tables

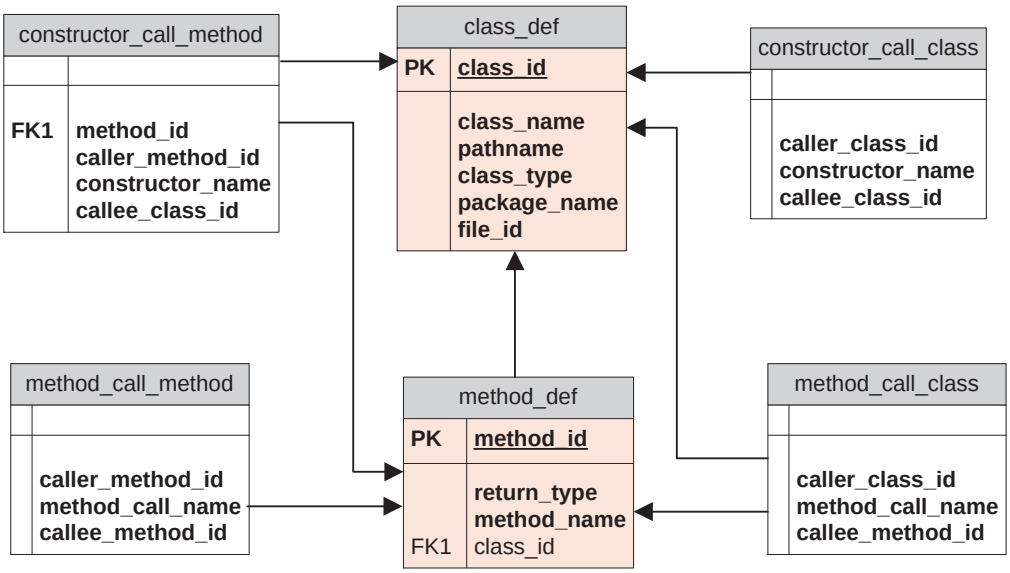

Figure 4.10: Method Call Level Tables

Chapter 5: APPLICATIONS OF THE COUPLING MODEL

This chapter gives an overview of how the coupling model and measures are applied to the three problems of this thesis. As such, it serves as an introduction to the following three chapters. Figure 5.1, a UML conceptual analysis model, gives an overview of the current application of coupling measures defined in this research. The research starts by parsing program source. Although not shown in the figure, the parsing is done by the help of parsing tools. A parsing tool generates abstract syntax trees for each class in a package. Next, as described in Chapter 4, coupling measures are computed by analyzing the abstract syntax trees. Then, the coupling measures are applied to three specific problems: class integration and test order, change impact analysis, and design pattern detection. Other possible applications are discussed in the future work section of Chapter 10.

5.1 Applying Coupling Measures to Class Integration and Test Order

In theory, a high quality software design would not include dependency cycles. In reality, however, dependency cycles are common. Some designers explicitly include cycles, either accidentally or disregarding the best advice of their teachers and books. Programmers make implementation decisions that add cycles that did not exist in the design. Consequently, software developers must break cycles to find optimal orders

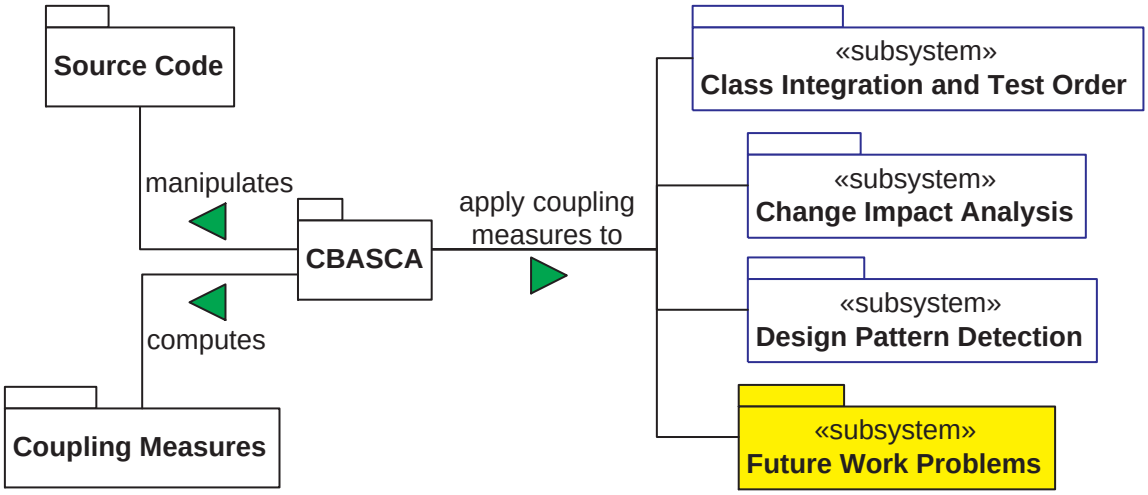

Figure 5.1: Application of the Coupling Model

to integrate and test classes.

The goal of class integration and test order research is to find an optimal integration order of classes for testing. When no cycle of dependency exists, then classes can be integrated and tested in reverse topological order. However, when dependency cycles exist, ordering becomes nontrivial. Figure 5.2 gives an example of a dependency cycle. There are three nodes and three edges in this diagram. Any edge can be broken to have a test order. When an edge is broken to eliminate a cycle, a test stub must be created for the absent class. Test stub creation adds additional cost to testing. Depending on the nature of relationships among classes, the cost of creating a stub can be quite different. This research defines *test stub complexity* using coupling measures, and computes it for each class. Then, the *test stub complexities* of classes are used to compute an optimal order for integration and testing of classes. The details of this work are given in Chapter 6.

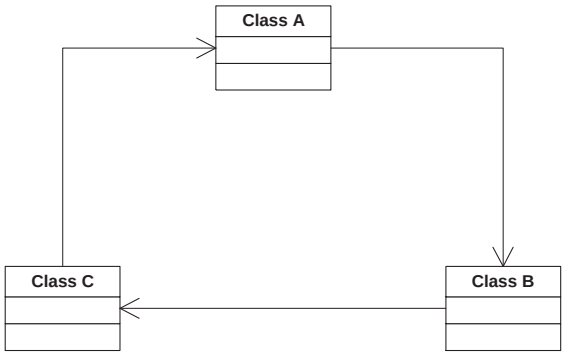

Figure 5.2: An Example of a Dependency Cycle

5.2 Applying Coupling Measures to Change Impact Analysis

A major problem for developers in a changing environment is that changes in one class, even small changes, can ripple through software to cause major unintended impacts elsewhere. Change impact analysis identifies possible ripple effects of a change and tries to build change or effort prediction models for how to effectively and efficiently implement changes. The key insight to this research is that a change can only have ripple effects through its relationships with other classes. Thus, it can be said that **there is no ripple effect without couplings** between classes. This research uses coupling measures to compute the *change impact set* of a given change at implementation level, then uses the result in computing two related metrics, *change impact* and *sensitivity to change*. The details of this work are given in Chapter 7.

5.3 Applying of Coupling Measures to Design Pattern Detection

Design patterns are used to impose structure on the system through abstractions. Consequently, the ability to automatically identify design patterns in the implementation can help developers comprehend existing designs and provide the information needed for refactoring. Thus, design pattern identification from source code can help improve software maintainability and evaluate how well implementation conform to design.

We have developed a design pattern detection methodology in which both the system under study as well as the design pattern to be detected are described in terms of graphs. In particular, the approach employs a coupling matrix representing all important aspects of their static structure. To detect patterns, we employ a graph similarity algorithm, which takes the system and the pattern graph as input, and calculates similarity scores between their vertices. The details of this work are given in Chapter 8.

Chapter 6: COUPLING-BASED CLASS INTEGRATION AND TEST ORDER

A common problem in inter-class integration testing of object-oriented software is to determine the order in which classes are integrated and tested [KGH+95a]. When one class requires another to be available before it can be executed, we define this kind of relationship to be a *dependency*. These two classes can be characterized as *server* and *client* classes. The client class is being compiled or executed and the server class must be present. This dependency has a direction, and is based on one or more object-oriented relationships.

Computing an optimal class integration and test order (CITO) is a focus of the software testing community. When there is no cycle in the dependency of classes or subsystems, the *class integration and test order* (CITO) can be computed by a simple reverse topological ordering of classes based on their dependencies. However, dependency cycles are common in real-world systems and when present, topological sorting is not possible [KGH+95a].

To solve the CITO problem in the presence of cycles, the cycles must be broken. The effect of breaking a dependency cycle is that a *stub* must be created for the class that is no longer present, thus increasing the cost of integration testing. Our goal is to find an optimal order that minimizes the stubbing effort. Stubbing effort has many elements to consider, and, therefore, cannot be completely measured or estimated [BFL02]. It has also been suggested that creating stubs can be error prone and costly

[Bei90].

Coupling measurements capture class relationships. The main goal of this study is to use coupling measurement to estimate stubbing effort and develop an efficient technique to find an *optimal integration order*.

This chapter discusses existing solutions, introduces our model, then presents results from a case study taken from Briand et al.'s paper [BFL02].

Section 6.1 summarizes existing approaches to the class integration and test order (CITO) problem and Section 6.2 describes our model and algorithms. The algorithms are explained in detail with a running example. Section 6.4 presents a case study that uses the same system as used by Briand et al. [BFL02] and Section 6.6 summarizes the results.

6.1 Summary of Existing Solutions

The class integration and test order problem has been addressed by several researchers and several solutions have been proposed. The solutions can be categorized into *graph-based* and *genetic algorithm-based*(GA) approaches. Section 2.2 recapitulates existing solutions and discusses their advantages and disadvantages.

In graph-based approaches, classes and their relationships in software are modeled as object relation diagrams (ORD) or test dependency graphs (TDG). An ORD or TDG is a directed graph $G(V, E)$, where V is a set of nodes representing classes and E is a set of edges representing the relationships among classes. The class integration and test order problem is to find an ordering of nodes in the graph so that the classes can be integrated and tested with minimum effort.

In most papers [BLW03, KGH$^+$95a, TD97, TJJM00], the testing effort is estimated

by counting the number of test stubs that need to be created during integration testing. This method assumes that all stubs are equally difficult to write. One recent paper tries to consider test stub complexity when estimating the testing effort [MCL03].

In the genetic algorithm-based approach [BFL02], inter-class coupling measurements and genetic algorithms are used in combination to assess the complexity of test stubs and to minimize complex cost functions.

To summarize, the existing graph-based approaches use high level, course grained, estimates of test stub complexity. The GA approach must be run many times, greatly complicating the process. The coupling-based algorithms described here only run once and use more information to provide a more precise estimation of test stub complexity.

6.2 A New Model and Algorithms

This section introduces a new graph-based solution for the CITO problem. Our approach is different from other graph-based approaches in three respects. First, we model classes and their relationships with weights on both nodes and edges. Second, weights of nodes and edges are based on a quantitative measure of coupling. Last, we use algorithms that incorporate edge and node weights as well as the number of cycles in breaking cycles.

6.2.1 Modeling Class Integration and Test Order

As said in Section 6.1, dependencies among classes are usually modeled in graphs. The CITO problem then becomes finding an acyclic graph with minimum cost. The cost is usually modeled as the number of test stubs to be generated during the testing

activities. This section uses a different abstraction to test dependencies among classes and a different cost model for the testing effort. We model the test dependencies among classes using a *Weighted Object Relation Diagram (WORD)*, and model the testing effort by computing test stub complexities using coupling information.

Chapter 3 defined eight coupling base types and four coupling measures for each base type. The coupling base types are *Inheritance, Abstract class implementation, interface implementation, Composition Coupling*, *Aggregation*, *Exception Coupling*, *Association*, and *Dependency*. For each coupling base type, coupling measures are defined to measure the dependencies between a server and client class in terms of the number of distinct and total variables used, the number of distinct and total methods (including constructors) called, the number of distinct and total parameters sent, and the number of distinct and total return value types. These measures are connected through a "dot" notation. In this research, the CITO problem is addressed using the distinctly counted measures defined in Chapter 3 with slight modification.

$$CM(c_i, c_j) = CBT.V_d.M_d.R_d.P_d, \tag{6.1}$$

where c_i and c_j represent two classes that are coupled together,

$$CBT = \begin{cases} 5 & when\ coupling\ is\ based\ on\ inheritance \\ & or\ composition \\ 1 & for\ other\ coupling\ types \end{cases} \tag{6.2}$$

The dot notation is used to indicate that the five measures are independent but related. V_d, *number of distinct vars*, represents the number of distinct public vars of c_j that are directly used by c_i. M_d, *number of distinct methods*, represents the number of distinct public methods of c_j that are called by c_i. R_d, *number of distinct return*

types, represents the number of distinct return types that appear in M_d. P_{dist}, *number of distinct parameters*, represents the number of distinct parameters that appear in M_d.

Equation 6.2 assigns two values to CBT. This is because in CITO problem, inheritance, implementation, and composition relationships are considered as complex relationships and considered in one category, and the rest of the relationships considered in one category. We observe that the combination of coupling base types and the four coupling measures can syntactically estimate the content of a test stub needed by a client class. Thus, using the coupling measures, a *test stub complexity* for each class can be estimated in the context of the entire system.

We define two kinds of stub complexity. If a client class A depends on a server class B to function, we can quantify this dependency by identifying the scope of B used by A, as measured by coupling. We define this to be a *specific test stub complexity* of B to A. However, there can be other client classes that depend on B, and some usage of B can be overlapped among client classes. Figure 6.1 shows an example. In this example, classes A, B, and C all depend on D. Although it seems that D is heavily used, in fact only $m1()$ is called by all three classes. Thus, if we stubbed D, we would only need a stub for one method. We define the sum of dependencies/usages from all other client classes to B as the *total stub complexity* of server class B. When we compute the total stub complexity of a class, we will take into account the overlapping possibility of specific stubs. Thus, a total stub complexity of a class takes a value between the maximum and sum of several specific stubs complexity. Previous research did not consider overlapping.

We use the *specific test stub complexity* and *total stub complexity* to assign weights to edges and nodes in our WORD. In a WORD, nodes represent classes and edges

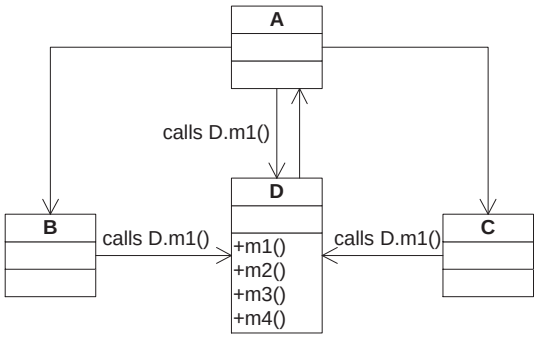

Figure 6.1: Example of Method Call Overlap

represent test dependencies among classes. Both nodes and edges are weighted. The node weight represents the *total stub complexity* of a class, and the edge weight represents the *specific stub complexity* of a server class to the client class that is connected by the edge. The graph can be acyclic or cyclic. If the graph is acyclic, then we can carry out integration testing in the reverse topological order of the graph. If the graph is cyclic, then we have to first break cycles. This forces us to create test stubs for the broken edges or removed nodes. We model the testing effort as the total complexity of stubs that are introduced during integration testing. The goal is to make the graph acyclic by removing certain edges and/or nodes, and the total weight of removed edges and/or nodes has to be minimum.

The model for the class integration and test order problem is formally defined as follows:

Let $G(V, E)$ be a node- and edge-weighted directed graph that models classes or components and their relationships. In the graph, nodes represent classes or components, and edges represent test dependencies among classes. The edge weights represent *specific stub complexities* and node weights represent *total stub complexities*. Our problem is to determine the nodes and edges with minimum total weight

to remove so that there are no cycles in G.

6.2.2 Measuring Stub Complexity

We use coupling measures to assign weights to edges and nodes in our weighted object relation diagram (WORD). After all couplings are measured in the form of equation 6.1, coupling measures between the source and target node classes are aggregated as one measure as follows:

$$\begin{aligned} cm_{e_i} &= \{\sum_{k=1}^{8} CM_k(v_m, v_n) \\ &\quad | \; v_m, v_n \in V, e_i = v_m \rightarrow v_n, e_i \in E\} \\ &= max(CBT).\sum_{k=1}^{8} V_{d_k}.\sum_{k=1}^{8} M_{d_k}.\sum_{k=1}^{8} R_{d_k}.\sum_{k=1}^{8} P_{d_k} \end{aligned} \tag{6.3}$$

This measure will be used to compute the weight of an edge and represents a *specific test stub complexity* of a class.

The coupling measures on edges are then further aggregated to nodes. A coupling measure on a node is computed from the coupling measures of the incoming edges of the node in the following manner:

$$\begin{aligned} cm_{v_i} &= \left\{ \left[max(cm_{e_{1,i}}, cm_{e_{2,i}}, ...cm_{e_{k,i}}), \sum_{l=1}^{k} cm_{e_{l,i}} \right] \right. \\ &\quad \left. | \; v_i \in V, e_{l,i} = v_l \rightarrow v_i, e_{l,i} \in E, |e_{l,i}| = k \right\} \end{aligned} \tag{6.4}$$

where $cm_{e_{1,i}}, cm_{e_{2,i}}, ...cm_{e_{k,i}}$ are coupling measures on the incoming edges of node v_i and the summation of coupling measures is the same as in equation 6.3. Equation cm_{v_i}

takes a value between the maximum and sum of coupling measures on the incoming edges of node v_i. This measure will be used to compute the weight of a node and represents the *total test stub complexity* of a class.

Our rationale for introducing weights for nodes is that specific stubs for a class may overlap. It is possible that certain methods or variables of a server class can be used by a number of clients in the same way. In this case, creating one stub for the server class can satisfy the needs of several clients.

We use Briand et al.'s method for estimating stubbing complexity [BFL02] from the coupling measures of edges and nodes. For a measure $Cplx()$, a complexity measure $\overline{Cplx()}$ is normalized as

$$\overline{Cplx(i,j)} = Cplx(i,j)/(Cplx_{max} - Cplx_{min}) \tag{6.5}$$

where $Cplx(i,j)$ represents a complexity information matrix, $Cplx_{min} = Min\{Cplx(i,j), i,j = 1,2,...\}$ and $Cplx_{max} = Max\{Cplx(i,j), i,j = 1,2,...\}$. They use two coupling measures A() and M(), the number of locally defined variables and the number of methods, to compute overall stubbing complexity:

$$SCplx(i,j) = (W_A.\overline{A}(i,j)^2 + W_M.\overline{M}(i,j)^2)^{1/2} \tag{6.6}$$

where W_A and W_M are weights and $W_A + W_M = 1$. Thus, for a given test order o, with d dependencies to be broken, an overall stubbing complexity for the order o is computed as

$$OCplx(o) = \sum_{k=1}^{d} SCplx(k) \tag{6.7}$$

The principle of not breaking inheritance and composition edges was ensured by

constraints in Briand et al.'s work. This paper takes a different approach, specifically, assigning higher values to the variable C in equation 6.1.

As shown in equation 6.1, our coupling measures use the number of parameters and the number of return value types in addition to the number of variables and the number of methods. We use the same normalization method as Briand et al., and include the additional coupling measures in the stubbing complexity estimation.

Using aggregated coupling measures on edges and nodes, a stubbing complexity is estimated as follows:

$$SCplx(i,j) = CBT + \quad (W_V \times \overline{V}(i,j)^2 + W_M \times \overline{M}(i,j)^2 + W_R \times \overline{R}(i,j)^2 + W_P \times \overline{P}(i,j)^2)^{1/2} \tag{6.8}$$

where CBT is the first measure from equations 6.3 and 6.4, W_V, W_M, W_R, and W_P are weights and $W_V + W_M + W_R + W_P = 1$. The $\overline{V}(i,j)$, $\overline{M}(i,j)$, $\overline{R}(i,j)$, and $\overline{P}(i,j)$ values are computed from equation 6.5 using values from equations 6.3 and 6.4.

Our objective is to find an optimal integration and test order o by determining a set of k nodes and/or l edges to be removed to make the $WORD$ acyclic such that the sum of the stubbing complexities for these nodes and edges is minimum:

$$OCplx(o) = \sum_{i=1}^{k} SCplx_{node}(i) + \sum_{j=1}^{l} SCplx_{edge}(j) \tag{6.9}$$

The following subsections present **three** general algorithms for making a cyclic graph acyclic using simple weight assignments on edges, nodes, and both.

6.2.3 Heuristics Algorithm for Breaking Cycles Using Edge Weights

This subsection presents an algorithm for breaking cycles using weight assignments on edges.

Algorithm 3 Eliminating Cycles in WORD (V,E)

1: Find all SCCs in WORD
2: **for** (each $scc_i(V_{scc_i}, E_{scc_i}) \in SCCs$) **do**
3: find all cycles CYCLES (totalCycles)
4: **for** (each $e \in E_{scc_i}$) **do**
5: find the number of cycles that use e ($cardinal\{cycles - through - e\}$)
6: compute the cycle-weight ratio
7: **end for**
8: **while** (totalCycles != 0) **do**
9: order all *edges* in descending order of their cycle-weight ratio
10: remove *edge* with highest cycle-weight ratio
11: totalCycle = totalCycle - number of cycles broken
12: update the number of cycles that use e ($cardinal\{cycles - through - e\}$) in the remaining edge set
13: recompute the cycle-weight ratio for the remaining edges
14: **end while**
15: **end for**

Algorithm 3 is illustrated through the example in Figure 6.2. Figure 6.2 is taken from Briand et al.'s paper [BFL02]. The edges represent general dependencies between two classes, not UML-specific relationships, with edge labels representing the specific stub complexity. Step 1 in Algorithm 3 finds one SCC in Figure 6.2, $\{E, A, C, H, D, B, F\}$. Steps 2 and 3 find the following 11 cycles in the SCC:

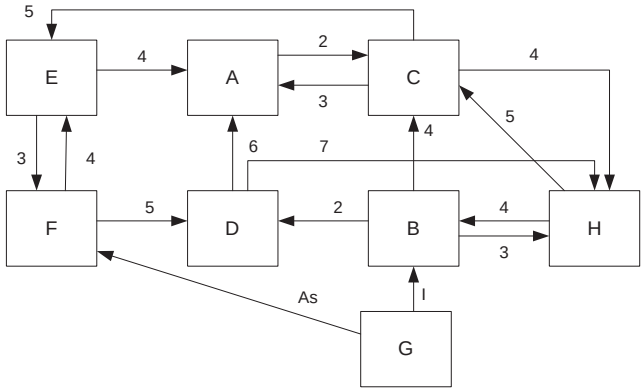

Figure 6.2: Example Weighted Object Relation Diagram (WORD)

(1) $E \rightarrow F \rightarrow E$

(2) $E \rightarrow A \rightarrow C \rightarrow E$

(3) $E \rightarrow F \rightarrow D \rightarrow A \rightarrow C \rightarrow E$

(4) $E \rightarrow F \rightarrow D \rightarrow H \rightarrow C \rightarrow E$

(5) $E \rightarrow F \rightarrow D \rightarrow H \rightarrow B \rightarrow C \rightarrow E$

(6) $A \rightarrow C \rightarrow H \rightarrow B \rightarrow D \rightarrow A$

(7) $A \rightarrow C \rightarrow A$

(8) $C \rightarrow H \rightarrow C$

(9) $C \rightarrow H \rightarrow B \rightarrow C$

(10) $H \rightarrow B \rightarrow H$

(11) $H \rightarrow B \rightarrow D \rightarrow H$

Table 6.1 shows the results from steps 4 through 7 of Algorithm 3. After the initial computation of CWR values for edges, the algorithm works in the following steps:

Table 6.1: **Cycle-weight Ratio for Edges in** SCC $\{E, A, C, H, B, D, F\}$

No.	Edge	Wt.	Cycles Involved	NC	CWR
1	**A → C**	**2**	**{2, 3, 6, 7}**	**4**	**2**
2	$B \rightarrow C$	4	{5, 9}	2	0.5
3	$B \rightarrow D$	2	{6, 11}	2	1
4	$B \rightarrow H$	3	{10}	1	0.33
5	$C \rightarrow E$	5	{2, 3, 4, 5}	4	0.8
6	$C \rightarrow A$	3	{7}	1	0.33
7	$C \rightarrow H$	4	{6, 8, 9}	3	0.75
8	$D \rightarrow A$	6	{3, 6}	2	0.33
9	$D \rightarrow H$	7	{4, 5, 11}	3	0.43
10	$E \rightarrow A$	4	{2}	1	0.25
11	$E \rightarrow F$	3	{1, 3, 4, 5}	4	1.33
12	$F \rightarrow D$	5	{3, 4, 5}	3	0.6
13	$F \rightarrow E$	4	{1}	1	0.25
14	$H \rightarrow B$	4	{5, 6, 9, 10, 11}	5	1.25
15	$H \rightarrow C$	5	{4, 8}	2	0.4

1. Choose an edge from table 6.1 with maximum cycle-weight ratio (CWR) and remove that edge from the *WORD*. At this point, the edge with the maximum cycle-weight ratio is $A \rightarrow C$ with a ratio of 2. Removing edge $A \rightarrow C$ breaks four cycles, 2, 3, 6, and 7, leaving seven cycles.

2. Recompute the cycle-weight ratio for the remaining edges. The result is shown in table 6.2. Two edges have the same maximum cycle-weight ratio, 14 and 11 with ratios of 1 (shown in bold). Our rule in this situation is to choose the edge that is involved in the larger number of cycles. This is edge 14, $H \rightarrow B$. Removing $H \rightarrow B$ breaks four cycles, 5, 9, 10, and 11, leaving three cycles.

3. Recompute the cycle-weight ratio for remaining edges in table 6.2. The result

Table 6.2: **Cycle-weight Ratio for Edges in** SCC $\{E, A, C, H, B, D, F\} - \{A \rightarrow C\}$

No.	Edge	Wt.	Cycles Involved	NC	CWR
2	$B \rightarrow C$	4	{5, 9}	2	0.5
3	$B \rightarrow D$	2	{11}	1	0.5
4	$B \rightarrow H$	3	{10}	1	0.33
5	$C \rightarrow E$	5	{4, 5}	2	0.4
6	$C \rightarrow A$	3	{ }	0	0
7	$C \rightarrow H$	4	{8, 9}	2	0.5
8	$D \rightarrow A$	6	{ }	0	0
9	$D \rightarrow H$	7	{4, 5, 11}	3	0.43
10	$E \rightarrow A$	4	{ }	0	0
11	$\mathbf{E \rightarrow F}$	**3**	**{1,4,5}**	**3**	**1**
12	$F \rightarrow D$	5	{4, 5}	2	0.4
13	$F \rightarrow E$	4	{1}	1	0.25
14	$\mathbf{H \rightarrow B}$	**4**	**{5, 9, 10, 11}**	**4**	**1**
15	$H \rightarrow C$	5	{4, 8}	2	0.4

is shown in table 6.3. In this table, the edge with maximum cycle-weight ratio is edge 11, $E \rightarrow F$. Removing edge $E \rightarrow F$ breaks two cycles, 1 and 4, leaving only cycle 8.

4. Recompute the cycle-weight ratio for the remaining edges, and at this point the edge with maximum cycle-weight ratio is edge 7, $C \rightarrow H$. Removing $C \rightarrow H$ breaks cycle 8, and makes the *WORD* acyclic.

Thus, we break all 11 cycles by removing four edges, $A \rightarrow C$, $H \rightarrow B$, $E \rightarrow F$, and $C \rightarrow H$. The total cost is 2+4+3+4 = 13.

Table 6.3: **Cycle-weight Ratio for Edges in** $SCC\{E, A, C, H, B, D, F\}$ **-**$\{A \rightarrow C, H \rightarrow B\}$

Edge No.	Edge	Weight	Cycles Involved	# of Cycles Involved	Cycle-Weight Ratio
2	$B \rightarrow C$	4	{ }	0	0
3	$B \rightarrow D$	2	{ }	0	0
4	$B \rightarrow H$	3	{ }	0	0
5	$C \rightarrow E$	5	{4}	1	0.2
6	$C \rightarrow A$	3	{ }	0	0
7	$C \rightarrow H$	4	{8 }	1	0.25
8	$D \rightarrow A$	6	{ }	0	0
9	$D \rightarrow H$	7	{4}	1	0.14
10	$E \rightarrow A$	4	{ }	0	0
11	$E \rightarrow F$	3	{1,4}	2	0.66
12	$F \rightarrow D$	5	{4}	1	0.2
13	$F \rightarrow E$	4	{1}	1	0.25
15	$H \rightarrow C$	5	{4,8}	2	0.4

6.2.4 Applying Algorithm 3 to A Special Case

A key difference between this research and previous research is the modeling of the cost of stubbing as edge weights. If we assign a weight of 1 to each edge, then our model is equivalent to the previous models. In addition, previous researchers modeled node weights as the sum of all incoming edges, which corresponds to our pessimistic approach. To facilitate comparison, we assign all edges in the graph in Figure 6.2 weight 1, to see if our algorithm gets the same results as Briand's [BLW03, BFL02].

Table 6.4 shows initial CWR values for edges. We briefly describe the process: first, edge $H \rightarrow B$ is chosen to be removed, breaking five cycles. The re-computation of CWR values for the remaining edges are not shown, but the next edge to remove is $E \rightarrow F$, breaking three cycles. Next, edge $A \rightarrow C$ is chosen, breaking two cycles.

Table 6.4: **Cycle-weight Ratio for Edges in** SCC $\{E, A, C, H, B, D, F\}$ **- All Edges Have the Same Weight**

No.	Edge	Wt.	Cycles Involved	NC	CWR
1	$A \rightarrow C$	1	{2, 3, 6, 7}	4	4
2	$B \rightarrow C$	1	{5, 9}	2	2
3	$B \rightarrow D$	1	{6, 11}	2	2
4	$B \rightarrow H$	1	{10}	1	1
5	$C \rightarrow E$	1	{2, 3, 4, 5}	4	4
6	$C \rightarrow A$	1	{7}	1	1
7	$C \rightarrow H$	1	{6, 8, 9}	3	3
8	$D \rightarrow A$	1	{3, 6}	2	2
9	$D \rightarrow H$	1	{4, 5, 11}	3	3
10	$E \rightarrow A$	1	{2}	1	1
11	$E \rightarrow F$	1	{1, 3, 4, 5}	4	4
12	$F \rightarrow D$	1	{3, 4, 5}	3	3
13	$F \rightarrow E$	1	{1}	1	1
14	$\mathbf{H \rightarrow B}$	**1**	**{5, 6, 9, 10, 11}**	**5**	**5**
15	$H \rightarrow C$	1	{4, 8}	2	2

There is one cycle left, 8, and we can break it by either removing $H \rightarrow C$ or $C \rightarrow H$. Here we can apply the heuristic of not breaking an *Aggregation* relationship and choose $C \rightarrow H$ to remove. Thus, we removed four edges in total: $H \rightarrow B$, $E \rightarrow F$, $A \rightarrow C$, and $C \rightarrow H$. This result is the same as the result from Briand et al.'s graph-based research, although the edges were removed in a different order.

6.2.5 Heuristics Algorithm for Breaking Cycles Using Node Weights

This subsection presents an algorithm for breaking cycles using weight assignments on nodes. Algorithm 4 is illustrated through the following examples.

Algorithm 4 Eliminating Cycles in WORD(V,E) Using Node Weights

```
1: Find all SCCs in WORD
2: for (each scc_i(V_scc_i, E_scc_i) ∈ SCCs) do
3:    find all cycles CYCLES (totalCycles)
4:    for (each v ∈ V_scc_i) do
5:       find the number of cycles that use v (cardinal{cycles − through − v})
6:       compute the cycle-weight ratio
7:    end for
8:    while (totalCycles != 0) do
9:       order all nodes in descending order of their cycle-weight ratio
10:      remove node with highest cycle-weight ratio
11:      totalCycle = totalCycle - number of cycles broken
12:      update the number of cycles that use v (cardinal{cycles − through − v}) in
         the remaining node set
13:      recompute the cycle-weight ratio for the remaining nodes
14:   end while
15: end for
```

The node weights model the amount of effort needed to create a stub for that class, which will be used by all classes that use it (*user classes*). In a pessimistic approach, each user class will need completely different stub functionality, so each user class needs a completely independent stub. For example, one user class may call methods $M1()$ and $M2()$, and another may call methods $M3()$ and $M4()$. This situation is modeled by case (1), where the node weight is the **sum** of incoming edge

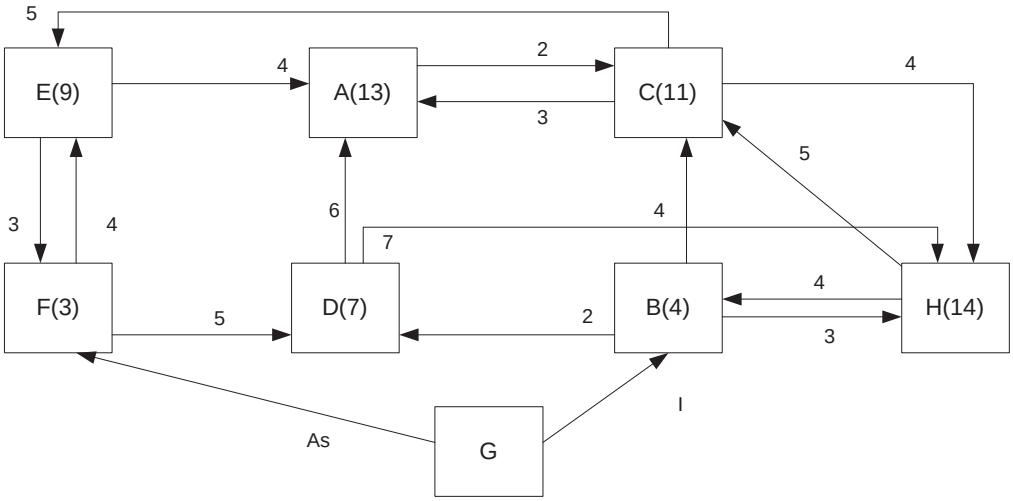

Figure 6.3: WORD - Node Weight is the **Sum** of Incoming Edge Weights (Algorithm 2, case 1)

weights. In an optimistic approach, all the user classes will need the exact same stub functionality, so one stub can satisfy all user classes. This situation is modeled by case (2), where the node weight is the **maximum** of incoming edge weights. In most situations, the reality is probably in between. So case (3) models the situation where the node weight is between the maximum and the sum of the incoming edge weights. Which choice to make depends on domain knowledge and probably needs to be made by the tester. Figures 6.3 and 6.4 show node weights for case 1 and 2.

Case 1 is the same as considering all edges only, and thus no further explanation is given. Cases 2 and 3 are illustrated through examples.

Case 2:

The node weight is defined as the maximum incoming edge weight. Table 6.5 shows the results from applying steps 4 through 7 of algorithm 4 on Figure 6.4. After the initial computation of CWR values for nodes, the algorithm follows the following

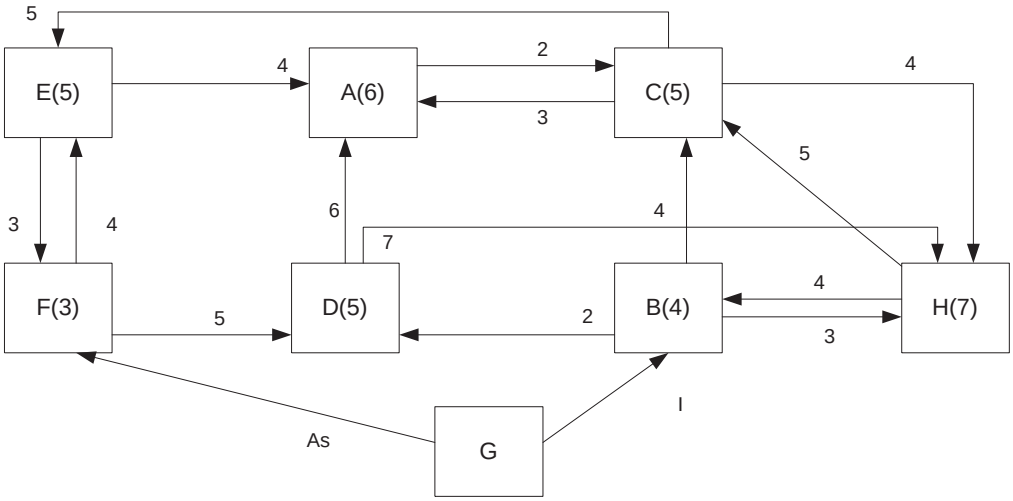

Figure 6.4: WORD - Node Weight is the **Maximum** of Incoming Edge Weights (Algorithm 2, case 2)

Table 6.5: **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\}$ **in Figure 6.4**

No.	Node	Wt.	Cycles Involved	NC	CWR
1	A	6	$\{2, 3, 6, 7\}$	4	0.67
2	B	4	$\{5, 6, 9, 10, 11\}$	5	1.25
3	C	5	$\{2, 3, 4, 5, 6, 7, 8, 9\}$	8	1.6
4	D	5	$\{3, 4, 5, 6, 11\}$	5	1
5	E	5	$\{1, 2, 3, 4, 5\}$	5	1
6	F	3	$\{1, 3, 4, 5\}$	4	1.33
7	H	7	$\{4, 5, 8, 9, 10, 11\}$	6	0.86

steps:

A. Choose a node with maximum cycle-weight ratio in table 6.5 and remove that node from the *WORD*. The node with maximum CWR is C with a CWR of

Table 6.6: **Cycle-weight Ratio for Nodes in $SCC\{E, A, C, H, B, D, F\}$ -{C} in Figure 6.4**

No.	Node	Wt.	Cycles Involved	NC	CWR
1	A	6	$\{\}$	0	0
2	B	4	$\{10, 11\}$	2	0.5
4	D	5	$\{11\}$	1	0.2
5	E	5	$\{1\}$	1	0.2
6	F	3	$\{1\}$	1	0.33
7	H	7	$\{10, 11\}$	2	0.29

1.6. Removing C breaks eight cycles, 2, 3, 4, 5, 6, 7, 8, and 9, leaving three cycles.

B. Recompute the cycle-weight ratio for the remaining edges in table 6.5. The result is shown in table 6.6.

   The node with maximum CWR value is node B. Removing node B breaks two cycles, 10 and 11, leaving cycle 1.

C. Recompute the cycle-weight ratio for the remaining nodes in table 6.6. The result is shown in table 6.7. The node with the maximum cycle-weight ratio is F. Removing F breaks cycle 1 and makes the *WORD* acyclic.

Thus, removing three nodes, C, B, and F, made the graph acyclic with a total cost of 12. This is a lower cost than with algorithm 1 (cost of 13).

Case 3:

The node weight is assumed to be between the maximum and the sum of the incoming edge weights. Table 6.8 shows the result from steps 4 to 7 of algorithm 4 on Figure

Table 6.7: **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\}$ **-{C, B} in Figure 6.4**

No.	Node	Wt.	Cycles Involved	NC	CWR
1	A	6	{}	0	0
4	D	5	{}	0	0
5	E	5	{1}	1	0.2
6	**F**	3	{1}	1	0.33
7	H	7	{}	0	0

6.5. After the initial computation of CWR values for nodes, the algorithm works in the following steps:

A. Choose a node with maximum cycle-weight ratio in table 6.8 and remove that node from the *WORD*. Both C and F have the same maximum CWR. Our rule in this situation is to choose the node that is involved in more cycles (node C). Removing C breaks eight cycles, 2, 3, 4, 5, 6, 7, 8, and 9, leaving three cycles.

B. Recompute the cycle-weight ratio for the remaining edges in table 6.8. The result is shown in table 6.9. The node with maximum CWR value is node B. Removing node B breaks two cycles, 10 and 11, leaving one cycle.

C. Recompute the cycle-weight ratio for the remaining nodes in table 6.9. The node with maximum cycle-weight ratio is F. Removing F breaks cycle 1 and makes the *WORD* acyclic.

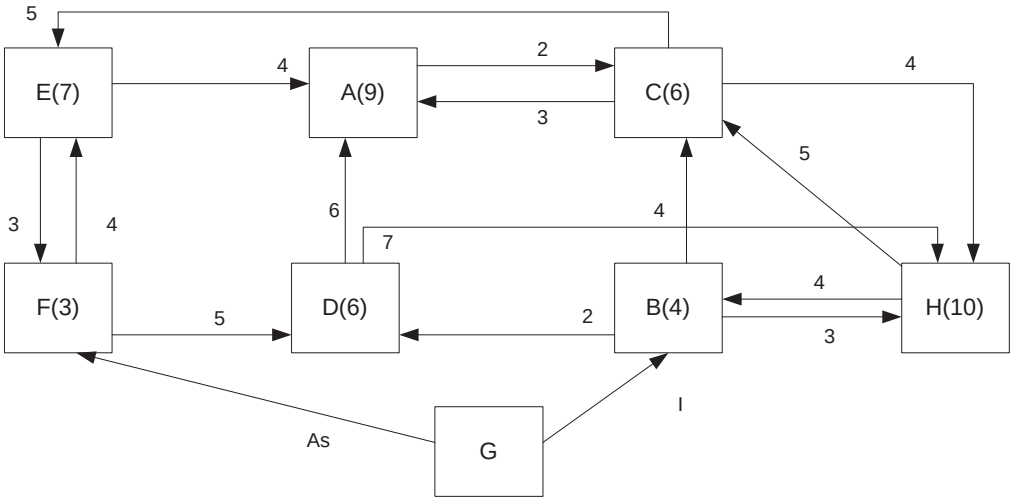

Figure 6.5: WORD - Node Weight is **between the Maximum and Sum** of Incoming Edge Weights (Algorithm 2, case 2)

Table 6.8: **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\}$ **in Figure 6.5**

No.	Node	Wt.	Cycles Involved	NC	CWR
1	A	9	$\{2, 3, 6, 7\}$	4	0.67
2	B	4	$\{5, 6, 9, 10, 11\}$	5	1.25
3	C	6	$\{2, 3, 4, 5, 6, 7, 8, 9\}$	8	1.33
4	D	6	$\{3, 4, 5, 6, 11\}$	5	0.83
5	E	7	$\{1, 2, 3, 4, 5\}$	5	0.71
6	F	3	$\{1, 3, 4, 5\}$	4	1.33
7	H	10	$\{4, 5, 8, 9, 10, 11\}$	6	0.60

Thus, we break all 11 cycles by removing three nodes, C, B, and F. The total cost is 6+4+3 = 13. In this example, the cost is the same as the cost of using algorithm 3.

Table 6.9: **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\} - \{C\}$

No.	Node	Wt.	Cycles Involved	NC	CWR
1	A	9	$\{\}$	0	0
2	B	4	$\{10, 11\}$	2	0.5
4	D	6	$\{11\}$	1	0.17
5	E	7	$\{1\}$	1	0.14
6	F	3	$\{1\}$	1	0.33
7	H	10	$\{10, 11\}$	2	0.20

Table 6.10: **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\} - \{C, B\}$

No.	Node	Wt.	Cycles Involved	NC	CWR
1	A	9	$\{\}$	0	0
4	D	6	$\{\}$	0	0
5	E	9	$\{1\}$	1	0.11
6	F	3	$\{1\}$	1	0.33
7	H	10	$\{\}$	0	0

6.2.6 Heuristics Algorithm for Breaking Cycles Using Node and Edge Weights

This section presents an algorithm for breaking cycles using weight assignments on both nodes and edges. The algorithm is shown in Algorithm 5 and illustrated through examples in Figures 6.4 and 6.5.

Recall that a node can have one of three possible weights. The first, where the node weight is equal to the sum of the incoming edge weights, gives the same result as

Algorithm 5 Eliminating Cycles in WORD (V,E) Using Node and Edge Weights

1: Find all SCCs in WORD
2: **for** (each $scc_i(V_{scc_i}, E_{scc_i}) \in SCCs$) **do**
3: find all cycles CYCLES (totalCycles)
4: **for** (each $v \in V_{scc_i}$) **do**
5: find the number of cycles that use v ($cardinal\{cycles - through - v\}$)
6: compute the cycle-weight ratio
7: **end for**
8: **for** (each $e \in E_{scc_i}$) **do**
9: find the number of cycles that use e ($cardinal\{cycles - through - e\}$)
10: compute the cycle-weight ratio
11: **end for**
12: **while** (totalCycles != 0) **do**
13: order all *nodes* and *edges* in descending order of their cycle-weight ratio
14: remove the *node* or *edge* with the highest cycle-weight ratio
15: totalCycle = totalCycle - number of cycles broken
16: update the number of cycles that use v ($cardinal\{cycles - through - v\}$) or that use e ($cardinal\{cycles - through - e\}$) in the remaining node and edge sets
17: recompute the cycle-weight ratio for the remaining nodes and edges
18: **end while**
19: **end for**

considering only edge weights. Hence, two cases are considered: (1) node weights are the maximum of incoming edge weights, and (2) node weights are between the sum and maximum of incoming edge weights. Figures 6.4 and 6.5 are used for illustration.

Figure 6.4 shows node weights as the maximum of incoming edge weights. Previous tables 6.1 and 6.5 provide initial CWR values for edges and nodes. The rest of algorithm 5 works as follows:

1. Choose an edge or a node with maximum CWR from both tables 6.1 and 6.5.

Then remove whichever has the greater CWR value from the *WORD*. Among the edges, $A \rightarrow C$ has a maximum CWR value of 2, and among the nodes, C has a maximum CWR value of 1.6. Thus, the edge $A \rightarrow C$ is chosen. Removing edge $A \rightarrow C$ breaks four cycles, 2, 3, 6, and 7, leaving seven.

2. Recompute the cycle-weight ratio for the remaining edges and nodes in tables 6.1 and 6.5. The result for edges is the same as in table 6.2. The result for nodes is shown in table 6.11. Note that the weight of the node associated with the removed edge is also recomputed. The new node weight is reduced by the removed edge weight. From tables 6.2 and 6.11, choose a node or an edge with maximum CWR, which is node C. Removing node C breaks four cycles, 4, 5, 8, and 9, leaving three.

3. Compute the cycle-weight ratio for the remaining edges in table 6.2 and nodes in table 6.11. Note that removing a node also removes all edges associated with the node. The results are shown in tables 6.12 and 6.13. Two edges and one node have the same maximum value of 0.5. Our rule in this situation is to choose the node that is involved in more cycles, in this case B. Removing B breaks two cycles, 10 and 11, leaving one cycle, 1.

4. Compute the cycle-weight ratio for the remaining edges and nodes in tables 6.12 and 6.13. The results are not shown because they are simple and can be seen in tables 6.12 and 6.13. According to the rule, F is removed, completing the cycle removing process.

In conclusion, the graph is made acyclic by removing one edge, $A \rightarrow C$, and three nodes, C, B, and F. The total cost is $2 + 3 + 4 + 3 = 12$. In this example, this is

Table 6.11: **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\} - \{A \rightarrow C\}$

No.	Node	Wt.	Cycles Involved	NC	CWR
1	A	6	$\{\}$	0	0
2	B	4	$\{5, 9, 10, 11\}$	4	1
3	C	3	$\{4, 5, 8, 9\}$	4	1.33
4	D	5	$\{4, 5, 11\}$	3	0.60
5	E	5	$\{1, 4, 5\}$	3	0.60
6	F	3	$\{1, 4, 5\}$	3	1
7	H	7	$\{4, 5, 8, 9, 10, 11\}$	6	0.86

Table 6.12: **Cycle-weight Ratio for Edges in** $SCC\{E, A, C, H, B, D, F\}$ **-**$\{A \rightarrow C, C\}$

No.	Node	Wt.	Cycles Involved	NC	CWR
3	$B \rightarrow D$	2	{ 11 }	1	0.5
4	$B \rightarrow H$	3	{ 10}	1	0.33
8	$D \rightarrow A$	6	{ }	0	0
9	$D \rightarrow H$	7	{11}	1	0.14
10	$E \rightarrow A$	4	{ }	0	0
11	$E \rightarrow F$	3	{1 }	1	0.33
12	$F \rightarrow D$	5	{ }	0	0
13	$F \rightarrow E$	4	{1}	1	0.25
14	$H \rightarrow B$	4	{10,11 }	2	0.5

the same cost as using node weights as the maximum of incoming edge weights and not considering edge weights.

Table 6.13: **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\} - \{A \rightarrow C, C\}$

No.	Node	Wt.	Cycles Involved	NC	CWR
1	A	6	$\{\}$	0	0
2	B	4	$\{10, 11\}$	2	0.5
4	D	5	$\{11\}$	1	0.20
5	E	5	$\{1\}$	1	0.20
6	F	3	$\{1\}$	1	0.33
7	H	7	$\{\}$	0	0

Figure 6.5 shows node weights for case 3, between the maximum and the sum of incoming edge weights. Previous tables 6.1 and 6.8 provide initial CWR values for edges and nodes. The rest of algorithm 5 works as follows:

1. Choose an edge or a node with maximum CWR from both tables 6.1 and 6.8. Then remove whichever has the greater CWR value from the *WORD*. Among the edges, $A \rightarrow C$ has a maximum CWR value of 2, and among the nodes, C and F have a maximum CWR value of 1.33. Thus, edge $A \rightarrow C$ is removed. Removing edge $A \rightarrow C$ breaks four cycles, 2, 3, 6, and 7, leaving seven.

2. Recompute the cycle-weight ratio for the remaining edges and nodes in tables 6.1 and 6.8. The result for edges is the same as in table 6.2. The result for nodes is shown in table 6.14. Note that the weight of the node associated with the removed edge is also recomputed. The new node weight is reduced by the removed edge weight. From tables 6.2 and 6.14, choose a node or an edge with maximum CWR value. When more than one node or edge has the same

maximum CWR value, our rule is to choose a **node** that is involved in the larger number of cycles. B and C have the same weight and are also involved in the same number of cycles. In this situation, we arbitrarily chose B. Removing B breaks four cycles, 5, 9, 10, and 11, leaving three.

3. Recompute the cycle-weight ratio for the remaining edges in table 6.2 and nodes in table 6.14. Note that removing a node also removes all edges associated with the node. The results are shown in tables 6.15 and 6.16. Node F is removed, which breaks two cycles, 1 and 4, leaving one cycle, 8.

4. Recompute the cycle-weight ratio for the remaining edges and nodes in tables 6.15 and 6.16. The results are not shown because they are simple and can be seen in tables 6.15 and 6.16. According to the rule, C is removed, and this completes the cycle removing process.

In conclusion, the graph is made acyclic by removing one edge, $A \rightarrow C$, and three nodes, B, F, and C. The total cost is $2 + 4 + 3 + 4 = 13$. This is the same cost as using edge weights only.

In conclusion, using both node and edge weights are similar to using node weights only. This is because node weights are computed using edge weights.

6.3 Algorithm for Ordering Classes for Integration Testing

Once cycles are broken by automation, the integration tester needs a specific ordering of the classes, especially for classes that appear in different SCCs [Tar72]. Algorithm 6 describes an approach for ordering classes for integration and testing. Although

Table 6.14: **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\} - \{A \rightarrow C\}$

No.	Node	Wt.	Cycles Involved	NC	CWR
1	A	9	$\{\}$	0	0
2	B	4	$\{5, 9, 10, 11\}$	4	1
3	C	4	$\{4, 5, 8, 9\}$	4	1
4	D	6	$\{4, 5, 11\}$	3	0.50
5	E	7	$\{1, 4, 5\}$	3	0.43
6	F	3	$\{1, 4, 5\}$	3	1
7	H	10	$\{4, 5, 8, 9, 10, 11\}$	6	0.60

Table 6.15: **Cycle-weight Ratio for Edges in** $SCC\{E, A, C, H, B, D, F\}$ **-**$\{A \rightarrow C, B\}$

No.	Node	Wt.	Cycles Involved	NC	CWR
5	$C \rightarrow E$	5	{4 }	1	0.2
6	$C \rightarrow A$	3	{ }	0	0
7	$C \rightarrow H$	4	{8 }	1	0.25
8	$D \rightarrow A$	6	{ }	0	0
9	$D \rightarrow H$	7	{4}	1	0.14
10	$E \rightarrow A$	4	{ }	0	0
11	**E → F**	**3**	**{1,4}**	**2**	**0.67**
12	$F \rightarrow D$	5	{4}	1	0.2
13	$F \rightarrow E$	4	{1}	1	0.25
15	$H \rightarrow C$	5	{4, 8}	2	0.4

this algorithm is not particularly hard to develop, no algorithm has been published for this problem. The algorithm first generates a *precedence table* for nodes in the

Table 6.16: **Cycle-weight Ratio for Nodes in** $SCC\{E, A, C, H, B, D, F\} - \{A \rightarrow C, B\}$

No.	Node	Wt.	Cycles Involved	NC	CWR
1	A	9	$\{\}$	0	0
3	C	4	$\{4, 8\}$	2	0.5
4	D	6	$\{4\}$	1	0.17
5	E	7	$\{1, 4\}$	2	0.29
6	F	3	$\{1, 4\}$	2	0.67
7	H	10	$\{4, 8\}$	0	0

WORD, then finds all strongly connected components (SCCs) in the weighted object relation diagram (WORD). A *precedence table* is indexed by the number or name of nodes in the WORD, and shows the nodes that are connected to a node through outgoing edges from the node. Then, each SCC is compressed into a node, and the multiple edges between SCCs are combined into one edge. As a result, an acyclic directed graph $WORD_{comp}$, a compressed WORD, is produced. For example, Figure 6.6 represents a WORD with three SCCs, $\{1, 2\}$, $\{3, 4\}$, and $\{5\}$. Figure 6.7 shows the resulting $WORD_{comp}$. The algorithm finds the reversed topological order, O_{SCCs}, for the $WORD_{comp}$ as (1) $\{5\}$, (2) $\{3, 4\}$, and (3) $\{1, 2\}$.

Then, each scc_i is made acyclic using Algorithm 3, and $scc_{i-acyclic}$ represents the resulting subgraph. The removed edges represent specific test stubs to be developed. Suppose edges $1 \rightarrow 2$ and $3 \rightarrow 4$ are removed from SCCs $\{1, 2\}$ and $\{3, 4\}$. The result is that there are two specific test stubs for 2 and 4. A reverse topological

Algorithm 6 Ordering Classes for Integration and Testing

1: Generate *precedence table* for nodes in the WORD
2: Find all SCCs in WORD
3: Generate an acyclic compressed version of WORD, $WORD_{comp}$, by representing each scc_i as a single node and by compressing multiple edges between every two nodes
4: Find reverse topological order, O_{SCCs}, of nodes in $WORD_{comp}$
5: **for** (each $scc_i \in SCCs$) **do**
6: make scc_i acyclic by using Algorithm 3 and record removed edges
7: find reverse topological order, O_{scc_i}, for nodes in the acyclic $scc_{i-acyclic}$
8: **end for**
9: Start testing according to the order of SCCs in O_{SCCs}
10: **for** (each ordered $scc_i \in SCCs$) **do**
11: test nodes in the order O_{scc_i}
12: **end for**

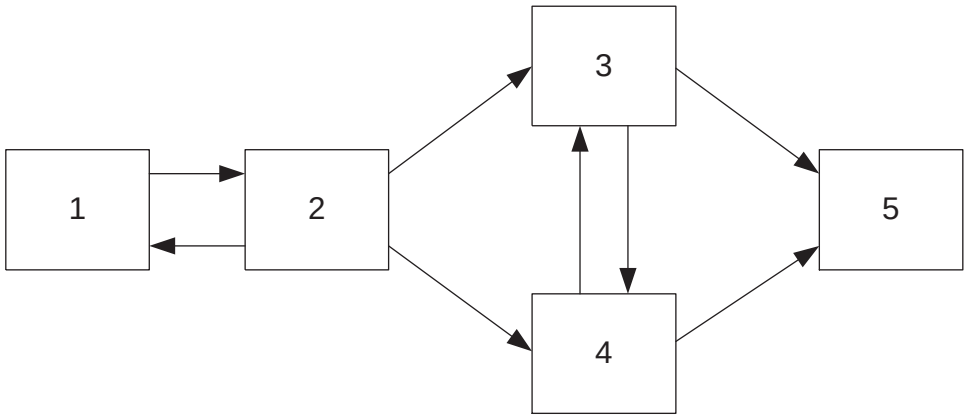

Figure 6.6: Finding Overall Test Order in a WORD

order, O_{scc_i}, is generated for $scc_{i-acyclic}$. In this example, SCCs $\{1, 2\}$ and $\{3, 4\}$ have reverse topological orders of 1, 2 and 3, 4. Testing starts according to the order O_{SCCs}. For each node in O_{SCCs}, first, $scc_{i-acyclic}$ is restored, and included nodes are

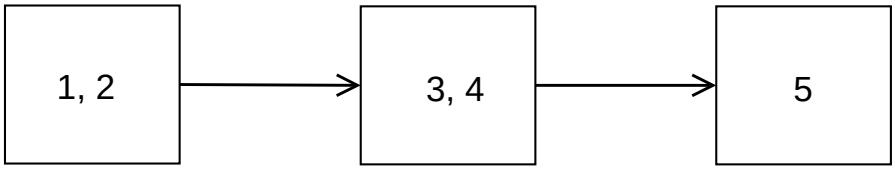

Figure 6.7: Compressed WORD

tested according to the order O_{scc_i}. For Figure 6.6, the integration and test order is 5, 3, 4, 1, 2. Before a node is tested, the *precedence table* is checked. If a node was connected to a removed edge, then include the corresponding test stub in the test order. For example, when node 3 is tested, the precedence table indicates that node 3 is connected to an untested node. Thus, the test stub for node 4 is included at this point.

6.4 Case Study

This section provides an evaluation of the model and algorithms by comparing results with the same project, the ATM system, used by Briand et al. [BFL02]. Briand et al. chose the number of broken dependencies, attribute couplings, method couplings, and a combination of attributes and methods as four cost functions to produce an integration test order, and compared the results to decide which cost function gives the best result. Our approach is first to use dependencies, attribute coupling measures, method coupling measures, and a combination of attribute and method coupling measures as weights on edges and apply our algorithm to check what kind of result can be obtained under similar situations. Then, we collect coupling data from the implementations using coupling definitions and coupling measures defined in Section 6.2.1, construct the weighted object relation diagram (WORD), and compute the

Table 6.17: **Coupling Measures for Edges in** SCC $\{8, 9, 10, 11, 12, 13, 14, 15\}$

No.	8	9	10	11	12	13	14	15
8		1.0.0.0.0	1.0.2.1.3					
9	1.0.1.1.3							
10	1.0.6.4.4	1.0.1.1.1		1.0.3.5.3	1.0.1.1.0	10.1.1.0	1.0.1.1.0	1.0.1.1.0
11	1.0.1.1.3	1.0.0.0.0	1.0.1.1.0					
12	1.0.4.3.11	1.0.4.3.11	1.0.2.2.0	5.0.0.0.0				
13	1.0.4.3.6	1.0.4.3.14	1.0.2.2.0	5.0.0.0.0				
14	1.0.3.2.6	1.0.3.3.12	1.0.2.2.0	5.0.0.0.0				
15	1.0.2.1.6	1.0.3.3.10	1.0.2.2.0	5.0.0.0.0				

edge weights for SCCs in the WORD using equations 6.3 and 6.8.

The ATM system has 21 classes and eight form a strongly connected component that has 30 cycles [BFL02]. Table 6.17 shows the coupling measures in the format of equation 6.1. Table 6.18 shows the different edge weights that are used in this evaluation. In particular, the columns labeled Dependency, # of Attributes, # of Methods, and A & M show the edge weights obtained from Briand et al.'s four cost functions. The last column shows edge weights that are computed from Table 6.17 using equations 6.5, 6.3, 6.4, and 6.8. The constraints of not breaking inheritance and composition edges are achieved by assigning 5 to variable CBT in equation 6.1 for inheritance and composition, and 1 for the others.

In all five approaches, seven dependencies were removed. When using weights in the columns labeled # of Attributes, # of Methods, A & M, and A & M-new of Table 6.18, exactly the same set of edges were removed. Hence, the stubbing cost for these approaches are equal. Although seven dependencies were broken when using the existence of dependencies as a cost function, the stubbing cost may vary because the edge weights are the same and thus cannot reflect any stubbing cost at the time of deciding to choose an edge to remove between two equal weight edges.

Table 6.18: **Different Weights for Edges in** SCC $\{8, 9, 10, 11, 12, 13, 14, 15\}$

No.	Edge	Dep.	# of Attr.	# of Meth.	A & M	A & M-new
1	$8 \rightarrow 9$	1	13	1	0.71	1
2	$8 \rightarrow 10$	1	9	2	0.53	1.22
3	$9 \rightarrow 8$	1	13	7	1	1.17
4	$10 \rightarrow 8$	1	13	7	1	1.66
5	$10 \rightarrow 9$	1	13	2	0.74	1.13
6	$10 \rightarrow 11$	1	0	0	0	1.57
7	$10 \rightarrow 12$	1	2	2	0.23	1.13
8	$10 \rightarrow 13$	1	2	2	0.23	1.13
9	$10 \rightarrow 14$	1	3	2	0.26	1.13
10	$10 \rightarrow 15$	1	1	2	0.21	1.13
11	$11 \rightarrow 8$	1	13	2	0.74	1.17
12	$11 \rightarrow 9$	1	13	1	0.71	1.00
13	$11 \rightarrow 10$	1	9	2	0.53	1.13
14	$12 \rightarrow 8$	1	13	4	0.81	1.60
15	$12 \rightarrow 9$	1	13	4	0.81	2.59
16	$12 \rightarrow 10$	1	9	2	0.53	2.01
17	$12 \rightarrow 11$	1	∞	∞	∞	5.00
18	$13 \rightarrow 8$	1	13	4	0.81	1.50
19	$13 \rightarrow 9$	1	13	4	0.81	2.62
20	$13 \rightarrow 10$	1	9	2	0.53	2.01
21	$13 \rightarrow 11$	1	∞	∞	∞	5.00
22	$14 \rightarrow 8$	1	13	3	0.77	1.39
23	$14 \rightarrow 9$	1	13	3	0.77	2.58
24	$14 \rightarrow 10$	1	9	2	0.53	2.01
25	$14 \rightarrow 11$	1	∞	∞	∞	5.00
26	$15 \rightarrow 8$	1	13	2	0.74	1.29
27	$15 \rightarrow 9$	1	13	3	0.77	2.58
28	$15 \rightarrow 10$	1	9	2	0.53	2.01
29	$15 \rightarrow 11$	1	∞	∞	∞	5.00

The results indicate that when we consider the stub complexity as weights on edges, graph-based algorithms can produce results as good as those produced by genetic algorithms, but with significantly lower costs. A larger empirical evaluation

needs to be carried out to verify that this result generalizes.

6.5 Summary of Existing Graph Algorithms for Cycle Elimination

For a given directed graph $G = (V, E)$, the problem of eliminating cycles with minimum cost has been formulated as the *maximum acyclic subgraph problem* [BS90], the *minimum feedback arc set problem* [ELS93], and the *minimum feedback vertex set problem* [LJ00]. The problem is known to be NP-complete on general graphs and on bipartite graphs.

Given a directed graph $G = (V, E)$, the *maximum acyclic subgraph problem* is to identify the minimum subset E' in E such that $G' = (V, E')$ is acyclic and E' has maximum cardinality [BS90].

Given a directed graph $G = (V, E)$, the *feedback arc set problem* is to determine a minimum cardinality set of arcs that breaks all cycles [ELS93].

Given a directed graph $G = (V, E)$, the *minimum feedback vertex set (MFVS)* problem is to identify the minimum subset V' in V such that after removing all vertices in V', the remaining graph is acyclic [LJ00].

Lin and Jou [LJ00] developed an algorithm for the MFVS problem, which can be used to improve our heuristic algorithms for the CITO problem. Given a directed, node-weighted graph $G = (V, E)$, Lin and Jou used ($\alpha\times$ (number of Π-edges) + $\beta\times$ (in-degree + out-degree))/(vertex cost) as the weight function for a vertex. The Π-edges are defined as follows.

Π-*edge Definition:* Given a directed graph $G = (V, E)$ and an edge

(u, v) in E, the edge (u, v) is a Π-edge if there also exists an edge (v, u) in E.

Our heuristic algorithms can be improved by using the Π-edge concept. First, the Π-edges are ordered in descending order of their cycle-weight ratio values. The Π-edge with the maximum CWR is removed, then CWR is re-computed for the remaining Π-edges. This process is repeated until there are no more Π-edges. After this step, our original algorithms can be used as defined in Section 6.2.

6.6 Summary

This chapter presents an improved technique and algorithms to automate the CITO problem. The technique uses weights to represent the cost of creating stubs. This has been done before, but the weights in this research are derived from quantitative analysis of couplings, thus obtaining more precise results. These weights are placed on a *Weighted Object Relation Diagram (WORD)*, which represents classes as nodes and relationships as edges. This chapter also introduces the idea of applying weights to nodes to estimate the cost of removing the nodes. If a class is used by multiple classes, then all or part of the same stub for that class may be shared among all classes that use it, thus reducing the cost of stubbing. The weight of a node is at least as high as the maximal weight of all incoming edges (assuming total sharing of the stub), and no higher than the sum of the weights of all incoming edges (assuming no sharing of the stub).

New algorithms to solve the CITO problem are introduced. These algorithms use edge and node weights. They were compared with algorithms by previous researchers, and found to be just as effective if edge weights are ignored. They can be more effective

if edge weights are used. Overall, the results in this research improve the ability for developers to automate the CITO problem.

Chapter 7: COUPLING-BASED CHANGE IMPACT ANALYSIS

Change is an inherent and necessary part of a software system's life. The importance of change is reflected in the distribution of software costs. Estimates show that 65-75% of total software costs are subsumed in maintenance activities [Som95]. Warren [War99] states that software systems change for two reasons: (1) the environment in which a system operates is dynamic, and (2) software development invariably introduces errors. As software systems become increasingly large and complex, it becomes more necessary to predict and control the effects of software changes. Studies over the last three decades have shown that making software changes without fully understanding their effects can lead to poor effort estimates, delays in release schedules, degraded software design, unreliable software products, and premature retirement of software systems [BA96].

Key aspects of change impact analysis include identifying possible changes in a system, defining the concept of an *impact*, developing algorithms for computing the impact set of a proposed change, and defining cost and effort prediction models for implementing changes. Once changes are identified and their impacts are defined, computing impact sets becomes a critical part of the analysis activity. Three different approaches to computing the impact set of a software change are *qualitative*, *quantitative*, and *theoretical*. The *qualitative* approach computes the impact set from intuition of developers or a manager. The *quantitative* approach computes the impact

set from information from software design, program source code, and dynamic run time information. The *theoretical* approach tries to compute the theoretical minimum impact set. It relies on determining whether definitions reach their uses. This problem can be reduced to the halting problem, which is undecidable. Hence, finding a theoretical minimum is not possible.

Precision and size are two main factors to consider in change impact set computation. A precisely computed impact set of a change includes the exact elements that will be affected by the change. The size of a change impact set is the number of elements that will be affected by the proposed change. The elements of a change impact set are classes, methods, and statements. The goal is to compute the exact change impact set so that the maintenance model based on the change impact set is precise. An impact set that is computed for a change could look like one of the sets shown in Figure 7.1. The *exact set* includes only those elements that are truly affected by a change. Set 1 includes the exact set and some other elements that will not be affected by a change. Set 2 includes part of exact set and some other elements that will not be affected by a change. Since the theoretical minimum is impossible to compute, the goal of an algorithm used to compute change impact set is to compute Set 1 with as small a size as possible. The size of an impact set is important because it will be used as a prediction model for the cost and effort of making a change in the system. If the size of a computed impact set is bigger than the actual size of the impact set then more cost and effort is required to accomplish the change. However, it is difficult to decide whether a set includes the impact set.

If we refine the aforementioned approaches to include more details, we get the following five possible ways to compute impact sets:

1. Qualitative-based on intuition. A person **estimates** an impact set that includes

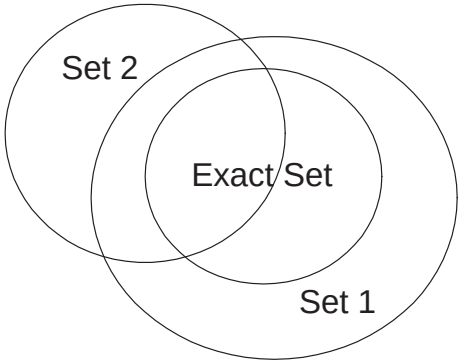

Figure 7.1: Change Impact Sets

all components that could possibly be impacted by a change. The size of this type of computed impact set is usually the largest, but depends to a large degree on the person making the estimate.

2. Quantitative-based on design analysis. This approach computes the impact set based on the interactions among components in the **design** of a system. This set is smaller than the qualitative set mentioned above.

3. Quantitative-based on program static analysis. In this approach, program source code is **statically** analyzed to extract the detailed interaction information among components of a system. Although the computed impact set will be smaller than if based on design information, it cannot take run time interactions into consideration.

4. Quantitative-based on run time dynamic information. The computation considers both static and **run time** information. This allows the computed impacted set to be smaller.

5. Theoretical minimum impact set-relies on whether **definitions** reach **uses**. This

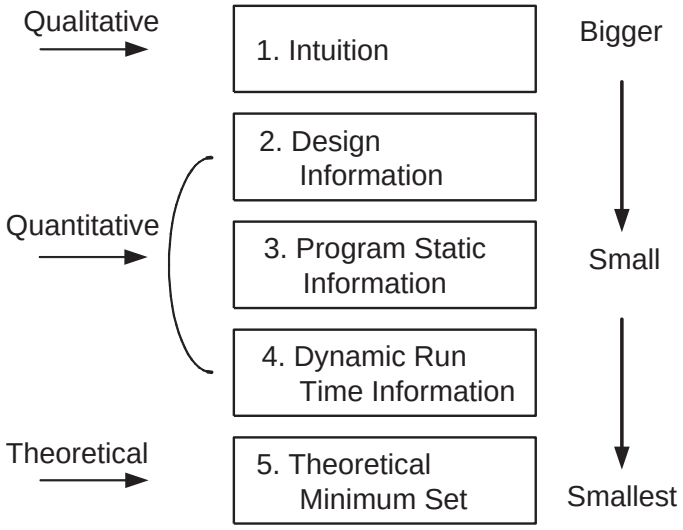

Figure 7.2: Change Impact Analysis

problem can be reduced to the halting problem, which is undecidable. Hence, the theoretical minimum is not reachable.

Figure 7.2 summarizes these five approaches. Previous research has been on the first and second approaches, and less with the third approach. This chapter presents research based on the third approach; static program analysis. We will consider the fourth approach in the future.

Coupling analysis is a powerful tool for analyzing interactions among components in software. Because changes in one component propagate to other components only through interactions, the results from coupling analysis can be used for change impact analysis, saving time and effort.

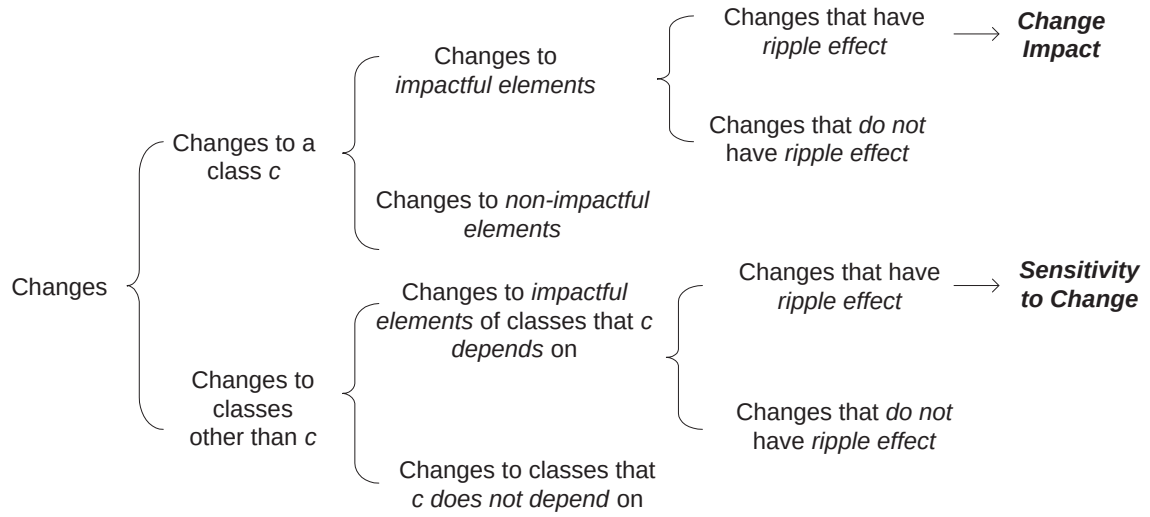

Figure 7.3: Change Categories

7.1 Class Change Impact and Change Sensitivity

When one class requires another to be available before objects of that class can be compiled, we call this relationship a *dependency* between a *server* and a *client* class. The client class is being compiled or executed and the server class must be present. This dependency has a direction, and, at a more detailed level, is based on one or more object-oriented relationships.

Existing systems are modified for the purpose of corrective, adaptive, perfective, or preventive maintenance. One problem in modifying an existing system is that small changes can ripple through the software and have unintended impacts elsewhere. Figure 7.3 categorizes changes according to their potential effects on client classes. First, not all changes cause ripple effects. For example, if the value of a variable changes within its bounds, this change should not impact other classes. However, if the access specifier of a variable changes from *public* to *private*, then this change may

have ripple effects.

> **Definition 1**: If a change can have a ripple effect, this type of change is a **causal change**.

When a *causal change* is made in a class c, the change can only possibly affect c's clients. Furthermore, only changes to those elements of c that are used by others can impact its client classes. We define this to be the set of *impactful elements* of c, IE_c. IE_c determines the *change impact* of c in the system. *Change Impact* is a metric that measures a class's overall impact on other classes. In addition to changes to itself, class c can be affected by changes in classes that it depends on. If c depends on k classes, c_1, c_2, ..., c_k, only changes to elements in $\cup_{i=1}^{k} IE_{c_i}$ can affect c. Hence, $\cup_{i=1}^{k} IE_{c_i}$, c_1, c_2, ..., c_k, can be said to determine the *sensitivity* of c to changes in other classes. *Sensitivity* is a metric that measures the *reactiveness* of a class to changes in the system.

In this research, dependencies among classes are estimated using couplings. To distinguish directions of dependencies, a class's interactions with other classes are measured using two sets of couplings.

> **Definition 2**: Let C be the set of classes in a system, and $c \in C$. Let $Client(c)$, $Client(c) \subset C$, be the set of classes that uses IE_c. The coupling measure $cm_{c_{in}}$ represents the ***incoming coupling*** of c.

> **Definition 3**: Let C be the set of classes in a system, and $c \in C$. Let $Server(c)$, $Server(c) \subset C$, be the set of classes that c uses as servers. The coupling measure $cm_{c_{out}}$ represents the ***outgoing coupling*** of c.

A class's coupling to its clients determines its *change impact*. Couplings from server classes determine the *sensitivity* of a class to changes in the system. In other words, the *incoming couplings* define the change impact of a class, and the *outgoing couplings* define a class's sensitivity to changes in other classes.

7.2 Measuring Class Change Impact and Class Change Sensitivity

This section uses the coupling measures defined in Chapter 3. Chapter 3 defined the incoming couplings of a class c, $cm_{c_{in}}$, from k clients $(d_1..d_k)$ as follows

$$\begin{aligned} cm_{c_{in}} &= \{\sum_{i=1}^{k} CM(d_i, c) \mid d_i, c \in C\} \\ &= \sum_{i=1}^{k} CBT_{d_i,c}.\sum_{i=1}^{k} V_{d_i,c}.\sum_{i=1}^{k} M_{d_i,c}.\sum_{i=1}^{k} R_{d_i,c}.\sum_{i=1}^{k} P_{d_i,c} \\ &= CBT_{in}.V_{in}.M_{in}.R_{in}.P_{in} \end{aligned} \tag{7.1}$$

and the outgoing couplings of a class c, $cm_{c_{out}}$, to k servers $(d_1..d_k)$ as follows

$$\begin{aligned} cm_{c_{out}} &= \{\sum_{i=1}^{k} CM(c, d_i) \mid c, d_i \in C\} \\ &= \sum_{i=1}^{k} CBT_{c,d_i}.\sum_{i=1}^{k} V_{c,d_i}.\sum_{i=1}^{k} M_{c,d_i}.\sum_{i=1}^{k} R_{c,d_i}.\sum_{i=1}^{k} P_{c,d_i} \\ &= CBT_{out}.V_{out}.M_{out}.R_{out}.P_{out} \end{aligned} \tag{7.2}$$

The sets that contain all incoming and outgoing coupling base types of a class c are denoted as $R_{c_{in}}$ and $R_{c_{out}}$, and are computed as follows:

$$R_{c_{in}} = \{CBT_{d_i,c} \mid d_i, c \in C, i = 1...k\} \tag{7.3}$$

$$R_{c_{out}} = \{CBT_{c,d_i} \mid c, d_i \in C, i = 1...k\} \tag{7.4}$$

For change impact analysis, we use the measures with the individual counting options. This means the V, M, R, and P measures are computed from the $\mathcal{A}_{\mathcal{T}}$ (Equation 3.2 from Chapter 3), $\mathcal{M}_{\mathcal{T}}$ (Equation 3.4 from Chapter 3), $\mathcal{RV}_{\mathcal{T}}$ (Equation 3.6 from Chapter 3), and $\mathcal{P}_{\mathcal{T}}$ (Equation 3.8 from Chapter 3) multisets. The rationale is that a change to an element will impact all (repeated) uses of that element. In addition, we consider the number of coupling base types as one measure. This is because a server class will impact its clients even if its methods or variables are not used. We define T_{in} as the number of incoming coupling base types, $T_{in} = |R_{in}|$, and T_{out} as the number of outgoing coupling base types, $T_{out} = |R_{out}|$.

Thus, $cm_{c_{in}}$ and $cm_{c_{out}}$ become

$$cm_{c_{in}} = T_{in}.V_{in}.M_{in}.R_{in}.P_{in} \tag{7.5}$$

$$cm_{c_{out}} = T_{out}.V_{out}.M_{out}.R_{out}.P_{out} \tag{7.6}$$

To estimate change impact and sensitivity of a class from the total incoming and outgoing coupling measures, we employ Briand et al.'s method [BFL02]. For a measure $Cplx()$, a complexity measure $\overline{Cplx()}$ is normalized as

$$\overline{Cplx(i,j)} = Cplx(i,j)/(Cplx_{max} - Cplx_{min}) \tag{7.7}$$

where $Cplx(i,j)$ represents a complexity information matrix, $Cplx_{min} = Min\{Cplx(i,j), i,j = 1,2,...\}$ and $Cplx_{max} = Max\{Cplx(i,j),\ i,j = 1,2,...\}$. Briand et al. used two coupling measures, A() and M(), the number of locally defined variables and the number of methods, to compute overall stubbing complexity in testing:

$$SCplx(i,j) = (W_A.\overline{A}(i,j)^2 + W_M.\overline{M}(i,j)^2)^{1/2} \tag{7.8}$$

where W_A and W_M are weights and $W_A + W_M = 1$.

We use Briand et al.'s concept [BFL02] in computing the change impact and sensitivity. As shown in equations 7.5 and 7.6, our coupling measures measure the number of coupling base types, the number of parameters, and the number of return value types in addition to the number of variables and the number of methods. We use the same normalization method as Briand et al., and include the additional coupling measures in the stubbing complexity estimation.

Using the five total incoming coupling measures, $T_{in}()$, $V_{in}()$, $M_{in}()$, $R_{in}()$, and $P_{in}()$, the change impact of a class c is estimated as follows:

$$\begin{aligned} Ch_{imp}(c) = (W_T \times \overline{T_{in}}(i,j)^2 + W_V \times \overline{V_{in}}(i,j)^2 + W_M \times \overline{M_{in}}(i,j)^2 + \\ W_R \times \overline{R_{in}}(i,j)^2 + W_P \times \overline{P_{in}}(i,j)^2)^{1/2} \end{aligned} \tag{7.9}$$

where W_T, W_V, W_M, W_R, and W_P are weights and $W_T + W_V + W_M + W_R + W_P = 1$. The $\overline{T_{in}}(i,j)$, $\overline{V_{in}}(i,j)$, $\overline{M_{in}}(i,j)$, $\overline{R_{in}}(i,j)$, and $\overline{P_{in}}(i,j)$ values are computed from equation 7.7 using values from equation 7.5.

Using the five total outgoing coupling measures, $T_{out}()$, $V_{out}()$, $M_{out}()$, $R_{out}()$, and

$P_{out}()$, the sensitivity of a class c to changes in the system is estimated as follows:

$$Ch_{sen}(c) = (W_T \times \overline{T_{out}}(i,j)^2 + W_V \times \overline{V_{out}}(i,j)^2 + W_M \times \overline{M_{out}}(i,j)^2 + W_R \times \overline{R_{out}}(i,j)^2 + W_P \times \overline{P_{out}}(i,j)^2)^{1/2} \quad (7.10)$$

where W_T, W_V, W_M, W_R, and W_P are weights and $W_T + W_V + W_M + W_R + W_P = 1$. The $\overline{V_{out}}(i,j)$, $\overline{M_{out}}(i,j)$, $\overline{R_{out}}(i,j)$, and $\overline{P_{out}}(i,j)$ values are computed from equation 7.7 using values from equation 7.6.

7.2.1 Usefulness of Measures

The *change impact* and *sensitivity* measures can be used by developers in several ways:

1. If a class is known to have a large *change impact*, then developers should try to avoid changing the class.

2. Both measures can be used to evaluate design patterns. For example, a system can be designed using different patterns. Then each pattern can be evaluated according to the *sensitivity* and *change impact* of classes in each pattern.

3. These measures will help developers and maintainers understand the system and its structure.

4. The sensitivity measure of a class can be used to predict its stability. Sensitivity of a class is a snapshot view of the class at a particular moment while stability is measured over a period of time.

5. The *sensitivity* measure can be used to rank classes for their reusability. Classes with lower sensitivity are more desirable candidates for reuse.

7.2.2 Object-Oriented Program Changes

One activity in *change impact analysis* is to identify changes and explore their natures. This research considers source code level changes of object-oriented software. Object-oriented software consists of classes and their relationships. Furthermore, classes are composed of member variables and methods. A class defines variables and the methods common to all objects that are in the type hierarchy defined by the class. For the purposes of corrective, adaptive, perfective, or preventive maintenance, changes can be made to any element of the software: classes, their member variables, their methods, and relationships among classes. While the concepts of class declaration, variables, and methods have representations in the source code, most class relationships do not have explicit actualization. In fact, most class relationships are hidden in the definitions and uses of variables and methods. This research only considers changes to explicit class relationships in the source and thus identifies four categories of changes: changes to **class structure**, changes to **variables**, changes to **methods**, and changes to **class relationships**. Furthermore, only causal changes are identified for each category.

Deleting a variable, *changing the visibility of a change*, and *changing* the type of a variable are considered to be causal changes for variables. For methods, changes such as *adding* or *deleting* a method (could be empty), *changing* the access specifier of a method, *changing* the return type of a method, *changing* the type of any of the parameters of a method, *adding* or *deleting* parameters of a method, *making a method abstract*, and *adding a body to an abstract method* are considered to be causal changes. For class relationships, changes such as *adding* or *deleting* inheritance relationship, *adding or deleting* an implementation, and *adding* or *deleting* import statements are

considered to be causal changes. For class declarations, *adding* a new class, *deleting* a class, *changing* class visibility are considered to be causal changes. *Renaming* is the same as *deleting* and then *adding*, hence, it is not considered separately.

7.2.3 Computing Impact Sets for Individual Changes

Two factors affect the impact of a change [CKKL02]. One is that different types of changes lead to different sets of impacted classes. The other is the type of interaction among components. The interactions can be captured by coupling analysis. By the definition of coupling, a change can only impact those classes that are coupled with the class that is being changed.

> **Definition: Maximum Impact Set**. Let C be the set of classes in a system, and $c \in C$. Let SC_c be the set of classes are coupled with c by using c as a server. For all changes in c, the *maximum impact set* of c, MIS_c, is equal to SC_c.

The set MIS_c includes those classes that can possibly be impacted through ripple effects of changes in c. The size of MIS_c can be large and using it may lead to unnecessary maintenance effort. It is possible to compute a more precise impact set for a particular change, which should be substantially smaller than the MIS_c, by examining the usage of impactful elements of c, IE_c, by each client class. Furthermore, impactful elements of classes can be identified during coupling analysis. If a class c is coupled with k classes, $d_1, d_2, ..., d_k$, and c is a server, the impactful elements of c, IE_c, is

$$IE_c = \cup_{i=1}^{k} \mathcal{A}_{D(d_i,c)} \bigcup \cup_{i=1}^{k} \mathcal{M}_{D(d_i,c)} \bigcup \cup_{i=1}^{k} \mathcal{RV}_{D(d_i,c)} \bigcup \cup_{i=1}^{k} \mathcal{P}_{D(d_i,c)} \tag{7.11}$$

where $\mathcal{A}_{D(d_i,c)}$, $\mathcal{M}_{D(d_i,c)}$, $\mathcal{RV}_{D(d_i,c)}$, and $\mathcal{P}_{D(d_i,c)}$ are sets defined in equations 3.1, 3.3, 3.5, and 3.7 in Chapter 3.

In the JCAT tool, the impactful elements are saved in database tables. Figures 7.4 through 7.6 show tables that are designed for impactful elements. This research computes the impact set of variable and method related changes. Intuitively, variable and method level changes can impact only classes that use the changed variables or invoke the changed methods.

Figures 7.4 and 7.5 show tables about variable definition and variable usage. The variable definition level information is stored in the var_def, var_modifiers, method_var_def, method_var_modifiers, method_parameters, and param_modifiers tables, as shown in Figure 7.4. There are three types of variables: class, method, and parameter. A *class variable* is defined at the class level. A *method variable* is defined in a method. The parameters to a method are saved separately in the method_parameters table to make it convenient for later analysis. The related class_def and method_def tables are also presented to show their relationship with the variable tables.

The *variable usage* level information is stored in the var_use_method and var_use_class tables. Figure 7.5 shows these two tables and other related tables. Since only the class level variables can be shared among classes, the var_def table is shown from the variable level tables. The *method call* level information is stored in the constructor_call_method, constructor_call_class, method_call_class, and method_call_method tables. Figure 7.6 shows the tables and their structure. The shaded tables are the parent tables and the arrows show the relationships among the tables. The method calls are saved in separate tables according to their scope. For example, if a method call occurs inside a method, this event and the related information are saved in the method_call_method table. Similarly, if a method call occurs at the class

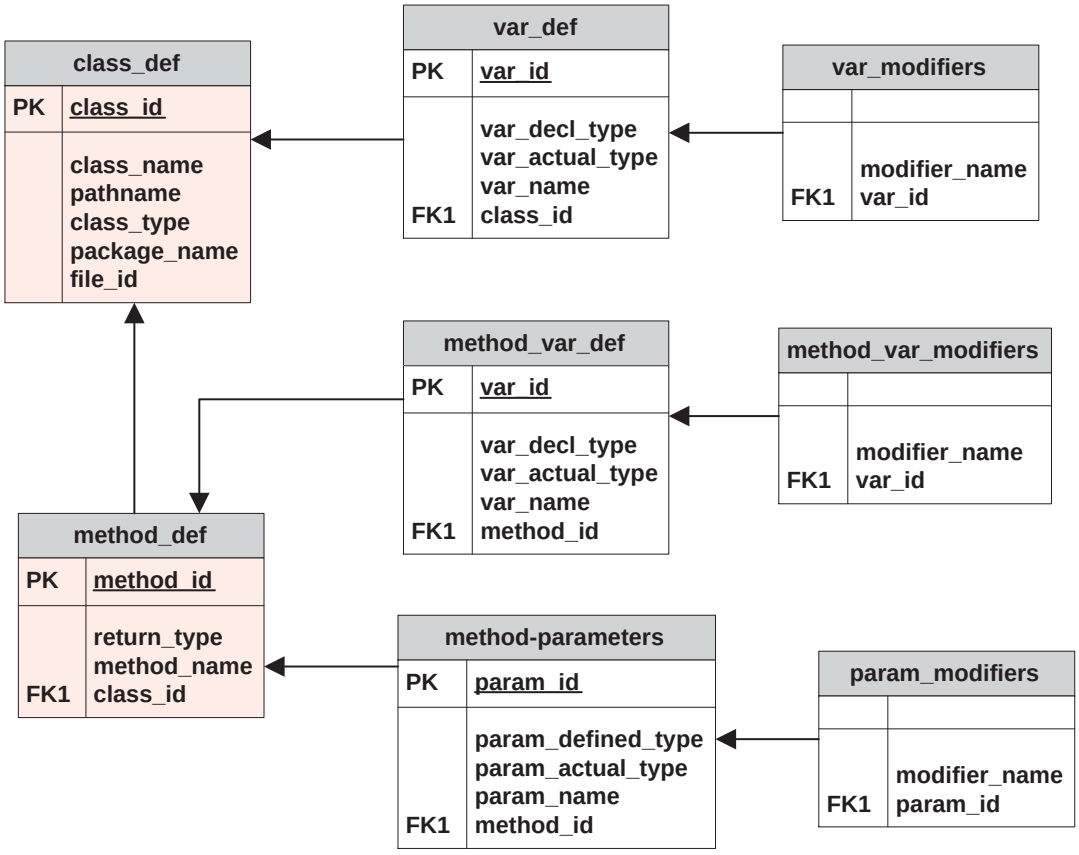

Figure 7.4: Variable Level Tables

level, the information is saved in the method_call_class table. The invocation of a constructor is handled in the same way.

By querying these tables, change impact sets of each element can be precisely computed. Algorithms 7 and 8 describe the impact set computation steps for a variable and method change, including changes to return type and parameters of a method.

$qry1$: SELECT $vd.var\_id$ AS $v_{ch_i}\_id$

FROM $class\_def$ AS c, var_def AS vd

WHERE $c.pathname = c_{ch_i}.pathname$ AND $c.class\_name = c_{ch_i}.name$

Algorithm 7 Find Variable Change Impact Set

Require: The change type, ch_i, the variable to be changed, v_{ch_i}, and the class c_{ch_i} to which v_{ch_i} belongs, are specified.

Ensure: The impact set $IS(v_{ch_i})$ is identified.

1: **if** $cm_{c_{in}} \neq 0$ **then**
2: **if** (ch_i is causal change) AND ($v_{ch_i} \in IE_c$) **then**
3: *execute qry1* (find the ID of v_{ch_i} in database)
4: *execute qry2* (find classes that uses v_{ch_i} in their class space)
5: *execute qry3* (find classes that uses v_{ch_i} in their methods)
6: **end if**
7: **end if**

AND $vd.var\_name = v_{ch_i}.name$ AND $vd.class\_id = c.class\_id$;

$qry2$: SELECT $C.pathname$ AS $impacted\_class\_name$

FROM $class\_def$ AS C, var_use_class AS vuc

WHERE $vuc.var\_id = v_{ch_i}\_id$ AND $vuc.class\_id = C.class\_id$;

$qry3$: SELECT $C.pathname$ AS $impacted\_class\_name$

FROM $class\_def$ AS C, var_use_method AS vum, var_def AS vd,

$method\_def$ AS md

WHERE $vum.method\_id = md.method\_id$ AND $md.class\_id = C.class\_id$

AND $vum.var\_id = v_{ch_i}\_id$;

Algorithm 7 first checks if a change is a causal change, then it checks if the variable to be changed is in the set of impactful elements of class c. If these conditions are met, then the algorithm selects classes that use the variable by making queries to tables that store the impactful variables. $qry1$ identifies the ID of the variable in the table. If a variable is used in the scope of a class, then this information is

stored in the *var_use_class* table; if a variable is used in the scope of a method, then this information is stored in the *var_use_method* table. $qry2$ and $qry3$ make separate queries to these two tables and select classes that use the variable that will be changed.

Algorithm 8 Find Method Change Impact Set

Require: The change type, ch_i, the method to be changed, m_{ch_i}, and the class c to which m_{ch_i} belongs are specified.

Ensure: The impact set $IS(m_{ch_i})$ is identified.

1: **if** $cm_{c_{in}} \neq 0$ **then**
2: **if** (ch_i is causal change) AND ($m_{ch_i} \in IE_c$) **then**
3: *execute qry4* (find the ID of m_{ch_i} in database)
4: *execute qry5* (find classes that calls m_{ch_i} in their class space - m_{ch_i} is a method)
5: *execute qry6* (find classes that calls m_{ch_i} in their methods - m_{ch_i} is a method)
6: *execute qry7* (find classes that calls m_{ch_i} in their class space - m_{ch_i} is a constructor)
7: *execute qry8* (find classes that calls m_{ch_i} in their methods - m_{ch_i} is a constructor)
8: **end if**
9: **end if**

$qry4$: SELECT $md.method\_id$ AS $m_{ch_i}\_id$

FROM $class\_def$ AS c, $method\_def$ AS md

WHERE $c.pathname = c_{ch_i}.pathname$ AND

$c.class\_name = c_{ch_i}.name$ AND

$md.method\_name = m_{ch_i}.name$ AND

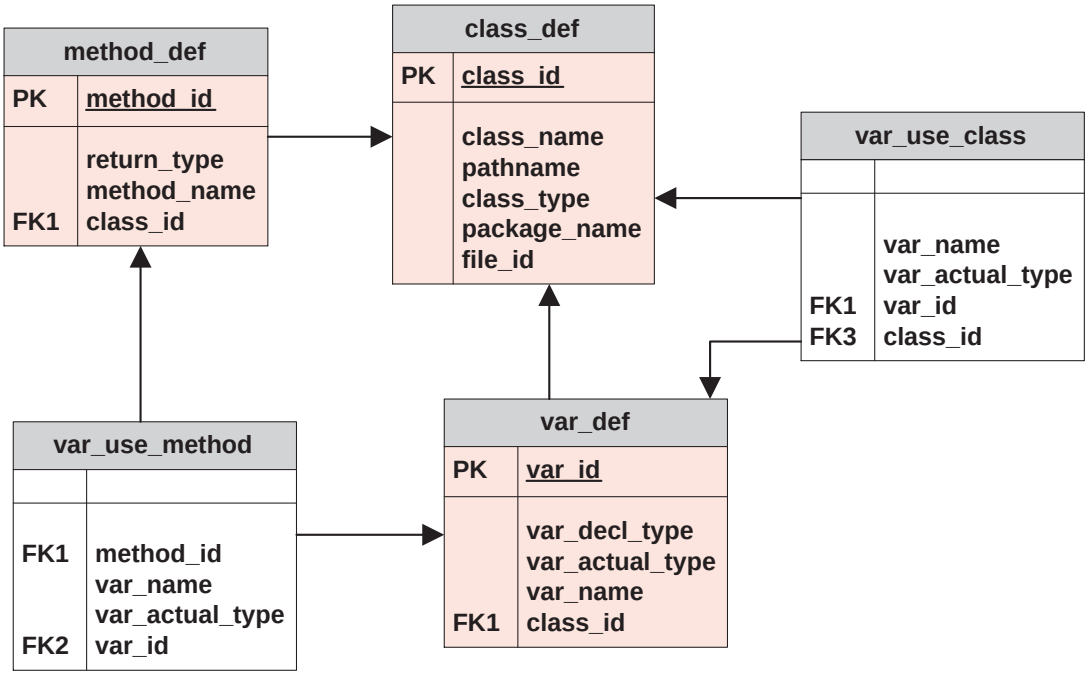

Figure 7.5: Variable Usage Level Tables

$md.class\_id = c.class\_id$;

$qry5$: SELECT $C.pathname$ AS $impacted\_class\_name$

FROM $class\_def$ AS C, $method\_call\_class$ AS mcc WHERE $mcc.method\_id =$

$m_{ch_i}\_id$ AND

$mcc.class\_id = C.class\_id$;

$qry6$: SELECT $C.pathname$ AS $impacted\_class\_name$

FROM $class\_def$ AS C, $method\_call\_method$ AS mcm, $method\_def$ AS vd, $method\_def$ AS md

WHERE $mum.method\_id = md.method\_id$ AND $md.class\_id = C.class\_id$ AND $mum.method\_id = m_{ch_i}\_id$;

$qry7$: SELECT $C.pathname$ AS $impacted\_class\_name$

FROM *class_def* AS *C*, *constructor_call_class* AS *ccc*, *method_def* AS *md*

WHERE *ccc.method_id* = *md.method_id* AND

md.class_id = *C.class_id* AND

ccc.method_id = m_{ch_i}*_id*;

*qry*8 : SELECT *C.pathname* AS *impacted_class_name*

FROM *class_def* AS *C*, *constructor_call_method* AS *ccm*, *method_def* AS *md*

WHERE *ccm.method_id* = *md.method_id* AND *md.class_id* = *C.class_id* AND

ccm.method_id = m_{ch_i}*_id*;

Algorithm 8 first checks if a change is a causal change, then it checks if the method to be changed is in the set of impactful elements of class c. If these conditions are met, then the algorithm selects the classes that call the method by making queries to tables that store impactful methods. *qry*4 identifies the ID of the method in the table. If a method or a constructor is called in the scope of a class, then this information is stored in the *method_call_class* table or the *constructor_call_class* table; if a method or a constructor is called in the scope of a method, this information is stored in the *method_call_method* or in the *constructor_call_method* table. *qry*5, *qry*6, *qry*7, and *qry*8 make separate queries to these four tables and select classes that call the method to be changed.

7.3 Case Study

This section illustrates the CIA method described in this research through the *JRGrep* application. JRGrep is a graphical, Java-based, implementation of the well-known grep utility [Fie06].

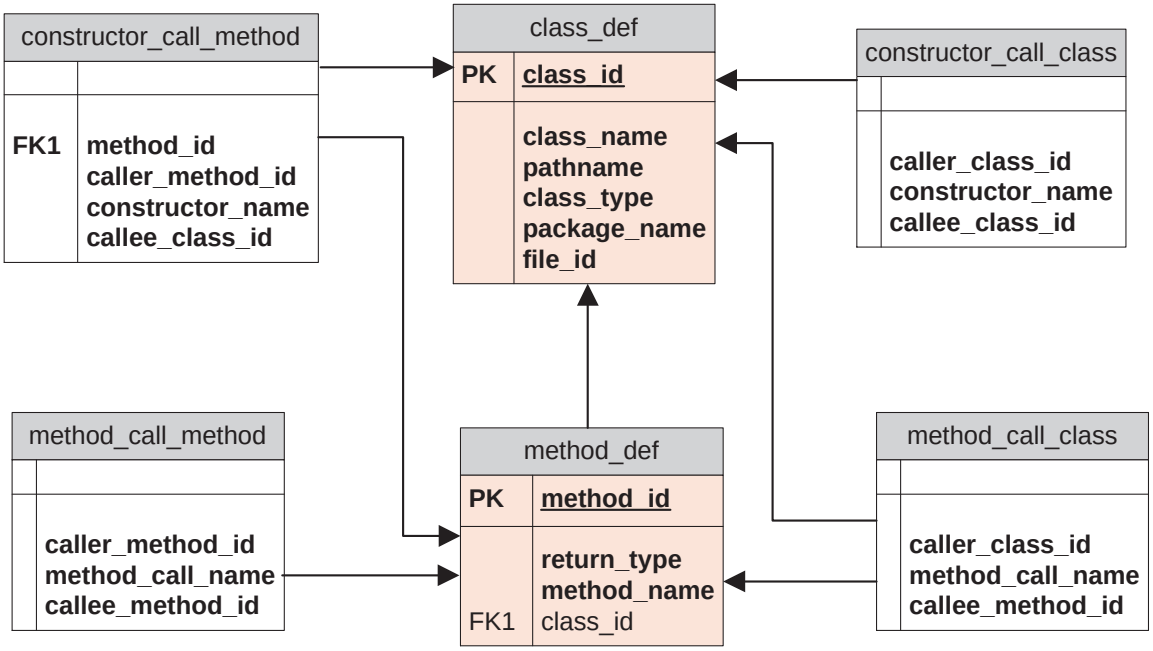

Figure 7.6: Method Call Level Tables

Figure 7.7 depicts the class diagram of JRGrep. Table 7.2 summarizes the couplings found among classes in Figure 7.7. Table 7.3 shows an analysis of class relations and the values for the change impact and the sensitivity to change metrics. Rankings based on the sensitivity and the change impact values of classes are presented in Tables 7.4 and 7.6. According to Table 7.4, *MainWindow* is the most sensitive. *FileSearchListener*, *ResultsListModel*, *FileFoundEvent*, and *Bundle* are not sensitive to any change. The class diagram shows that *MainWindow* uses or depends on five other classes while the not-sensitive classes do not use or depend on any other class except for *FileSearchListener*. There is an association between *FileSearchListener* and *FileFoundEvent* but an instance of *FileFoundEvent* is referenced without using any of its variables or calling its methods. According to Table 7.6, *Bundle* has the greatest impact. *tty* and *Main* do not have any change impacts. The class diagram

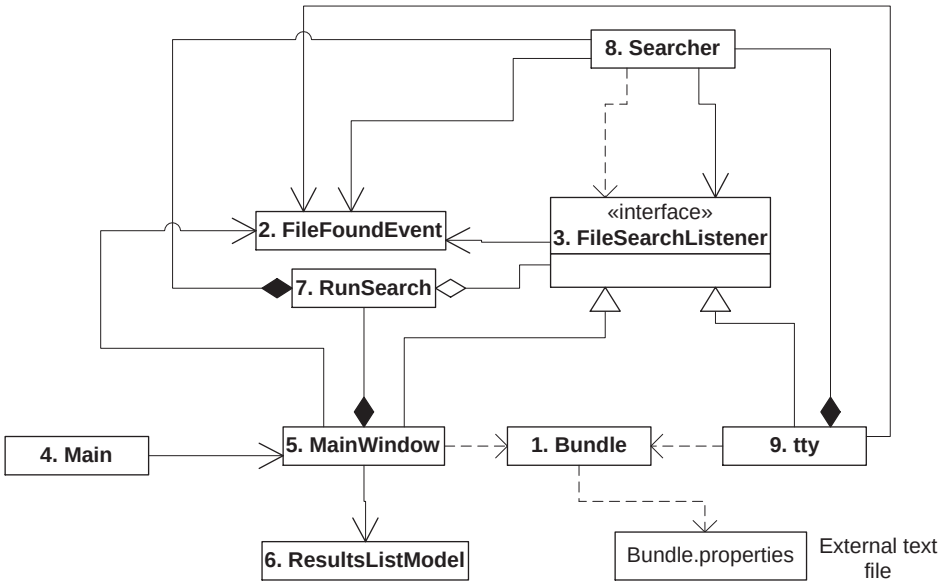

Figure 7.7: Class Diagram for JRGrep Application

shows that *Bundle* is used by two classes while the non-impactive classes *tty* and *Main* are not used or depended on by any other classes. The tables show that there is no correlation between the change impact metric and the size of the client set, and there is no correlation between the sensitivity to change metric and the size of the server set.

7.3.1 Visualization of Metrics

Applying visualization to software metrics is a valuable technique for rapid analysis and decision making. Examples of metrics that could be visualized include sensitivity, change impact, number of impactful elements, number of clients, number of servers, number of methods, number of public variables, lines of code. A popular method of visualizing software metrics on a given code module is the Kiviat diagram [SG01].

Table 7.1: **Descriptive Statistics of JRGrep Application**

CN	Classes	LOC	NOM	NOV	NIE
1	Bundle	18	3	2	2
2	FileFoundEvent	10	2	2	4
3	FileSearchListener	6	3	0	5
4	Main	7	1	0	0
5	MainWindow	285	10	17	0
6	ResultsListModel	22	5	2	4
7	RunSearch	29	4	7	7
8	Searcher	115	14	11	9
9	tty	62	6	3	0

Table 7.2: **JRGrep Application Coupling Matrix**

CN	1	2	3	4	5	6	7	8	9
1									
2									
3		2.0.0.0.0							
4					2.0.1.1.0				
5	4.0.19.0.19	2.0.1.1.0	8.0.3.0.2			2.0.3.1.1	64.0.3.1.6		
6									
7			32.0.0.0.0					64.0.4.1.6	
8		2.0.1.1.2	6.5.3.0.2						
9	4.0.7.0.7	2.0.1.1.0	8.0.3.0.2					64.0.4.1.7	

This diagram allows multiple components of varied ranges to be shown on the same chart. Each software measurement (metric) is indicated by radial axes. The upper and lower limit for each metric are indicated by the outer- and inner-concentric circles, respectively. The measured value for each category then is plotted on each axis, and the connecting lines form a visual pattern of code complexity.

Figure 7.8 depicts the Kiviat diagram for class FileSearchListener. In this figure, NOV is for number of variables defined in FileSearchListener, NOM is for number

Table 7.3: **JRGrep Application Coupling Analysis**

CN	Servers	Clients	Outgoing Coupling Sum	Incoming Coupling Sum	Sensitivity to Changes	Change Impact
1		5, 9		4.0.26.0.26	0.00	0.71
2		3, 5, 8, 9		2.0.3.3.2	0.00	0.50
3	2	5, 7, 8, 9	2.0.0.0.0	46.5.9.0.6	0.00	0.21
4	5		2.0.1.1.0		0.17	0.00
5	1, 2, 3, 6, 7	4	80.0.29.3.28	2.0.1.1.0	0.87	0.17
6		5		2.0.3.1.1	0.00	0.18
7	3, 8	5	96.0.4.1.6	64.0.3.1.6	0.21	0.21
8	2, 3	7, 9	6.5.4.1.4	64.0.8.2.13	0.54	0.44
9	1, 2, 3, 8		78.0.15.2.16		0.51	0.00

Table 7.4: **JRGrep Application Class Ranking Based On Change Sensitivity**

Rank	Class	Sens. to change	Server set size
1	MainWindow	0.89	5
2	tty	0.61	4
3	Searcher	0.59	2
4	RunSearch	0.34	2
5	Main	0.25	1
6	FileSearchListener	0.20	1
7	ResultsListModel	0.00	0
8	FileFoundEvent	0.00	0
9	Bundle		0

of methods defined in FileSearchListener, LOC is for lines of code of FileSearchListener, *ChIm* is for the change impact metric of FileSearchListener, and *Sen* is for the sensitivity to change metric. The Kiviat diagram also shows the number of servers (NOS), the number of client (NOC), and the number of impactful elements (NIE) of class FileSearchListener.

Table 7.5: **JRGrep Application Class Ranking Based On Change Impact**

Rank	Class	Change impact	Client set size
1	FileSearchListener	0.66	4
2	Bundle	0.66	2
3	FileFoundEvent	0.58	4
4	Searcher	0.44	2
5	RunSearch	0.21	1
6	ResultsListModel	0.18	1
7	MainWindow	0.17	1
8	tty	0.00	0
9	Main	0.00	0

Table 7.6: **Maximum and Minimum Values of JRGrep Application Metrics**

No	Metric	Max	Min	FileSearchListener
1	LOC	285	6	6
2	NOV	17	0	0
3	NOM	14	1	3
4	NOC	4	0	4
5	NOS	5	0	1
6	NIE	9	0	5
7	ChIm	0.66	0	0.66
8	Sen	0.89	0	0.20

7.4 Summary

This chapter analyzed the characteristics of changes in OO software and presented techniques to analyze their impacts. The techniques are based on automated analysis of couplings in the program source. Several algorithms for computing the impact of different change categories are presented and metrics for evaluating classes are defined. The technique is illustrated through a case study.

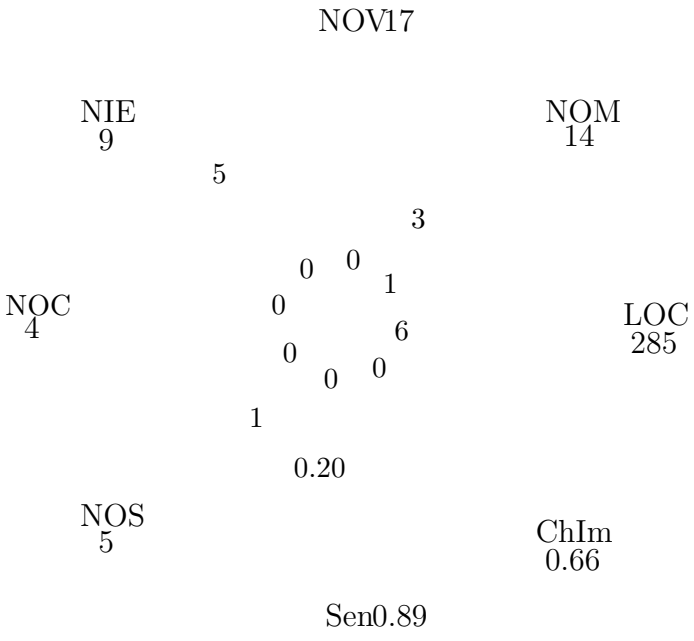

Figure 7.8: Kiviat Diagram for FileSearchListener Metrics

Chapter 8: COUPLING-BASED DESIGN PATTERN DETECTION

A *design pattern* is a general repeatable solution to commonly occurring problems in software design [GHJV01]. It is a description or template for how to solve a problem that can be used in many different situations. A design pattern is not a finished design that can be transformed directly into code. Object-oriented design patterns typically show relationships and interactions among classes or objects, without specifying the final application classes or objects that are involved. Algorithms are not regarded as design patterns, since they solve computational problems rather than design problems.

Design patterns can speed up the development process by allowing designers to use structure that have been successful in previous projects. Effective software design requires considering issues that may not become visible until later in the implementation, after deployment, or when portions of the system are reused in other systems. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for programmers and design architects who are familiar with the patterns.

Often, software developers only understand how to apply certain software design techniques to certain problems. However, these techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that does not require specifics to be tied to a particular problem. In addition, patterns allow developers to communicate using well-known names for software

interactions. Common design patterns can be improved over time, making them more robust than single-use designs.

Because most current software projects deal with evolving products consisting of a large number of components, their architectures can become complicated and cluttered. Design patterns can impose structure on the system through common abstractions. Consequently, identifying implemented design patterns could be useful for comprehending existing designs and provide information needed for refactoring [Vok06]. Thus, design pattern identification from source code can help improve software maintainability and reuse of designs.

This research presents a design pattern structure detection methodology, in which both the system under study as well as the design pattern to be detected are described in terms of graphs. The approach employs a coupling matrix representing all important aspects of their static structure. Relations in patterns are realized as couplings in software, and we find these couplings through analysis. To detect patterns, we employ a graph similarity algorithm [BGH$^+$04], which takes the system and the pattern graph as input, and calculates similarity scores between their vertices. The major advantage of this approach is the ability to detect patterns in their basic form (the one usually found in the literature), as well as their modified versions. This is a significant prerequisite since most design patterns can be implemented with innumerable variations [FBB$^+$99], [SS].

8.1 Representation of System and Patterns

Prior to the pattern detection process, it is necessary to define a representation of the structure of both the system under study and the design patterns to be detected. Such

a representation should incorporate all information that is vital to the identification of patterns. This research models the relationships between classes (as well as other static information) in an object-oriented design using matrices. The key idea is that the class diagram is essentially a directed graph that can be perfectly mapped into a square matrix. The main advantage of this approach is that matrices can be easily manipulated.

The relationships or attributes of the system entities to be represented depend on the specific characteristics of the patterns that the designer wishes to detect. Characteristics of a pattern include generalizations, implementations, associations, dependencies, compositions, and aggregations with some specific conditions. For example, generalization in the context of design patterns implies that the parent class should be an abstract class. Furthermore, the overriding of a method may require specifically that it should invoke the overridden method. However, the similarity algorithm does not depend on the specific types of matrices that are used. The designer can freely set any kind of information as input, provided that one can describe the system and the pattern as matrices using this information.

In Chapter 3, section 3.2.3, we defined couplings in terms of object-oriented relationships and assigned weights for each coupling base type. We use the sum of weights of coupling base types in matrices that model the system and patterns.

8.1.1 Analysis and Representation of Design Patterns

Design patterns have two aspects: structure and semantics. Structural aspects of design patterns tell us what kind of relationship should be formed among the participants of a design pattern. Semantics tell us what functionalities or actions should be performed in this relationship.

Design patterns have been decomposed into recurring elements in order to improve their detection process and results [SS02]. SPQR (System for Pattern Query and Recognition) [SS] is one of a few tools that exploit this approach, and it detects design patterns automatically in C++. *Elemental Design Patterns* (EDPs), the subcomponents of design patterns, play a central role in the context of SPQR. EDPs are divided into three groups [AMR]:

1. *Object Element*, which contains three elemental patterns dealing with the creation and definition of objects (Create Object, Abstract Interface, and Retrieve)

2. *Type Relation*, which contains one elemental pattern describing the inheritance relationship

3. *Method Invocation*, which contains twelve elemental patterns describing the common method calls identified in the GoF catalog [GHJV01]

Coupling measures try to capture the structural aspects of software systems. Thus, coupling measures can be used to identify the structural aspects of design patterns that have interacting participants. Identifying the structural aspects as a first step greatly reduces the search space for final confirmation of design patterns.

The following subsections analyze Adapter, Composite, and Observer patterns, and identify couplings among their participants from a sample implementation. Dozens of patterns have been defined in the literature [GHJV01]. These three patterns are chosen for several reasons: frequency of usage, number of participants, maintenance related traits, and feasibility for this study. Other patterns would be analyzed and detected in similar ways. These three patterns are used in OO design with medium to high frequency and they each have at least three participants [dof07], which makes

coupling analysis possible. Among these patterns, the observer pattern is considered to have the least change impact effect. They are also among the identified patterns in the evaluation of a recent published design pattern detection methodology [TCSH06], which makes it possible for us to compare our results.

8.1.2 Adapter Pattern

The intent of the Adapter pattern is to convert the interface of a class into another interface that clients expect [GHJV01]. The Adapter pattern is implemented by creating a new class with the desired interface and then wrapping the original class methods to effectively contain the adapted object. Adapter lets classes work together that could not otherwise because of incompatible interfaces, and it binds the client to an interface, not an implementation.

The Adapter pattern has four participants: *Target*, *Adapter, Adaptee*, and *Client*. *Target* defines the domain-specific interface that Client uses. *Adapter* adapts the interface *Adaptee* to the Target interface. *Adaptee* defines an existing interface that needs adapting. *Client* collaborates with objects that conform to the Target interface.

There are two kinds of adapters: *Object* Adapters and *Class* Adapters [FFBS04]. Object Adapters use composition in adapting the adaptee, while Class Adapters use inheritance. A Class Adapter uses multiple inheritance, which Java does not support. Therefore, this research does not consider Class Adapter pattern. Figures 8.1 and 8.2 depict the structure of Object and Class Adapter patterns. Table 8.1 shows the couplings that are expected in a proper implementation of the Object Adapter pattern.

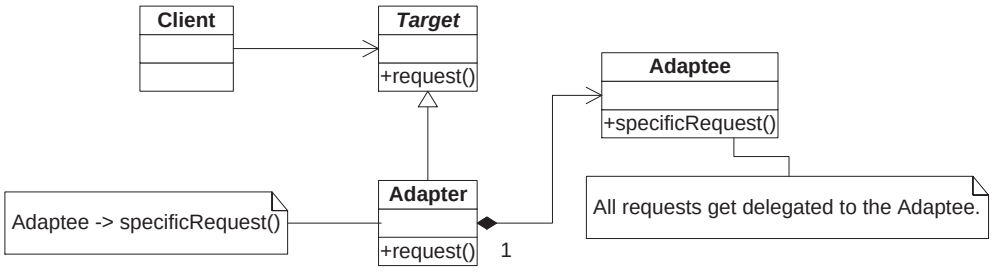

Figure 8.1: UML Class Diagram of the Object Adapter Pattern

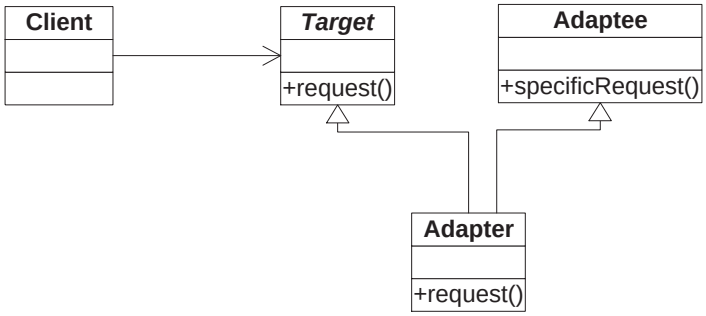

Figure 8.2: UML Class Diagram of the Class Adapter Pattern

Table 8.1: **Couplings in Object Adapter Pattern Structure**

Classes	Client	Target(I)	Adapter	Adaptee	$R_{c_{out}}$
Client		4.0.1.0.0	4.0.1.1.0		$\{4\}$
Target					$\{\}$
Adapter		64.0.1.0.0		32.0.2.1.0	$\{64, 32\}$
Adaptee					$\{\}$
$R_{c_{in}}$	$\{\}$	$\{4, 64\}$	$\{4\}$	$\{32\}$	$\{4, 32, 64\}$

8.1.3 Composite Design Pattern

The Composite pattern composes objects into tree structures to represent part-whole hierarchies. The Composite pattern lets clients treat individual objects and compositions of objects uniformly. The Composite pattern is frequently used.

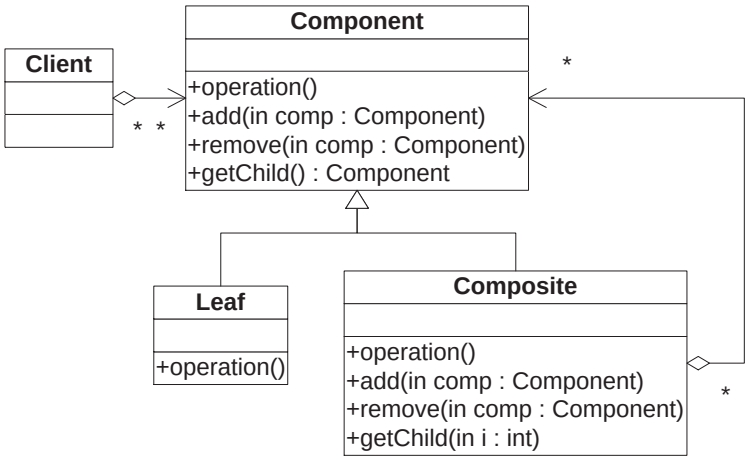

Figure 8.3: UML Class Diagram of Composite Pattern

There are four participants in the Composite pattern: *Component, Leaf, Composite,* and *Client*. *Component* declares the interface for objects in the composition, implements default behavior for the interface that is common to all classes, declares an interface for accessing and managing its child components, and optionally, defines an interface for accessing a component's parent in the recursive structure, and implements it if needed. *Leaf* represents leaf objects in the composition. A leaf has no children and defines behavior for primitive objects in the composition. *Composite* defines behavior for components that have children, stores child components, and implements child-related operations in the Component interface. *Client* manipulates objects in the composition through the Component interface.

Figure 8.3 depicts the structure of the Composite design pattern, and Table 8.2 presents the couplings that are expected in a proper implementation of the Composite pattern.

Table 8.2: **Coupling Matrix for Composite Pattern Structure**

Classes	Component(A)	Leaf	Composite	Client	$R_{c_{out}}$
Component	2.0.0.0.0				{2}
Leaf	128.0.1.0.0				{128}
Composite	128.0.3.0.2 16.0.0.0.0				{128, 16}
Client	16.0.1.0.0				{16}
$R_{c_{in}}$	{2, 16, 16, 128, 128}	{}	{}	{}	{2, 16, 16, 128, 128}

8.1.4 Observer Pattern

The Observer pattern is one of the most commonly used design patterns [GHJV01]. It defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically. The intent of this pattern is to minimize the change impacts of the subject object through loose coupling to observers. Figure 8.4 depicts the structural characteristics of the Observer pattern. The Observer pattern has four main participants: *Subject, Observer, ConcreteSubject*, and *ConcreteObject*. *Subject* knows its observers and can have any number of observers. Subject provides an interface for attaching and detaching Observer objects at run time. *Observer* provides an update interface to receive a signal from the Subject. *ConcreteSubject* stores subject states of interest to the observer and sends notifications to its observers. *ConcreteObserver* maintains a reference to a ConcreteSubject object, maintains observer state, and implements the update operation. Table 8.3 gives the coupling matrix for Figure 8.4.

Several variations are possible in implementations of the observer pattern. Participants can be combined so that one participant can play several roles [JUn07]. For

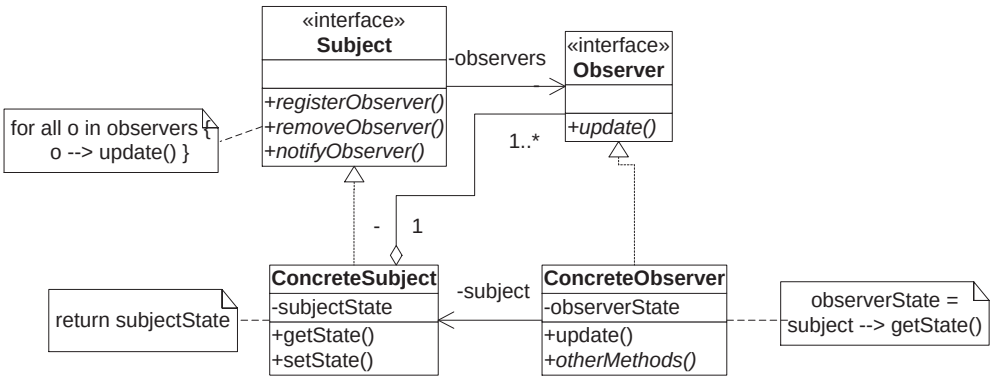

Figure 8.4: UML Class Diagram of the Observer Pattern

Table 8.3: **Coupling Matrix for Standard Observer Pattern Structure**

Classes	Subject(I)	Observer(I)	Concrete-Subject	Concrete-Observer	$R_{c_{out}}$
Subject		2.0.0.0.0			$\{2\}$
Observer					$\{\}$
ConcreteSubject	64.0.3.0.2	16.0.1.0.0			$\{64, 16\}$
ConcreteObserver		64.0.1.0.0	16.0.2.1.1		$\{64, 16\}$
$R_{c_{in}}$	$\{64\}$	$\{2, 16, 64\}$	$\{16\}$	$\{\}$	$\{2, 16, 16, 64, 64\}$

example, in Figure 8.5 the Subject combines the roles of three participants: Subject, ConcreteSubject, and ConcreteObserver. Table 8.4 gives the coupling matrix for Figure 8.5.

Figure 8.6 shows another variation of the Observer pattern. Here Subject combines the roles of Subject and ConcreteSubject. Table 8.5 gives the coupling matrix for Figure 8.6.

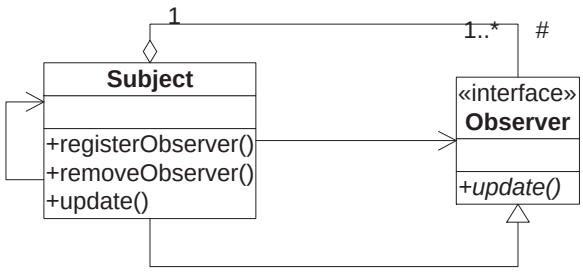

Figure 8.5: UML Class Diagram of a Variation of the Observer Pattern

Table 8.4: **Coupling Matrix for the Observer Pattern Variation 1**

Classes	Subject	Observer(I)	$R_{c_{out}}$
Subject	4.0.1.0.0	2.0.0.0.0 64.0.3.0.2 16.0.1.0.0	$\{4, 2, 64, 16\}$
Observer			$\{\}$
$R_{c_{in}}$	$\{4\}$	$\{2, 64, 16\}$	$\{2, 4, 16, 64\}$

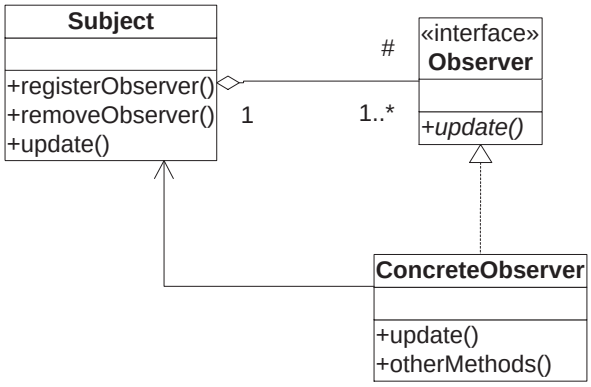

Figure 8.6: UML Class Diagram of another Variation of the Observer Pattern

8.2 Similarity Scoring Algorithm

The similarity scoring algorithm is the foundation of the design pattern detection methodology used in this paper. Blondel et al. [BGH+04] introduced a concept of

Table 8.5: **Coupling Matrix for the Observer Pattern Variation 2**

Classes	Subject	Observer(I)	ConcreteObserver	$R_{c_{out}}$
Subject		16.0.0.0.0		{16}
Observer				{}
ConcreteObserver	4.0.2.0.2	64.0.1.0.0		{4, 64}
$R_{c_{in}}$	{4}	{16, 64}	{}	{4, 16, 64}

similarity between vertices of directed graphs. Let G_A and G_B be two directed graphs with n_A and n_B vertices. They defined an $n_B \times n_A$ similarity matrix S whose real entry s_{ij} expresses how similar vertex j (in G_A) is to vertex i (in G_B); in other words, s_{ij} is their similarity score. In the special case where $G_A = G_B = G$, the matrix S is square and the score s_{ij} is the similarity score between the vertices i and j of G.

Let A be the matrix whose entry (i, j) is equal to the number of edges between the vertices i and j in G_A (the adjacency matrix of G_A), and let B be the matrix whose entry (i, j) is equal to the number of edges between the vertices i and j in G_B (the adjacency matrix of G_B). These matrices are combined to create a similarity matrix, $S(G_A, G_B)$. This is calculated by recomputing a recurrence relation until the value of the recurrence relation converges (stops changing). The recurrence relation is based on the matrices A, B, and their transposes, A^T and B^T. The recurrence relation is:

$$S_{k+1} = BS_kA^T + B^TS_kA$$

S_0 is the identity matrix–all entries are 1. Unfortunately, this function does not always converge. Sometimes it oscillates between two convergent values; one when k is even and one when k is odd. To avoid this problem, Blondel et al. [BGH$^+$04] used the one-norm of the matrix S_{k+1} to equalize the relation. The 1-norm of a matrix is

the maximum sum of the columns[1], usually denoted by $\|M\|_1$. Thus, the complete expression is

$$S_{k+1} = \frac{BS_kA^T + B^TS_kA}{\|BS_kA^T + B^TS_kA\|_1} \tag{8.1}$$

The recurrence computation stops after an even number of iteration (k is even).

The number of floating point operations for this algorithm [BGH+04] is on the order of

$$kn_An_B(\frac{e_A}{n_A} + \frac{e_B}{n_B}), \tag{8.2}$$

where e_A and e_B are the number of edges of graphs G_A and G_B. In the worst case, $e_A = n_A^2$ and $e_B = n_B^2$ (all entries in the corresponding adjacency matrices are equal to 1) and, therefore, the maximum number of floating point operations is on the order of $k(n_A^2n_B + n_An_B^2)$. However, the adjacency matrices required for pattern detection are usually sparse, reducing the computational complexity to $e_X \ll n_X^2$.

For design pattern detection, the similarity algorithm can be used to calculate the similarity between the vertices of the graph describing the pattern (G_A) and the corresponding graph describing the system (G_B). This will lead to a similarity matrix of size $n_B \times n_A$.

8.2.1 From Coupling Tables to Matrices

The similarity algorithm by Blondel et al. computes similarities between vertices of unweighted directed graphs [BGH+04]. One way to apply this algorithm to design pattern detection is to model the system and the patterns as unweighted directed graphs. In this case, there could be several graphs for each pattern based on the

[1]The literature gives many definitions for 1-norm; this one is used by Blondel et al. [BGH+04].

characteristics that distinguish patterns [TCSH06]. One question is whether it is possible to use one weighted digraph to represent the patterns and the system and whether the graph vertices similarity algorithm can still be applied to the weighted digraph. Fortunately, Blondel et al. points out that their algorithm can also be applied to networks (**weighted** directed graphs) [BGH$^+$04]. Further, it is possible to represent the patterns as weighted digraphs deriving weights from couplings. This makes it possible to model the system and patterns using one graph for each and still apply the similarity algorithm. The advantage of using one weighted digraph instead of several unweighted digraphs is that it reduces the complexity of the design pattern detection process. This research models the system and patterns as networks and derives the weights from couplings.

The coupling measures defined in Chapter 3 has five sub-measures. Among these sub-measures, the coupling base type indicates the type of relationship, and the other four measures represent the number of attribute references, method calls, return types, and parameters that appear in this relationship. The structural aspects of a design pattern are reflected in coupling type indicators. Therefore, to detect design pattern structures, we only need the values of the coupling base types.

A preliminary study of the similarity algorithm gives insight into how to apply this algorithm. This algorithm computes similarity scores between a pair of vertices by iteratively and simultaneously computing the similarity scores of their in/out neighbors pairs. Therefore, distinct values with large gaps should help increase the accuracy of similarity scores. In addition, there could be more than one type of coupling among classes. It should be kept in mind that the combination of these couplings should not resemble any other coupling type, i.e., the combination should be distinct and one should be able to tell the origins of this combination. Based on the analysis of

the implementation of design patterns and the similarity algorithm, we developed the following rules for deriving weights from couplings:

Rule 1:

1. Inheritance, interface implementation, and abstract class implementations are assigned separate values in general. However, in some design pattern implementations, *abstract class* and *interface* are used interchangeably. For this reason, interface implementation and abstract class implementation coupling types are combined and assigned the value of the interface coupling type.

2. Association and dependency do not have clear cut differences in implementations. Hence, these two coupling types are combined for this application.

3. To distinguish each kind of relationship, the coupling categories are assigned values as follows:

    - inheritance – 20,000
    - abstract class and interface implementation – 10,000
    - composition – 8,000
    - aggregation – 6,000
    - association, dependency – 100
    - exception – 1
    - external – 0 (This means this coupling is not considered. One reason is that the design patterns do not include external couplings. The other reason is that there is no external coupling in our example application.)

The weights may look arbitrary, however in our studies they fulfill the requirement for being distinctive for each coupling type in being used alone or combined. They explicitly do not indicate a quantitative judgement on the types.

8.2.2 An Example

Let us assume the system under study is the sample Observer pattern implementation in Appendix B. Figure 8.4 from section 8.1.4 depicts both the system and the pattern to be detected, and both have the same coupling table as in Table 8.3. Their coupling matrices are:

$$G_{A(observer)} = G_{B(system)} = \begin{pmatrix} 0 & 100 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 10000 & 100 & 0 & 0 \\ 0 & 10000 & 6000 & 0 \end{pmatrix}$$

The similarity algorithm produces the following similarity scores matrix for $G_{A(observer)}$ and $G_{B(system)}$:

$$Sim(G_A, G_B) = \begin{pmatrix} & (Subject) & (Observer) & (Concrete-Subject) & (Concrete-Observer) \\ (Subject) & \mathbf{0.2321} & 0.0198 & 0.0106 & 0.0043 \\ (Observer) & 0.0198 & \mathbf{0.6128} & 0.3675 & 0.0000 \\ (Concrete-Subject) & 0.0106 & 0.3675 & \mathbf{0.4453} & 0.0214 \\ (Concrete-Observer) & 0.0043 & 0.0000 & 0.0214 & \mathbf{0.6325} \end{pmatrix}$$

$Sim(G_A, G_B)$ is called a *self-similarity matrix* of the observer pattern. As expected, the pattern participants have a high similarity score with themselves; that is,

the diagonal entries are large. Blondel et al./ has shown that the largest entry of a self-similarity matrix always appears on the diagonal and that, except for trivial cases, the diagonal elements of a self-similarity matrix are non-zero [BGH$^+$04]. In this example, diagonal elements dominate all elements on the same row and column. The similarity score between classes *Observer* and *ConcreteSubject*, $s(Observer, ConcreteSubject)$, is 0.3675, and higher than the similarity scores for other class pairs. This indicates that next to themselves, *Observer* and *ConcreteSubject* are more similar to each other in structure than other classes. A check of the sample Observer pattern structure shows that Observer class has three incoming relationships, ConcreteSubject has two incoming relationships, Subject has one incoming relationships, and ConcreteObject has no incoming relationships. The score affirms the fact that Observer and ConcreteSubject are similar in having the higher numbers of incoming relationships.

8.3 Methodology for Detecting Design Pattern Structures

The overall methodology for detecting design patterns in software can be summarized as follows:

1. *Reverse engineering of the system under study.* All structural characteristics of the system under study (i.e., association, generalization, etc.) is represented as one $n \times n$ adjacency coupling matrix, where n is the number of classes.

2. *Construction of subsystem matrices.* A subsystem is defined as a portion of the entire system consisting of classes belonging to packages related through a pattern. We assume that there are at most two packages involved in any pattern, and we justify this assumption based on the principle of cohesiveness

of packages.

3. *Reducing search scope.* All incoming and outgoing coupling type sets are prepared for each pattern role in a pattern to be detected. Each class then is checked for its possibility of playing a role in the pattern by examining its incoming and outgoing coupling type sets. If there is no possibility, then this class and its corresponding row and columns are eliminated from the system coupling matrix.

4. *Application of the similarity algorithm between the subsystem matrices and the pattern matrices.* Similarity scores between each pattern role and each subsystem class are calculated. This corresponds to seeking patterns in each subsystem separately.

5. *Extraction of patterns in each subsystem.* Usually, one instance of each pattern is present in each subsystem, which means that each pattern role is associated with one class.

Similarity scores are used as a starting point. Every class that is identified as a possible pattern role is further examined by comparing its incoming and outgoing coupling type sets with incoming and outgoing coupling type sets of that pattern role.

Before we apply the similarity algorithm, we transform the coupling measures table of the system, CM_s, into *Reduced Design Pattern Detection Matrices* (RDPDM) for a given pattern. Algorithm 9 shows the steps for reducing the search space. Algorithm 10 shows the steps for extracting the structure of a pattern instance.

In Algorithm 9, R_s represents the set of relationships that appear in the target subsystem, R_p represents the set of relationships that appear in a pattern, and R_{c_i}

Algorithm 9 Find RDPDM (CM_s, CM_p)

Require: CM_s is an $(n+1) \times (n+2)$ matrix, where n is the number of classes in the system

Require: CM_p is an $(m+1) \times (m+2)$ matrix, where m is the number of roles in the pattern

```
1: i = 0; c_i ∈ C, C is the set of classes
2: if R_s ⊇ R_p then
3:   while (i < n) do
4:     if (R_ci ∩ R_p = ∅) then
5:       remove the row and column corresponding to c_i from CM_s
6:     else
7:       j = 0;
8:       c_i IsCandidatePatternRole = false;
9:       while (NOT (c_i IsCandidatePatternRole) ∧ j < m) do
10:        if (R_ciin ⊇ R_pjin ∧ R_ciout ⊇ R_pjout) then
11:          c_i IsCandidatePatternRole = true;
12:        end if
13:        j = j+1;
14:      end while
15:      if NOT (c_i IsCandidatePatternRole) then
16:        remove the row and column corresponding to c_i from CM_s
17:      end if
18:    end if
19:    i = i + 1;
20:  end while
21: end if
```

represents the set of incoming and outgoing relationships of class c_i. CM_s is an adjacency matrix of the system with one extra column for the total outgoing relationships set, two additional rows for total incoming relationships set and total relationships set for each class. The main purpose of Algorithm 9 is to remove classes that do not have structural characteristics of any pattern role from consideration. It carries

out two levels of checking. First it compares R_s and R_p. If R_s includes all relationships that a patterns must have, R_p, then the algorithm carries out the next level of checking. If $R_s \not\supseteq R_p$, then there is no need to check the subsystem for a pattern. If $R_s \supseteq R_p$, then the algorithm examines each class by comparing its total incoming and outgoing relationships set, R_{c_i}, with R_p. If class R_{c_i} and R_p have no common relationship element, $R_{c_i} \cap R_p = \emptyset$, then class c_i is removed from consideration. If $R_{c_i} \cap R_p \neq \emptyset$, then c_i is further examined by comparing its incoming and outgoing relationships with the incoming and outgoing relationships of each pattern role. c_i is considered as a candidate pattern role if both its incoming and outgoing relationships sets include the incoming and outgoing relationships of a pattern role.

Algorithm 10 first computes the similarity score matrix for the subsystem and pattern. Next, it chooses the class and pattern role pair corresponding to the largest similarity score and further examines the class to confirm its compatibility with the pattern role. If a class matches to a pattern, then the algorithm removes the row and column corresponding to the class and the pattern, and continues with the next largest value in the remaining similarity matrix. The algorithm terminates when all similarity scores greater than a value, *epsilon*, are examined. If the number of identified pattern roles equals the number of roles in the pattern, then the algorithm declares that the pattern structure is detected. Otherwise, it declares that there is no possibility for the pattern.

Algorithm 10 Design Pattern Structure Detection (G_A, G_B)

Require: Matrix G_A represents a design pattern with m pattern roles (pr) (pr_j, $j = 1$ *to* m)

Require: Matrix G_B represents a software system with n classes (c) (c_i, $i = 1$ *to* n)

1: Compute similarity scores matrix S for graphs G_A and G_B
2: rolesFound = 0; hopefulToFind = true; patternPossible = false; $\epsilon = 0.01$
3: **while** $(rolesFound < n \wedge hopefulToFind)$ **do**
4: Find the largest similarity score, $s_{i,j}$, in the matrix
5: **if** $(s_{i,j} < \epsilon)$ **then**
6: hopefulToFind = false;
7: **else**
8: **if** $(R_{c_{i\,in}} \supseteq R_{pr_{j\,in}}) \wedge (R_{c_{i\,out}} \supseteq R_{pr_{j\,out}})$ **then**
9: choose c_i for the corresponding pattern role pr_j, and add to patternRoleList
10: remove row i and column j from S
11: rolesFound = rolesFound + 1
12: $s_{i,j} = 0$
13: **else**
14: $s_{i,j} = 0$;
15: **end if**
16: **end if**
17: **end while**
18: **if** (rolesFound = m) $\wedge$ (roles are compatible with each other) **then**
19: patternPossible = true;
20: **end if**

8.4 Case Study - Couplings and Design Patterns in JUnit

The proposed methodology is evaluated on JUnit 3.7, which is a regression testing framework for implementing unit tests in Java. JUnit is selected for three reasons:

1. it is an open-source project so its source code is publicly available.

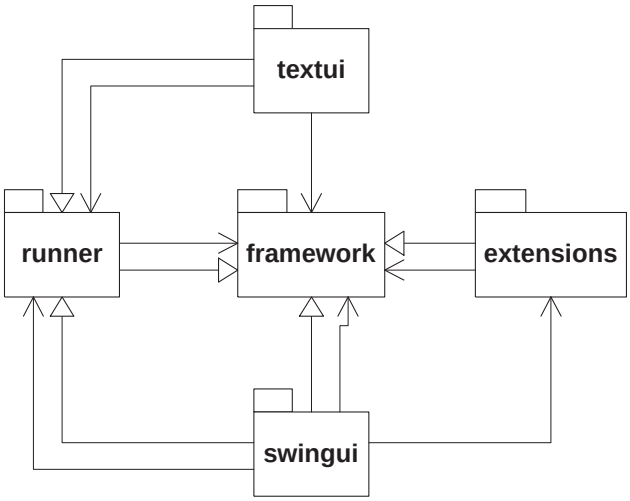

Figure 8.7: JUnit Package Diagram

2. JUnit relies heavily on some well-known design patterns.

3. the authors of JUnit explicitly indicate the implemented design patterns in the documentation and it has been used in the evaluation of another design pattern detection methodology [TCSH06]. Therefore, it was possible to compare the results of the proposed methodology.

JUnit has five packages: *framework* for the basic framework, *runner* for some abstract classes that run tests, *textui* and *swingui* for user interfaces, and *extensions* for some useful additions to the framework. Figure 8.7 shows the packages and their relationships, and Table 8.6 presents base type couplings among these five packages.

These five packages have a total of 40 classes. Dividing a large system into smaller scopes improves efficiency by reducing the convergence time of the similarity algorithm [TCSH06]. Depending on the purpose, a system can be partitioned into groups in many ways. Our goal for dividing the system is (1) to improve efficiency, and (2)

Table 8.6: Package Level Base Type Couplings

Packages	framework	runner	extensions	swingui	textui
framework					
runner	4, 128				
extensions	4, 128				
swingui	4, 128	4, 128	4		
textui	4	4, 128			

to minimize false negative, i.e., the partition should not separate participants of a possible pattern. To detect patterns, we first consider each package separately, then examine every pair of coupled packages. This case study focuses on *framework* package and the combination of *framework* and *swingui* packages.

Table 8.7 gives aliases for each classes in the framework and swingui packages. Table 8.8 presents the couplings in the framework package, Table 8.9 presents the couplings in the swingui package, and Table 8.10 presents the couplings from swingui to framework.

First, we transform the original coupling measures table into a coupling types table by applying Rule 1. Table 8.11 is generated from Table 8.8, Table 8.12 from 3.4, Table 8.13 from 8.2, and Table 8.14 from 8.3.

8.4.1 Analysis of the Adapter Pattern

Applying Algorithm 9 to Table 8.11 for the adapter pattern removes Assert, AssertionFailedError, and Protectable from the table, and generates the following RDPDM

Table 8.7: **Aliases for Class Names in *framework* and *swingui* Packages**

framework	Aliases	swingui	Aliases
Assert	f1	AboutDialog	s1
AssertionFailedError	f2	CounterPanel	s2
Protectable	f3	DefaulFailureDetailView	s3
Test	f4	FailureRunView	s4
TestCase	f5	ProgressBar	s5
TestFailure	f6	StatusLine	s6
TestListener	f7	TestHierarchyRunView	s7
TestResult	f8	TestRunContext	s8
TestSuite	f9	TestRunner	s9
		TestCollector	s10
		TestSuiteLoader	s11
		Version	s12
		TestTreeModel	s13

of the framework package for the adapter pattern:

$$G_{B(framework)} = \begin{pmatrix} 0 & 0 & 0 & 100 & 0 & 0 \\ 10000 & 0 & 0 & 0 & 100 & 0 \\ 6100 & 0 & 0 & 0 & 0 & 0 \\ 100 & 0 & 0 & 0 & 0 & 0 \\ 100 & 0 & 8000 & 6000 & 0 & 0 \\ 16100 & 100 & 0 & 0 & 100 & 100 \end{pmatrix}$$

The following matrix represents the structure of the adapter

$$G_{A(adapter)} = \begin{pmatrix} 0 & 100 & 100 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 10000 & 0 & 8000 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

The similarity algorithm in Section 8.2 produces the following similarity matrix

Table 8.8: **Coupling Matrix for the *framework* package in JUnit**

Classes	Assert	Assertion-FailedError	Protect-able(I)	Test(I)	TestCase	Test-Failure	Test-Listener(I)	TestResult	TestSuite	$R_{c_{out}}$
Assert		8.0.1.0.1								{8}
Assertion-FailedError										{}
Protectable(I)										{}
Test								2.0.0.0.0		{2}
TestCase	128.0.0.0.0			64.0.2.1.1				4.0.2.1.1 2.0.0.0.0		{128, 64, 4, 2}
Test-Failure				16.0.0.0.0 2.0.0.0.0						{16, 2}
Test-Listener		2.0.0.0.0		2.0.0.0.0						{2, 2}
TestResult		2.0.0.0.0 8.0.0.0.0	4.0.2.1.0	4.0.1.0.0 2.0.0.0.0		32.0.1.1.2	16.0.4.0.6			{2, 4, 4, 32, 16, 8, 2, }
TestSuite				64.0.2.1.1 2.1.0.0.0 16.0.1.0.0 4.0.1.0.1	4.0.1.1.1			4.0.1.1.0 2.0.0.0.0	4.0.1.1.1	{64, 4, 4, 4, 2, 2, 16, 4}
$R_{c_{in}}$ $R_{c_{in}}$ $R_{c_{in}}$	{128}	{8, 2, 2, 8}	{4}	{64, 16, 2, 2, 4, 2, 64 2, 16, 4}	{4}	{32}	{16}	{2, 4} {4} {4, 2}	{4}	

Table 8.9: **Coupling Matrix for the *swingui* package in JUnit**

Classes	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	s12	s13
AboutDialog (s1)									2.0.1.1.2				
CounterPanel (s2)													
DefaulFailure-DetailView (s3)													
FailureRun-View (s4)								16.0.1.1.	2.0.1.1.2	64.0.6.1.6			
ProgressBar (s5)													
Status-Line (s6)													
TestHierarchy-RunView (s7)								16.0.0.0.0	2.0.1.1.2	64.0.6.1.6		32.0.1.1.0	4.0.3.3.5
TestRun-Context(s8)													
TestRunner (s9)	4	4.0.1.1.0 32.0.6.1.4	4	4	4 32.0.4.1.2	2.2.0.0.0	4	4 64.0.2.1.1		4	4		
TestRunView (s10)													
TestSelector (s11)													
TestSuite-Panel (s12)													32
TestTreeModel (s13)													

Table 8.10: **Couplings from *swingui* to *framework***

Classes	f1	f2	f3	f4	f5	f6	f7	f8	f9
s1									
s2									
s3						4.0.1.1.0			
s4		2.0.1.1.1		2.0.0.0.0		4.0.3.3.2		2.0.0.0.0	
s5									
s6									
s7				2.0.0.0.0				4.0.2.0.2	
s8				2.0.0.0.0					
s9 s9				2	2	4		32.0.7.4.1 4	
s10				2				2	
s11									
s12		2		2			64		
s13				4					4

Table 8.11: **Coupling *Type* Matrix for the *framework* Package in JUnit**

Classes	**Assert**	**Assertion-FailedError**	**Protect-able(I)**	**Test(I)**	**TestCase**	**Test-Failure**	**Test-Listener(I)**	**TestResult**	**TestSuite**	$R_{c_{out}}$
Assert		1								{1}
Assertion-FailedError										{}
Protectable(I)										{}
Test								100		{100}
TestCase	20,000			10,000				100 100		{20000, 10000, 100}
Test-Failure				6000 100						{6000, 100}
Test-Listener		100		100						{100}
TestResult		100 1	100	100 100		8000	6000			{100, 6000, 1, 8000}
TestSuite				10000 100 6000 100	100			100 100	100	{10000, 100, 6000}
$R_{c_{in}}$	{20000}	{1, 100}	{100}	{10000, 100, 6000}	{100}	{8000}	{6000}	{100}	{100}	{1, 100, 6000 8000, 10000 20000}
$R_c = R_{c_{in}} \cup R_{c_{out}}$	{1, 20000}	{1, 100}	{100}	{100, 6000, 10000}	{100, 10000, 20000}	{100, 6000, } 8000}	{100, 6000}	{100} 1, 8000}	{100, 6000 10000}	{1, 100, 6000, 8000, 10000 20000}

Table 8.12: **Coupling Types in Adapter Pattern Structure**

Classes	Client	Target(I)	Adapter	Adaptee	$R_{c_{out}}$
Client		100	100		{100}
Target					{}
Adapter		10000		8000	{10000, 8000}
Adaptee					{}
$R_{c_{in}}$	{}	{100, 10000}	{100}	{8000}	{100, 8000, 10000}

Table 8.13: **Coupling Types in Composite Pattern Structure**

Classes	Component(A)	Leaf	Composite	Client	$R_{c_{out}}$
Component	100				{100}
Leaf	10000				{10000}
Composite	10000 6000				{10000, 6000}
Client	6000				{6000}
$R_{c_{in}}$	{100, 6000, 10000}	{}	{}	{}	{100, 6000, 10000}

Table 8.14: **Coupling Types in Standard Observer Pattern Structure**

Classes	Subject(I)	Observer(I)	Concrete-Subject	Concrete-Observer	$R_{c_{out}}$
Subject		100			{100}
Observer					{}
ConcreteSubject	10000	6000			{10000, 6000}
ConcreteObserver		10000	6000		{10000, 6000}
$R_{c_{in}}$	{10000}	{100, 6000}	{}	{}	{100, 6000, 10000}

between $G_{A(adapter)}$ and $G_{B(framework)}$ vertices:

$$Sim(G_A, G_B) = \begin{bmatrix} & (Client) & (Target) & (Adapter) & (Adaptee) \\ (Test) & 0.0000 & \mathbf{0.4194} & 0.0000 & \mathbf{0.3355} \\ (TestCase) & 0.0019 & 0.0017 & \mathbf{0.3083} & 0.0014 \\ (TestFailure) & 0.0011 & 0.0012 & \mathbf{0.1881} & 0.0009 \\ (TestListener) & 0.0000 & 0.0009 & 0.0031 & 0.0007 \\ (TestResults) & 0.0005 & 0.0028 & 0.0041 & 0.0022 \\ (TestSuite) & 0.0031 & 0.0017 & \mathbf{0.4964} & 0.0014 \end{bmatrix}$$

Table 8.15: **Coupling Type Matrix for Observer Pattern Variation 1**

Classes	Subject	Observer(I)	$R_{c_{out}}$
Subject	100	100 10000 6000	 $\{100, 6000, 10000\}$
Observer			$\{\}$
$R_{c_{in}}$	$\{100\}$	$\{100, 6000, 10000\}$	$\{100, 6000, 10000\}$

Table 8.16: **Coupling Type Matrix for the Observer Pattern Variation 2**

Classes	Subject	Observer(I)	ConcreteObserver	$R_{c_{out}}$
Subject		6000		$\{6000\}$
Observer				$\{\}$
ConcreteObserver	100	10000		$\{100, 10000\}$
$R_{c_{in}}$	$\{100\}$	$\{6000, 10000\}$	$\{\}$	$\{100, 6000, 10000\}$

Finally, we analyze this matrix using Algorithm 10. The algorithm finds only one role, Test as Target, and terminates. This shows that there is no adapter pattern in the framework package. A manual check of the JUnit documentation confirms these results.

8.4.2 Analysis of the Composite Pattern

Applying Algorithm 9 to Table 8.11 for the composite pattern removes Assert, AssertionFailedError, Protectable, and TestListener from the table, and generates the following RDPDM of the framework package for the composite pattern:

$$G_{B(framework)} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 10000 & 0 & 0 & 100 & 0 \\ 6100 & 0 & 0 & 0 & 0 \\ 100 & 0 & 8000 & 0 & 0 \\ 16100 & 100 & 0 & 100 & 100 \end{pmatrix}$$

The following matrix represents the structure of the composite pattern:

$$G_{A(composite)} = \begin{pmatrix} 100 & 0 & 0 & 0 \\ 10000 & 0 & 0 & 0 \\ 16000 & 0 & 0 & 0 \\ 6000 & 0 & 0 & 0 \end{pmatrix}$$

The following is the similarity matrix between $Graph_{A(composite)}$ and $Graph_{B(framework)}$:

$$Sim(G_A, G_B) = \begin{bmatrix} & (Component) & (Leaf) & (Composite) & (Client) \\ (Test) & \mathbf{0.2951} & 0.0000 & 0.0000 & 0.0000 \\ (TestCase) & 0.0031 & \mathbf{0.1933} & 0.3093 & 0.1160 \\ (TestFailure) & 0.0021 & 0.1179 & 0.1887 & \mathbf{0.0708} \\ (TestResults) & 0.0020 & 0.0024 & 0.0039 & 0.0015 \\ (TestSuite) & 0.0043 & 0.3113 & \mathbf{0.4981} & 0.1868 \end{bmatrix}$$

Algorithm 10 analyzes this matrix and identifies that the classes TestSuite, Test, TestCase, and TestFailure form the composite pattern. Here TestSuite plays the role of Composite, Test plays Component, TestCase plays Leaf, and TestFailure plays Client. A manual check of the JUnit documentation confirms these results.

8.4.3 Analysis of the Observer Pattern

Applying Algorithm 9 to Table 8.11 for the standard observer pattern removes Assert, AssertionFailedError, Protectable, and TestResult from the table, and generates the following RDPDM of the framework package for the standard observer pattern:

$$G_{B(framework)} = \begin{pmatrix} 0 & 0 & 0 & 100 & 0 \\ 10000 & 0 & 0 & 0 & 0 \\ 6100 & 0 & 0 & 0 & 0 \\ 100 & 0 & 0 & 0 & 0 \\ 16100 & 100 & 0 & 0 & 100 \end{pmatrix}$$

The following matrix represents the standard structure of the observer pattern:

$$G_{A(observer)} = \begin{pmatrix} 0 & 100 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 10000 & 100 & 0 & 0 \\ 0 & 10000 & 6000 & 0 \end{pmatrix}$$

The following is the similarity matrix between $G_{A(observer)}$ and $G_{B(framework)}$:

$$Sim(G_A, G_B) = \begin{bmatrix} & (Subject) & (Observer) & (Concrete-Subject) & (Concrete-Observer) \\ (Test) & 0.0180 & \mathbf{0.4451} & 0.2670 & 0.0000 \\ (TestCase) & 0.0023 & 0.0018 & 0.0118 & 0.3096 \\ (TestFailure) & 0.0014 & 0.0000 & 0.0065 & 0.1888 \\ (TestListener) & 0.0000 & 0.0000 & 0.0001 & 0.0031 \\ (TestSuite) & 0.0037 & 0.0018 & 0.0184 & \mathbf{0.4985} \end{bmatrix}$$

Algorithm 10 analyzes this matrix. The algorithm finds two roles, Test as Observer and TestSuite as ConcreteObserver, then terminates. This result indicates that there is no standard observer pattern in the framework package. We then consider two variations of the observer pattern. The following matrices represent the variations of the observer pattern:

$$G_{A(observer-v1)} = \begin{pmatrix} 100 & 16100 \\ 0 & 0 \end{pmatrix}$$

$$Graph_{A(observer-v2)} = \begin{pmatrix} 0 & 6000 & 0 \\ 0 & 0 & 0 \\ 100 & 10000 & 0 \end{pmatrix}$$

Applying the similarity algorithm to the reduced system matrix and $G_{A(observer-v1)}$

generates the following similarity matrix:

$$Sim(G_{A1}, G_B) = \begin{bmatrix} & (Subject) & (Observer) \\ (Test) & 0.0024 & \mathbf{0.3845} \\ (TestCase) & 0.3088 & 0.0016 \\ (TestFailure) & 0.1884 & 0.0000 \\ (TestListener) & 0.0031 & 0.0000 \\ (TestSuite) & \mathbf{0.4973} & 0.0016 \end{bmatrix}$$

Analysis result from Algorithm 10 finds an observer pattern variation in the framework package, where TestSuite is a Subject and Test is an Observer. This result is confirmed with the JUnit documentation.

Applying the similarity algorithm on the reduced system matrix and $G_{A(observer-v2)}$ generates the following similarity matrix:

$$Sim(G_{A2}, G_B) = \begin{bmatrix} & (Subject) & (Observer) & (ConcreteObserver) \\ (Test) & 0.0024 & \mathbf{0.3280} & 0.0000 \\ (TestCase) & \mathbf{0.1857} & 0.0013 & 0.3096 \\ (TestFailure) & 0.1133 & 0.0000 & 0.1888 \\ (TestListener) & 0.0019 & 0.0000 & 0.0031 \\ (TestSuite) & 0.2991 & 0.0013 & \mathbf{0.4985} \end{bmatrix}$$

Algorithm 10 could not find the observer pattern variation from this similarity score. A manual analysis reveals that this variation of the observer pattern exists in collaboration of the framework package and swingui package.

We transform the original coupling measures for the framework and swingui packages into a coupling types table, Table 8.17, by applying Rule 1.

Applying Algorithm 9 to Table 8.17 for the observer pattern variation 2 removes f1, f2, f3, s1, s2, s3, s5, s6, s10, s11, s12, and s13 from the table, and generates the

Table 8.17: **Coupling Matrix for Classes in *framework* and *swingui* packages**

CA	f1	f2	f3	f4	f5	f6	f7	f8	f9	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	s12	s13	$R_{c_{out}}$
f1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
f2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
f3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
f4	0	0	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
f5	20000	0	0	10000	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
f6	0	0	0	6100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
f7	0	100	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
f8	0	101	100	100	0	8000	6000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
f9	0	0	0	16100	100	0	0	100	100	0	0	0	0	0	0	0	0	0	0	0	0	0	
s1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	
s2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
s3	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
s4	0	100	0	100	0	100	0	100	0	0	0	0	0	0	0	0	6000	100	10000	0	0	0	
s5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
s6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
s7	0	0	0	100	0	0	0	100	0	0	0	0	0	0	0	0	6000	100	10000	0	8000	100	
s8	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
s9	0	0	0	100	100	100	10000	8100	0	100	8100	100	100	8100	100	100	10100	0	100	100	0	0	
s10	0	0	0	100	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
s11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
s12	0	100	0	100	0	0	10000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8000	
s13	0	0	0	100	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0	
$R_{c_{in}}$																							

following RDPDM:

$$G_{B(frame,swingui)} = \begin{pmatrix} 0 & 0 & 0 & 0 & 100 & 0 & 0 & 0 & 0 & 0 \\ 10000 & 0 & 0 & 0 & 100 & 0 & 0 & 0 & 0 & 0 \\ 6100 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 100 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 100 & 0 & 8000 & 6000 & 0 & 0 & 0 & 0 & 0 & 0 \\ 16100 & 100 & 0 & 0 & 100 & 100 & 0 & 0 & 0 & 0 \\ 100 & 0 & 100 & 0 & 100 & 0 & 0 & 0 & 6000 & 100 \\ 100 & 0 & 0 & 0 & 100 & 0 & 0 & 0 & 6000 & 100 \\ 100 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 100 & 100 & 100 & 10000 & 8100 & 0 & 100 & 100 & 10100 & 0 \end{pmatrix}$$

Applying the similarity algorithm on the reduced system matrix and $G_{A(observer-v2)}$ generates the following similarity matrix:

$$Sim(G_{A2}, G_B) = \begin{bmatrix} & (Subject) & (Observer) & (ConcreteObserver) \\ (f4) & 0.0024 & \mathbf{0.3234} & 0.0001 \\ (f5) & \mathbf{0.1776} & 0.0014 & 0.2961 \\ (f6) & 0.1083 & 0.0022 & 0.1806 \\ (f7) & 0.0018 & 0.0078 & 0.0030 \\ (f8) & 0.0051 & 0.0072 & 0.0087 \\ (f9) & 0.2860 & 0.0013 & \mathbf{0.4767} \\ (s4) & 0.0020 & 0.0001 & 0.0034 \\ (s7) & 0.0042 & 0.0001 & 0.0070 \\ (s8) & 0.0018 & 0.0075 & 0.0030 \\ (s9) & 0.0130 & 0.0006 & 0.0217 \end{bmatrix}$$

Algorithm 10 identifies f9, which is TestSuite in the framework package, as ConcreteObserver and f4, which is Test in the framework package, as Observer. Although f5, which is TestCase in the framework package, has the highest similarity score with respect to Subject, their incoming and outgoing couplings do not match. Hence, the algorithm declares that it cannot find the variation of the observer pattern structure, which is a false negative.

This result questions the suitability of the similarity algorithm in the design pattern detection. We have tried to detect three patterns using coupling measures and the graph similarity scoring algorithm. There have been no false positives but one false negative. The second variation of the observer pattern exists in the collaboration framework and swingui packages, but our approach did not detect it. This indicates that the similarity algorithm is not always reliable. The similarity algorithm is based on the notion that two nodes are similar if they have similar in/out neighbors. In general, graphs that represent patterns are sparse, but graphs that represent full systems vary. In detecting patterns in dense graphs, it is not guaranteed to have high similarity scores for nodes that match pattern nodes. As a result, the largest similarity score may help to find a single system node that matches a pattern node, but it may not lead to find the pattern in its entirety.

8.4.4 Lessons Learned - New Approach

This research has found that coupling measures are useful in reducing the search scope for design patterns. However, our approach with the the graph similarity algorithm [BGH+04], which applies it on weighted graphs in design pattern structure detection, does not always produce correct results, i.e. false negatives occurred. However, since this approach produces results faster and with no false positives, it is still useful.

From the experience in this case study, Algorithm 11 was devised to address the false negatives.

This algorithm identifies all possible subsets of classes that form a specific pattern structure. Applying this algorithm on Tables 8.17 and 8.16 identified s9, which is TestRunner in the swingui package, as ConcreteObserver, f7, which is TestListener in the framework package, as Observer, and f8, which is TestResult in the framework

package, as Subject. A manual check of the JUnit documentation confirms these results.

To summarize, the core questions in this research are:

1. Can coupling measures be used in design pattern detection? If yes, then at what extent? How much help can coupling measures offer?

2. Can the graph similarity scoring algorithm be used in combination with coupling measures? Are there any restrictions?

The results from the case study indicate that:

1. Coupling measures can be used in the design pattern *structure* detection. The semantics of patterns cannot be detected using coupling measures alone.

2. The graph similarity scoring algorithm can be used in combination with coupling measures to detect the structure of a particular design pattern. It is fast and does not give false positives. However, it is not always guaranteed to detect a pattern structure in the system graph.

3. Algorithm 11 only uses coupling measures to detect a pattern structure. This algorithm does not result in false negatives, and shows that coupling measures are useful in detecting design pattern structures.

8.4.5 Discussion of Difference from Other Approaches

This research is different from most other research in that it uses a graph similarity algorithm. There is one other method that uses the same graph similarity algorithm to detect design patterns. This section discusses the difference of our approach from the approach of Tsantalis et al. [TCSH06].

To detect design patterns in source, Tsantalis et al. [TCSH06]

1. Reduced the search space by constructing subsystems according to inheritance hierarchies.
2. Identified nine characteristics for patterns and used unweighted directed graphs to represent each characteristic. As a result, subsystems and patterns can have a number of graph representations depending on how many characteristics they have.
3. Applied a graph similarity algorithm for each pair of subsystems and pattern graphs to detect patterns.

The research in this dissertation

1. First used packages and package pairs to divide system into subsystems, then developed an algorithm to remove classes that cannot be pattern roles from the subsystem. Thus further reduced the search scope. The algorithm used coupling measures to judge classes for their possibility of being a pattern role.
2. Used weighted graphs to represent the reduced subsystem and a pattern. As a result, there is one graph for the subsystem and one for a pattern.
3. Applied the same graph similarity algorithm for a quick search of a design pattern structure.
4. Developed a coupling-based method to identify all possible subsets of classes that form a pattern structure.

Tsantalis et al. evaluated their approach on three projects: JUnit, JHotDraw, and JRefactory. They missed some patterns in JHotDraw and JRefactory. They

explain that those missed patterns lack certain pattern requirements to be considered as patterns, although the documents claim that they are patterns. This research is not evaluated on JHotDraw and JRefactory. Therefore, we currently cannot draw a conclusion about those missed patterns.

This research complements existing design pattern detection techniques by identifying classes that will possibly form a pattern. This result will simplify the rest of the detection process and also help decrease the false negatives in the pattern detection.

8.5 Summary

We have modeled the system and design patterns as weighted directed graphs using coupling information and then applied the graph vertices similarity algorithm. We also developed algorithms to reduce the pattern search scope and to detect a desired pattern structure from the similarity score matrix. The results are promising. Coupling information can help identify the structure of a pattern. The key part is to model the pattern correctly. Since there are variations in implementation, it is crucial to reflect them in the model. As a result, one pattern can have several models depending on possible implementations. Once the patterns and system are correctly modeled using coupling information, applying the graph similarity algorithm yields promising results.

Identifying design patterns also requires more specific examinations. Our current results help reduce the search scope for pattern detection by generating a class diagram of the system with the suspected patterns highlighted. Tools, such as jgraph [jgr] and graphviz [gra], can help.

Algorithm 11 Find All DPS (CM_s, CM_p)

Require: CM_s is an $(n+1) \times (n+2)$ matrix, where n is the number of classes in the system

Require: CM_p is an $(m+1) \times (m+2)$ matrix, where m is the number of roles in the pattern

i = 0; $c_i \in C$, C is the set of classes, PRL_j, $j = 1..m$, are sets for pattern roles
if $R_s \supseteq R_p$ **then**
  while $(i < n)$ **do**
    if $(R_{c_i} \cap R_p = \emptyset)$ **then**
      remove the row and column corresponding to c_i from CM_s
    else
      j = 0;
      c_iIsCandidatePatternRole = $false$;
      while (NOT (c_iIsCandidatePatternRole) $\wedge j < m$) **do**
        if $(R_{c_{iin}} \supseteq R_{p_{j_{in}}} \wedge R_{c_{iout}} \supseteq R_{p_{j_{out}}})$ **then**
          c_iIsCandidatePatternRole = $true$;
          add c_i to PRL_j;
        end if
        j = j+1;
      end while
      if NOT (c_iIsCandidatePatternRole) **then**
        remove the row and column corresponding to c_i from CM_s
      end if
    end if
    i = i + 1;
  end while
end if
$PRL_1 \times PRL_2 \times ... \times PRL_m$ are candidate patterns
while (more candidate exist) **do**
  if (member classes of the next candidate are compatible with each other as the pattern roles) **then**
    report this candidate
  end if
end while

Chapter 9: COUPLING-BASED FAULT MODEL

An object-oriented (OO) software system consists of components that interact with each other to implement the behavior of the system. Principles of object-oriented software development support reuse of software components and easier development and maintenance through better data encapsulation [CCHJ94]. However, dynamic binding, inheritance, polymorphism, and cycles in the dependency among components increase the complexity of the relationships in the object-oriented software. This increased complexity has brought new challenges to integration, testing, and maintenance of the OO software system. Thus modeling and measuring the relationships among components have become necessary and essential activities in finding solutions to the emerged problems.

Coupling analysis models the relationships among software components. Two components are coupled if they are connected. The coupling model in Chapter 3 can be used to develop a model of faults for OO software. This chapter presents a ***theoretical model of faults*** in OO software. The eventual goals are to determine causal correlations among faults in software, relationships among software components, and couplings. This would expand our knowledge of how to avoid and detect faults, improve designs, and create predictive models based on design information. This chapter is preliminary and theoretical; substantial amounts of data would need to be collected to refine and validate this model.

This chapter categorizes OO faults according to coupling types and examines whether coupling types differ with respect to faults and failures. To evaluate coupling

types with respect to faults and failures, we define a measure, **fault index**.

> **Definition**: **Fault index, fi**, measures the degree of correlation between a coupling type and OO faults.

This chapter considers a class to be the interacting component, and a coupling base type is formally defined as a tuple:

$$CBT =< C, LI, RT, D > \tag{9.1}$$

where C is a finite set of classes, LI is a finite set of locus of impact, RT is a finite set of relationship types, and D is a finite set of directions of connection between two classes. Furthermore, the correlation between a coupling base type and faults is defined as a tuple:

$$\langle CBT, F, FI, \delta \rangle \tag{9.2}$$

where CBT is a finite set of coupling base types, F is a finite set of faults, FI is a finite set of coupling fault indexes, and $\delta : CBT \rightarrow FI$ is the coupling fault index function.

Among these elements, C, LI, RT, and D can be statically identified from software artifacts. However, computing fault indexes for different coupling types is challenging. The reason is that computing fault indexes requires to develop a comprehensive fault model with detailed characteristics of each fault.

The organization of this chapter is as follows: Section 9.1 provides a conceptual model for coupling-based fault analysis. Section 9.2 describes the Object-Oriented (OO) coupling types and develops a coupling-based fault classification of OO software. Section 9.3 describes a method for relating fault indexes to coupling types. Finally, Section 9.4 summarizes the ideas.

9.1 Conceptual Model for Correlations Among Relationships, Couplings, and Faults in Object-Oriented Systems

To identify the determining factors in computing a fault index for each coupling type, we first study OO fault types and their relationship with OO coupling types.

Figure 9.1 shows a conceptual model that abstracts the connections among relationships, coupling types, and fault types in a software. It also shows the relationship between faults and failures that are caused by the faults. The severity of failures that are caused by faults is one of the determining factors in computing fault indexes to coupling types. Therefore, before we further explain our conceptual model, we analyze the connection between faults and failures.

A *fault* is defined as an incorrect step, instruction, or data definition in a program [IEE90]. A fault may result in a *failure*, which is observed when the system exhibits incorrect external behavior. An *error* is an internal difference between the computed, observed, or measured values or conditions, and the true, specified, or theoretically correct values or conditions. Finally, a *mistake* is a human misconception that results in a fault [IEE90]. We identify three attributes for a fault: type, severity, and frequency. The type shows in what kind of dependency a particular type of fault may occur. Severity is defined according to the severity of the failure that caused by a fault. Frequency means how often a particular type of fault occurs in OO software. Defining the frequency of OO faults is a nontrivial task and has not yet been addressed.

The relationship between faults and failures is extremely complex. Figure 9.2 illustrates the relationship between faults and failures through an example. The

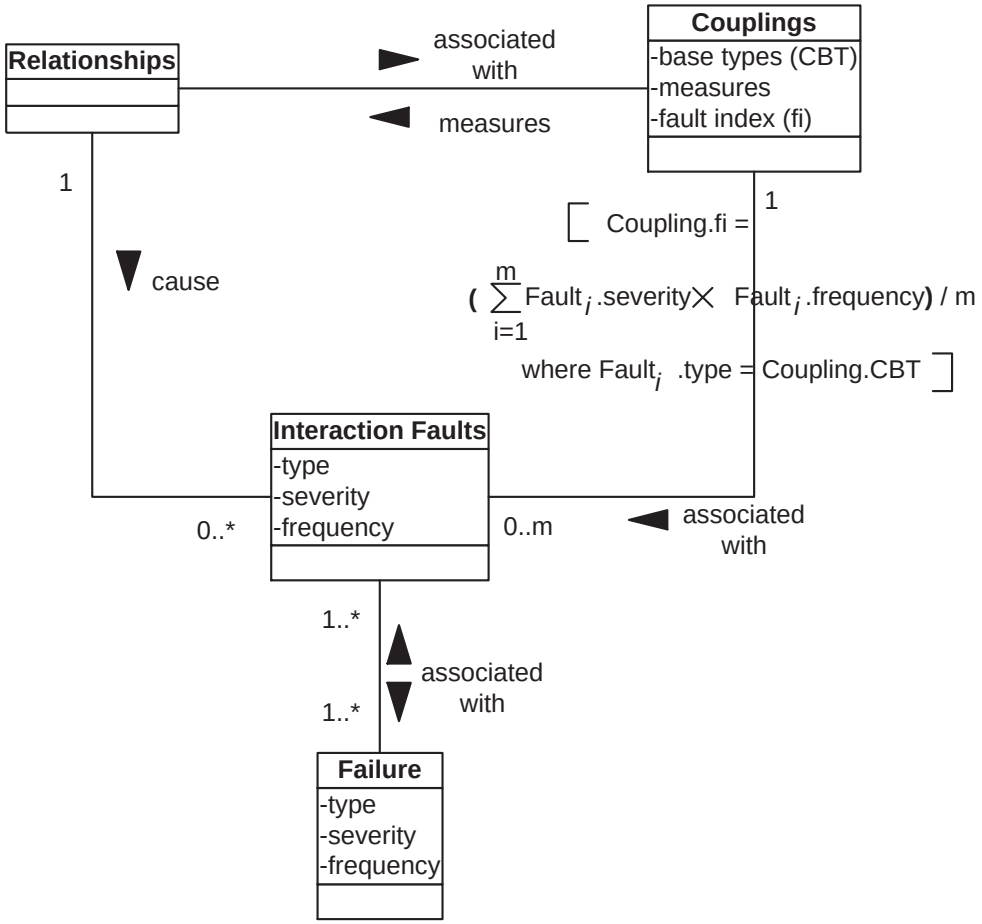

Figure 9.1: Conceptual Model for Correlation Among Relationships, Couplings and Faults in Object-Oriented Systems

figure shows that there is a many-to-many relationship between faults and failures; a fault can cause many failures and a failure can be caused by different types of faults. In addition to type, severity, and frequency, another attribute, percentages of faults causing a failure, can be ascribed to failures to more thoroughly reflect their association with faults. At the same time, the severity of a particular type of fault can be determined by the severity of failures that are caused by that fault.

The computation of a fault severity can be defined as follows:

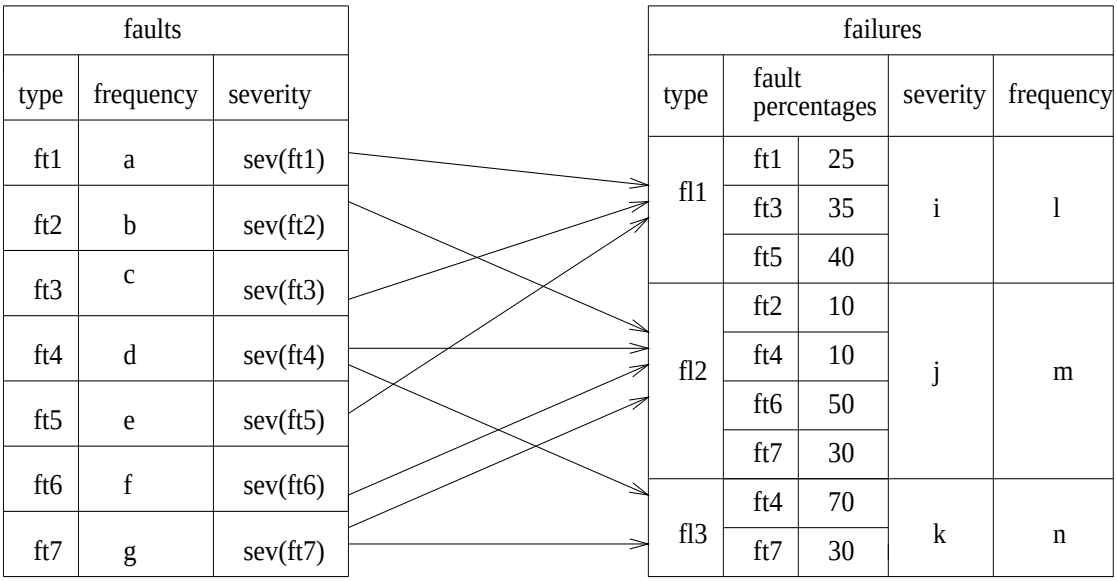

Figure 9.2: Conceptual Model for Relationships Between Faults and Failures

Let **FR** be a set of failures and **FT** be a set of faults. Let FR_i be the set of failures $< fr_1, fr_2, ...fr_n >$ that are caused by fault ft_i. The severity of fault ft_i is computed by the following formula:

$$ft_i.severity = \sum_{j=0}^{n} (fr_j.severity \times fr_j.percentage(ft_i)), \quad fr_j \in FR_i \tag{9.3}$$

The fault index, fi, of a coupling type is determined according to the faults that are associated with that particular coupling type. We use the following equation to compute the fault index of a coupling type:

Let FT_i be a set of faults $< ft_1, ft_2, ..., ft_n >$ that are associated with a coupling type CBT_i.

$$CBT_i.fi = \left[\sum_{i=1}^{n}(ft_i.severity \times ft_i.frequency)\right]/n \quad where \;\; ft_i \in FT_i. \qquad (9.4)$$

9.2 Coupling Levels and Coupling-based Fault Classification of Object-Oriented Software

Chapter 3 defined eight coupling base types for OO software:

1. *Inheritance Base (InhrCB)* type coupling measures inheritance-based interactions between two classes.

2. *Abstract Class Implementation Base (AbsCB)* type coupling measures abstract class implementation based interactions between two classes.

3. *Interface Implementation Base (IfimCB)* type coupling measures interface implementation based interactions between two classes.

4. *Composition Coupling Base (CompCB)* type coupling measures composition based interactions between two classes.

5. *Aggregation Coupling Base (AggrCB)* type coupling measures aggregation based interactions between two classes. *Exception Coupling Base (ExcpCB)* type coupling measures interaction between an exception throwing class and an exception handling class.

6. *Association Coupling Base (AssoCB)* type coupling measures association based interactions between two classes.

7. *Dependency Coupling Base (DpdnCB)* type coupling measures dependency based interactions between two classes.

Based on these coupling types, we develop a coupling-based fault classification for OO software. In functional programming, program faults are categorized as domain faults and computation faults [How76, Zei89, HOT97]. A *domain fault* occurs when, due to an error in control flow, a program generates incorrect output. A *computation fault* occurs when a program takes the correct path, but generates incorrect output because of faults in the computations along that path. Domain faults are further classified into two categories. A *missing path* fault is caused by a missing conditional or clause and the associated statements, and a *path selection* fault is caused by an incorrect decision at a predicate. Path-selection faults can result from an incorrect predicate (*predicate fault*) or from an incorrect assignment statement that propagates to a control point, leading to an incorrect decision (*assignment fault*).

We can extend the fault classification of functional programs for object-oriented software. Faults are categorized as local faults, interaction faults, or subtyping faults. A *local fault* occurs in a class or component, and it is independent of other classes or components in the system. An *interaction fault* occurs in the relationship between two classes or components. A *subtyping fault* may occur in an inheritance relationship. Figure 9.3 depicts the fault categorization in OO software.

Using our OO coupling types, we characterize each type of fault that occurs due to interaction or dependency as *inheritance*, *abstract class implementation*, *interface implementation*, *composition*, *aggregation*, *exception*, *association*, or *dependency* coupling faults. One open question about this model is how can we know that this model is comprehensive for interaction faults.

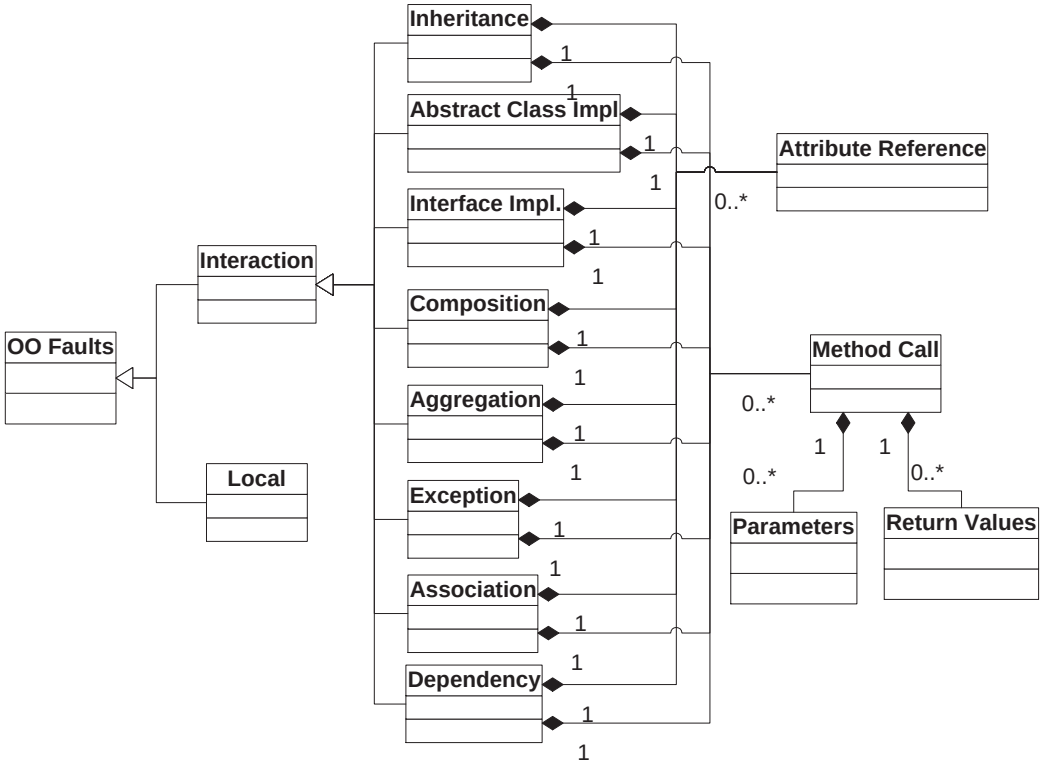

Figure 9.3: Coupling-based Object-Oriented Fault Classification

Inheritance faults share the same characteristics as **subtyping faults** as defined by Offutt et al. [OAW+01]. If class B uses subtype inheritance to inherit from class A, then it is semantically possible for any instance of B to be used (substituted) freely when an instance of A is expected. Offutt and his colleagues identified nine types of faults and anomalies that are related to *subtype* inheritance:

(1) Inconsistent type use

(2) State definition anomaly

(3) State definition inconsistency

(4) State defined incorrectly

(5) Indirect inconsistent state definition

(6) Anomalous construction behavior(1)

(7) Anomalous construction behavior(2)

(8) Incomplete (failed) construction (IC)

(9) State visibility anomaly (SVA)

The following description is taken from Offutt et al.'s paper [OAW+01].

For *Inconsistent Type Use (ITU)* faults, a descendant class does not override any inherited method. Thus, there can be no polymorphic behavior. Every instance of a descendant class C that is used where an instance of T is expected can only behave exactly like an instance of T. That is, only methods of T can be used. Any additional methods specified in C are hidden since the instance of C is being used as if it is an instance of T. However, anomalous behavior is still a possibility. If an instance of C is used in multiple contexts (that is, through coercion, say first as a T, then as a C, then a T again), anomalous behavior can occur if C has extension methods. In this case, one or more of the extension methods can call a method of T or directly define a state variable inherited from T. Anomalous behavior will occur if either of these actions results in an inconsistent inherited state.

In general, for a descendant class to be behaviorally compatible with its ancestor, the state interactions of the descendant must be consistent with those of its ancestor. That is, the refining methods implemented in the descendant must leave the ancestor in a state that is equivalent to the state that the ancestor's overridden methods would have left the ancestor in. For this to be true, the refining methods provided by the

descendant must yield the same net state interactions as each public method that is overridden. From a data flow perspective, this means that the refining methods must provide definitions for the inherited state variables that are consistent with the definitions in the overridden method. If not, then a potential data flow anomaly exists. Whether an anomaly actually occurs depends upon the sequences of methods that are valid with respect to the ancestor.

Any extension method that is called by a refining method must also interact with the inherited variables of the ancestor in a manner that is consistent with the ancestor's current state. Because the extension method provides a portion of the refining method's net effects, to avoid a data flow anomaly, the extension must not define inherited state variables in a way that would be inconsistent with the method being refined. Thus, the net effect of the extension method cannot be to leave the ancestor in a state that is logically different from when it was invoked. For example, if the logical state of an instance of a stack is currently not-empty/not-full, then execution of an extension method cannot result in the logical state spontaneously being changed to either empty or full. Doing so would preclude the execution of *pop* or *push* as the next methods in sequence.

The introduction of an indiscriminately named local state variable can easily result in a data flow anomaly where none would otherwise exist. If a local variable is introduced to a class definition where the name of the variable is the same as an inherited variable v, the effect is the inherited variable is hidden from the scope of the descendant (unless explicitly qualified, as in *super.v*). A reference to v by an extension or overriding method will refer to the descendant's v. This is not a problem if all inherited methods are overridden since no other method would be able to implicitly reference the inherited v. However, this pattern of inheritance is the exception rather

than the rule. There will typically be one or more inherited methods that are not overridden. A data flow anomaly might exist if a method that normally defines the inherited v is overridden in a descendant when an inherited state variable is hidden by a local definition.

Suppose an overriding method defines the same state variable v that the overridden method defines. If the computation performed by the overriding method is not semantically equivalent to the computation of the overridden method with respect to v, then subsequent state dependent behavior in the ancestor will likely be affected, and the externally observed behavior of the descendant will be different from the ancestor. While this problem is not a data flow anomaly, it is a potential behavior anomaly.

An inconsistent state definition can occur when a descendant adds an extension method that defines an inherited state variable. For example, consider the class hierarchy shown in Figure 9.4A where Y specifies a state variable x and method $m()$, and the descendant D specifies method $e()$. Since $e()$ is an extension method, it cannot be directly called from an inherited method, in this case *T::m()*, because $e()$ is not visible to the inherited method. However, if an inherited method is overridden, the overriding method (such as *D::m()* as depicted in Figure 9.4B) can call $e()$ and introduce a data flow anomaly by having an effect on the state of the ancestor that is not semantically equivalent to the overridden method (e.g. with respect to the variable *T::y* in the example). Whether an error occurs depends on which state variable is defined by $e()$, where $e()$ executes in the sequence of calls made by a client, and what state dependent behavior the ancestor has on the variable defined by $e()$.

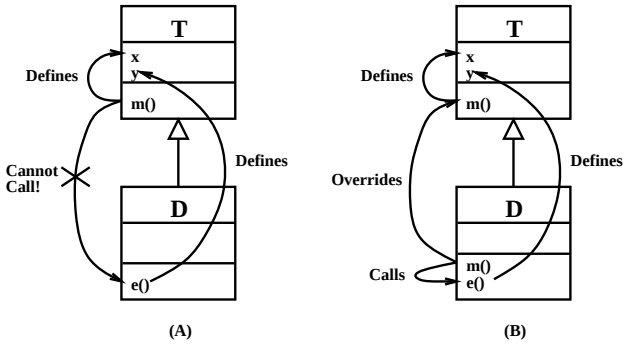

Figure 9.4: IISD: Example of Indirect Inconsistent State Definition.

9.3 Fault Index Computation for Couplings

Table 9.1 gives three coupling types and associated faults with their attributes. The values used for fault severity and frequency are based on intuition, and therefore should be considered speculative. We wish to use these data to show that it is possible to compute coupling fault indexes from fault attributes when data is available. In this example, we use a range [1..5] for fault severity where 5 is the most severe, and [1..100] for fault frequency.

Using equation 9.4 and the limited data available in Table 9.1, the fault indexes, fi, of coupling types are computed as follows:

$$\begin{aligned} CBT_{inheritance}.fi &= (5 \times 10 + 2 \times 15 + 2 \times 3 + 2 \times 20 + 3 \times 12 + 4 \times 5)/6 \\ &= 50 + 30 + 6 + 40 + 36 + 20)/6 \\ &= 182/6 \\ &= 30.33 \end{aligned}$$

$$
\begin{aligned}
CBT_{association}.fi &= (1 \times 10 + 2 \times 15 + 3 \times 6 + 4 \times 6 + 5 \times 3 + 3 \times 8 + 2 \times 15 + \\
&\quad 3 \times 7 + 1 \times 30 + 5 \times 2 + 1 \times 10 + 3 \times 5 + 2 \times 18)/13 \\
&= (10{+}10{+}18{+}24{+}15{+}8{+}10{+}21{+}12{+}10{+}10{+}30{+}24)/13 \\
&= 273/13 \\
&= 21
\end{aligned}
$$

$$
\begin{aligned}
CBT_{exception}.fi &= (5 \times 6 + 2 \times 4 + 2 \times 10 + 2 \times 5 + 4 \times 2) \\
&= 30 + 8 + 20 + 10 + 8)/5 \\
&= 76/5 \\
&= 15.2
\end{aligned}
$$

The fault index computation result shows that inheritance coupling has the highest fault index and the exception coupling has the lowest fault index. This order will be useful to develop a fault prediction model for a software system.

9.4 Summary

While we tried to reflect accurately the relationship between faults and failures, it became extremely difficult to enumerate the attributes of faults and failures by experimentation. First, it is almost impossible to enumerate all the faults and failures; especially the faults that cause a particular failure. Second, the frequency of a fault and the frequency of a failure are different concepts. However, frequencies of both faults and failures are hard to count. Third, severity of failures is a qualitative, not a quantitative, attribute. We have to consider the issue of assigning severity values

subjectively. Last, it is complicated to determine the percentages of faults that cause each failure.

We are searching for methods other than empirical evaluation to validate this approach. However, we believe this model has potential to help solve numerous problems in software engineering.

Table 9.1: **OO Coupling Levels and OO Faults - Example**

Coupling Level	**Fault**	**Fault Severity**	**Fault** Frequency	**Comment**
Inheritance	ITU	5	10	inconsistent type use
	SDA	2	15	state definition anomaly
	SDIH	2	3	state definition inconsistency due to state variable hiding
	SDI	2	20	sate defined incorrectly
	IISD	3	12	indirect inconsistent state definition
	IL	4	5	inheritance loops
Association	A	1	10	public method not used by object users
	B	2	15	message/object mismatch
	C	3	6	message sent to object without corresponding method
	D	4	6	message sent to wrong server object
	E	5	3	message parameters incorrect or missing resulting in wrong or failed binding
	F	3	8	message not implemented in the server
	G	2	15	formal and actual parameters inconsistent
	H	3	7	missing object
	I	1	30	unused object
	J	5	2	reference to undefined or deleted object
	K	1	10	missing initialization; incorrect constructor
	L	3	5	server contract violated
	M	2	18	incorrect visibility/scoping
Exception	ENC	5	6	Exception not caught
	EPOS	2	4	Exception propagates out of scope
	IRE	2	10	improperly raising an exception from server to client
	REIC	2	5	raising an exception under improper circumstances
	FRE	4	2	failure to raise an exception under proper circumstance

Chapter 10: CONCLUSION AND FUTURE WORK

This dissertation presented a new approach for Object-Oriented (OO) coupling analysis. This approach takes into account design level relationships among OO software components and their effects on implementation level couplings. Design level relationships are mapped onto implementation level couplings. The approach is based on the static analysis of object-oriented programs, and shows how to effectively measure relationships among components and apply the measures to specific problems in testing and maintenance.

The core theoretical results of this research were applied to three specific testing and maintenance problems. They can also be applied to other areas, such as web maintenance, coupling-based test case generation, fault analysis, etc. The following section discusses the contributions of this research in detail, and the final section of this chapter presents possible topics for future research.

10.1 Contributions

The main contribution of this research is theoretical. A key contribution is a technique for analyzing and measuring object-oriented couplings. The foundation of this technique is the distinction of relationships at the different level of abstractions.

This research also contributes **a set of coupling measures** for OO software. The coupling measures are defined using the unified OO coupling framework [BDW99]

with some modifications. The coupling measures are defined in a way that distinguishes high level relationships and low level connections in the measure. These measures are theoretically validated using mathematical properties of couplings, and also empirically shown to be applicable to concrete problems. This research has also mapped design level relationships to implementation level connections, and developed algorithms to compute coupling measures in source code.

Another contribution is the practical **application** of the coupling measures to the **Class Integration and Test Order (CITO)**, **Change Impact Analysis (CIA)**, and **Design Pattern Detection (DPD)** problems. In all three cases, this research developed **algorithms** to find optimal solutions to the problems using coupling measures.

For CITO, this research led to five results. (1) It found that whether the service provided by a class to its clients overlaps will result in different stubs. (2) A method to compute test stub complexities for classes using coupling measures was given. In particular, the method computes specific stub complexity for each client of a class and a total stub complexity for all clients. (3) A method to construct weighted object relation diagrams (WORD) using specific stub complexities on edges and total stub complexities on nodes was developed. (4) Algorithms were developed to eliminate cycles in WORDs. (5) Algorithms were developed to order classes for integration testing. The advantage of this approach is that the CITO problem is reduced to a weighted graph problem and the results are at least as good as the far more complicated genetic algorithm based approach.

For CIA, this research led to four results. (1) It analyzed the characteristics of changes in OO software and presented techniques to analyze their impacts. (2) Several

algorithms were developed to compute the impacts of different change categories. (3) *Change impact* and *sensitivity to change* metrics were defined to evaluate classes. (4) A kiviat diagram was used to visualize several metrics.

For DPD, this research led to three results. (1) A method to reduce the pattern detection scope was developed. The method first used packages and package pairs to divide a system into subsystems, then developed an algorithm to remove classes that cannot participate in the pattern. This further reduced the search scope. The algorithm used coupling measures to judge classes for possibility that they are part of a pattern. (2) Weighted graphs were used to represent the reduced subsystem and a pattern. This resulted in one graph for the subsystem and one for a pattern. (3) A graph similarity algorithm was used to determine if the subsystem matches a pattern.

This dissertation has also produced a **metamodel of couplings** in object-oriented system for analyzing and understanding the effects of relationships on couplings.

This research also resulted in a **proof of concept tool** that demonstrates the practicality and effectiveness of coupling-based analysis techniques.

Finally, this dissertation has contributed a **conceptual model for relationships, couplings, and faults in OO systems**.

This research has resulted in the following publications:

1. "Coupling-based Class Integration and Test Order", Aynur Abdurazik and Jeff Offutt. (AST 2006) [AO06]

2. "Using Coupling-based Weights for the Class Integration and Test Order Problem", Aynur Abdurazik and Jeff Offutt. Accepted for publication, Computer Journal [AO07]

3. "Quantitatively Measuring Object-Oriented Couplings", Jeff Offutt, Aynur Abdurazik, and Stephen R. Schach. Accepted for publication, Software Quality Journal [OAS07]

4. "An Analysis Tool for Coupling-based Integration Testing", Jeff Offutt, Aynur Abdurazik and Roger T. Alexander. (ICECCS '00) [OAA00]

Several papers are in various stages of preparation:

1. "Object-Oriented Coupling Measures for Testing and Maintenance" from chapter 3

2. "Coupling-based Change Impact Analysis" from chapter 7

3. "Coupling-based Design Pattern Detection" from chapter 8

4. "Coupling-based Class Ranking" from chapters 7 and 9

During my study, I have been involved in other research projects and have published the following papers:

1. "Generating Test Cases from UML Specifications", Jeff Offutt and Aynur Abdurazik. (UML 1999) - based on my MS thesis. [OA99]

2. "Using UML Collaboration Diagrams for Static Checking and Test Generation",Aynur Abdurazik and Jeff Offutt. (UML 2000) - based on my MS thesis. [AO00]

3. "Generating Test Data from State-based Specifications", Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. (STVR 2003) [OLAA03]

4. "Evaluation of Three Specification-based Testing Criteria", Aynur Abdurazik , Paul Ammann, Wei Ding and Jeff Offutt. (ICECCS 2000) [AADO00]

5. "Analyzing Software Architecture Descriptions to Generate System-level Tests", Aynur Abdurazik, Zhenyi Jin, Jeff Offutt, and Elizabeth L. White. (WESAS 2000) [AJWO00]

10.2 Future Work

The research reported in this thesis could potentially apply to web applications, integration fault analysis, component ranking, testing, and concurrent program analysis. The coupling analysis as presented could be used for some problems. For other problems, the same approach could be used, but different couplings would need to be identified and analyzed. The following subsections discuss some future research topics.

10.2.1 Application of Coupling Model to Web Applications

Web applications are accessed with a Web browser over a network such as the Internet or an intranet. Though many variations are possible, a common structure for web applications is a three-tiered application. In its most common form, a Web browser is the first tier, an engine using some dynamic Web content technology (such as ASP or ASP.NET, CGI, JSP, or PHP) is the middle tier, and a database is the third tier. The Web browser sends requests to the middle tier, which services them by making queries and updates against the database and generating a user interface. While there are some similarities with OO stand-alone applications, web applications have characteristics that are different than OO software. The exciting part is there are

some novel types of couplings. Coupling-based analysis of web applications would give researchers improved insight into the characteristics of web applications. It may also help promote testing and maintenance of web applications.

10.2.2 Coupling-based Fault Analysis

Chapter 9 associated OO couplings with OO faults. There are two possible future work directions for this research. One is to use statistical analysis and approximation techniques to find relationships between couplings and faults. Another is to carry out extensive empirical validation. This research has the potential of identifying and categorizing possible OO faults in a comprehensive way. It can also further validate coupling measures, and make it possible to use coupling measures in fault prediction models.

10.2.3 Comprehensive Empirical Validation of Three Specific Problems

Chapters 6, 7, and 8 discussed the application of coupling measures to three specific problems, class integration and test order, change impact analysis, and design pattern detection. The main contribution of this research was the theoretical foundation of the application. An open question is how well these results will scale to large systems and we hope to work on that problem soon. In the future, we plan to complete automation of this work and then carry out detailed experiments to fully assess the value of the technique.

10.2.4 Extension of Design Pattern Detection

The results from design pattern detection research are very promising. We can add elemental design patterns (EDPs) to the picture. EDPs are conceptual subcomponents of design patterns. As such, EDPs may even change the way we define coupling measures. Also, currently only three design patterns are identified. Adding all patterns is necessary for practical use, although this does not seem likely to lead new research results. Finally, other graph matching algorithm have been developed. An especially interesting one uses labeled graphs [CS]. Labeled graphs incorporate more information about the system on the graph than directed graphs. More information may help in developing decisive ways to detect design patterns. Also, it is worthwhile to try different graph matching algorithms and compare the results.

10.2.5 JCAT Enhancement

Currently, JCAT can compute couplings for connection types without distinguishing the coupling base types. The algorithms for CITO, CIA, and DPD are implemented separately, and have not been integrated into JCAT. JCAT can be enhanced to compute all measures and integrate application specific algorithms. Then, JCAT will not only have research value, but can be a useful tool for practitioners.

10.2.6 Coupling-based Reverse Engineering

This research produces enough details from source code to reverse engineer its design. The coupling information would particularly be useful in reverse engineering structural information, including the interactions among components. A pilot study has shown that it is easy to generate class diagrams from coupling tables using open

source graph visualization software, such as graphviz [gra] and jgraph [jgr].

10.2.7 Coupling-based Component Ranking

Ranking of classes or components is useful in assessing the reusability of a class and its relation to other classes. *Component Rank* is a method for ranking software components, based on analyzing actual use relations among the components and propagating the significance through the use relations [IYF+03, NIC06].

There is a possibility of ranking components/classes based on coupling measures. Coupling measures should help indicate how a component is used and its dependency on other classes. When ranking components and classes, two issues have to be considered: how a component is used and how it uses other components. The method applied for ranking also depends on the purpose of ranking. Because coupling measures reflect both how much a class is used and how much it depends on others, it could be useful in ranking classes for their reusability.

We plan to use change impact analysis metrics to develop a class ranking model for reuse, and compare our approach with other studies [IYF+03, NIC06].

10.2.8 Coupling-based Testing

A key area of research related to this dissertation is the generation of test cases that satisfy particular coupling-based criterion. Some couplings have been used in testing. Jin and Offutt used method call based coupling in their coupling based testing techniques [JO95], whereas Alexander and Offutt used inheritance and polymorphism [AO04]. This research can be extended to develop comprehensive coupling-based test criteria for OO software.

10.2.9 Coupling-based Analysis of Concurrent Software

Many object-oriented languages, such as Java and Eiffel, incorporate some type of threading mechanism. This results in greater complexity of relationships among software components. An interesting area of investigation that remains open is whether or not the coupling model and coupling measures would be applicable in the presence of multiple threads.

LIST OF REFERENCES

LIST OF REFERENCES

[AACGJ01] Herv Albin-Amiot, Pierre Cointe, Yann-Gal Guhneuc, and Narendra Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In *Proc. 16th Annual International Conference on Automated Software Engineering*, pages 166–173, July 2001.

[AADO00] Aynur Abdurazik, Paul Ammann, Wei Ding, and Jeff Offutt. Evaluation of three specification-based testing criteria. In *Proceedings of the Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00)*, pages 179–187, Tokyo, Japan, September 2000.

[ACPF01] G. Antoniol, G. Casazza, M. Di Penta, and R. Fiutem. Object-oriented design patterns recovery. *J. Syst. Softw.*, 59(2):181–196, 2001.

[AJWO00] Aynur Abdurazik, Zhenyi Jin, Liz White, and Jeff Offutt. Analyzing software architecture descriptions to generate system-level tests. In *Workshop on Evaluating Software Architectural Solutions – 2000*, Irvine, CA, May 2000.

[AMR] Francesca Arcelli, Stefano Masiero, and Claudia Raibulet. Elemental design patterns recognition in Java. In Kostas Kontogiannis, Ying Zou, and Massimiliano Di Penta, editors, *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)*, pages 196–205. IEEE Computer Society.

[AN91] A. Abran and H Nguyenkim. Analysis of maintenance work categories through measurement. In *Proceedings of the 7th International Conference on Software Maintenance*, pages 104–113, Sorrento, Italy, 1991.

[AO00] Aynur Abdurazik and Jeff Offutt. Using UML collaboration diagrams for static checking and test generation. In *Proceedings of the Third International Conference on the Unified Modeling Language (UML '00)*, pages 383–395, York, England, October 2000.

[AO04] Roger T. Alexander and Jeff Offutt. Coupling-based testing of O-O programs. *Journal of Universal Computer Science*, 10(4):391–427, April 2004. http://www.jucs.org/jucs_10_4/coupling_based_testing_of.

[AO06] Aynur Abdurazik and Jeff Offutt. Coupling-based class integration and test order. In *Workshop on Automation of Software Test (AST 2006)*, pages 50–56, Shanghai, China, May 2006.

[AO07] Aynur Abdurazik and Jeff Offutt. Using coupling-based weights for the class integration and test order problem. *Accepted for publication, Computer Journal*, 2007.

[Ari01] Erik Arisholm. *Empirical Asessement of Changeability in Object-Oriented Software*. PhD thesis, University of Oslo, 2001. ISSN 1501-7710, No. 143.

[Ari02] Erik Arisholm. Dynamic coupling measures for object-oriented software. In *Proceedings of the Eighth IEEE Symposium on Software Metrics (METRICS'02)*, pages 33–42. IEEE, June 2002.

[BA96] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, California, 1996.

[BBC+] Victor R. Basili, Lionel C. Briand, Steven E. Condon, Yong-Mi Kim, Walcélio L. Melo, and Jon D. Valett. Understanding and predicting the process of software maintenance release. In *Proceedings of the 18th International Conference on Software Engineering*, pages 464–474.

[BDW99] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, January/February 1999.

[Bei90] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.

[BF03] Zsolt Balanyi and Rudolf Ferenc. Mining design patterns from c++ source code. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 305, Washington, DC, USA, 2003. IEEE Computer Society.

[BFL02] Lionel C. Briand, J. Feng, and Yvan Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 43–50, Ischia, Italy, 2002. IEEE Computer Society Press.

[BGH+04] Vincent D. Blondel, Anahí Gajardo, Maureen Heymans, Pierre Senellart, and Paul Van Dooren. A measure of similarity between graph vertices: Applications to synonym extraction and web searching. *SIAM Rev.*, 46(4):647–666, 2004.

[BLW01] Lionel C. Briand, Yvan Labiche, and Yihong Wang. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. Technical report SCE-01-02, Careleton University, 2001.

[BLW03] Lionel C. Briand, Yvan Labiche, and Yihong Wang. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(7):594–607, July 2003.

[BP00] F. Bergenti and A. Poggi. Improving UML design using automatic design pattern detection. In *Proc. 12th. International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2000.

[BRJ98] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Publishing Company Inc., New York, NY, 1998. ISBN 0-201-57168-4.

[Bro96] Kyle Brown. Design reverse-engineering and automated design-pattern detection in smalltalk. Technical report, TR-96-07, Raleigh, NC, USA, 1996.

[BS90] Bonnie Berger and Peter W. Shor. Approximation alogorithms for the maximum acyclic subgraph problem. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 236–243, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[BWL99] Lionel C. Briand, Jürgen Wüst, and Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, Oxford, UK, 1999. IEEE Computer Society Press.

[CCHJ94] Nigel P. Capper, R. J. Colgate, J. C. Hunter, and M. F. James. The impact of object-oriented technology on software quality: Three case histories. *IBM Systems Journal*, 33(1):131–157, 1994.

[CK92] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1992.

[CKK+00] M. Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller, François Lustman, and Guy Saint-Denis. Design properties and object-oriented software changeability. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR '00)*, page 45, Washington, DC, USA, 2000. IEEE Computer Society.

[CKKL02] M. Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller, and François Lustman. A change impact model for changeability assessment in object-oriented software systems. *Science of Computer Programming*, 45(2):155–174, 2002.

[CLD+05] Gennaro Costagliola, Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Design pattern recovery by visual language parsing. *CSMR*, 00:102–111, 2005.

[CS] Pierre-Antoine Champin and Christine Solnon. Measuring the similarity of labeled graphs. In Kevin D. Ashley and Derek G. Bridge, editors, *ICCBR*, Lecture Notes in Computer Science, pages 80–95. Springer.

[dof07] Design Patterns. http://www.dofactory.com/Patterns/Patterns.aspx, February 2007. Data & Object Factory.

[Dos03] Gunjan Doshi. Best practices for exception handling. http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html, November 2003.

[ELS93] Peter Eades, Xuemin Lin, and W. F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993.

[EMM01] Khaled El Emam, Walcelio Melo, and Javam C. Machado. The prediction of faulty classes using object-oriented design metrics. *The Journal of Systems and Software*, 56(1):51–62, July 2001.

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

[FFBS04] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, Inc., Sebastopol, CA, October 2004.

[Fie06] Nathan Fiedler. JRGrep. http://www.bluemarsh.com/java/jrgrep/, October 2006.

[FM90] N. Fenton and A. Melton. Deriving structurally based software measures. *The Journal of Systems and Software*, 12(3):177–886, July 1990.

[FP97] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Company, Boston, MA, 1997.

[Fro07] FrontEndArt Ltd. http://www.frontendart.com, 2007.

[FW88] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[GAA04] Yann-Gaıel Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'04)*, Vancouver, British Columbia, Canada, October 2004. ACM Press.

[GHJV01] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Pearson Education, Indianapolis, IN, 2001.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1989.

[Gom00] Hassan Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison Wesley, Upper Saddle River, NJ, 2000.

[gra] Graphviz - graph visualization software. http://www.graphviz.org/.

[GSZ04] Yann-Gael Gueheneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 172–181, Washington, DC, USA, 2004. IEEE Computer Society.

[HB85] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, August 1985.

[HC90] W. Harrison and C. Cook. Insights on improving the maintenance process through software measurement. In *Proceedings of the International Conference on Software Maintenance (ICSM'90)*, pages 37–44, San Diego, USA, 1990.

[HHHL03] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic design pattern detection. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 94, Washington, DC, USA, 2003. IEEE Computer Society.

[HM92] M. J. Harrold and J. D. McGregor. Incremental testing of object-oriented class structures. In *14th International Conference on Software Engineering*, pages 68–80, Melbourne, Australia, May 1992. IEEE Computer Society Press.

[HOT97] Mary Jean Harrold, Jeff Offutt, and K. Tewary. An approach to fault modeling and fault seeding using the program dependence graph. *The Journal of Systems and Software*, 36(3):273–296, March 1997.

[How76] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, September 1976.

[IEE90] IEEE. *A Glossary of Software Engineering Terminology.* Institute of Electrical and Electronic Engineers, New York, 1990. ANSI/IEEE Std 610.12-1990.

[IYF+03] Katsuro Inoue, Reishi Yokomori, Hikaru Fujiwara, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto. Component rank: relative significance rank for software component search. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pages 14–24, Washington, DC, USA, 2003. IEEE Computer Society.

[Jal91] P. Jalote. *An Integrated Approach to Software Engineering.* Springer-Verlag, New York NY, 1991.

[jgr] Java graph visualization and layout. http://www.jgraph.com/.

[JO95] Zhenyi Jin and Jeff Offutt. Integration testing based on software couplings. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS 95)*, pages 13–23, Gaithersburg MD, June 1995. IEEE Computer Society Press.

[JO98] Zhenyi Jin and Jeff Offutt. Coupling-based criteria for integration testing. *The Journal of Software Testing, Verification, and Reliability*, 8(3):133–154, September 1998.

[JUn07] Lecture 17: Case Study: JUnit. http://ocw.mit.edu/NR/rdonlyres/Electrical-Engineering-and-Computer-Science/6-170Laboratory-in-Software-EngineeringFall2001/735992D6-3051-4B10-B3F9-30603975224A/0/lecture17.pdf, February 2007. Lecture Notes, MIT.

[KGH+94] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object oriented software maintenance. In *International Conference on Software Maintenance*, pages 202–211, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[KGH+95a] D. Kung, J. Gao, Pei Hsia, Y. Toyoshima, and C. Chen. A test strategy for object-oriented programs. In *19th Computer Software and Applications Conference (COMPSAC 95)*, pages 239–244, Dallas, TX, August 1995. IEEE Computer Society Press.

[KGH+95b] David Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima. Class firewall, test order, and regression testing of object-oriented programs. *Journal of Object-Oriented Programming*, 8(2):51–65, 1995.

[KGH+95c] David Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, Chris Chen, Young-Si Kim, and Young-Kee Song. Developing an object-oriented software testing and maintenance environment. *Communications of the ACM*, 38(10):75–87, 1995.

[KH81] D. Kafura and S. Henry. Software quality metrics based on interconnectivity. *The Journal of Systems and Software*, 2:121–131, 1981.

[LH93] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *The Journal of Systems and Software*, 23(2):111–122, 1993.

[LJ00] Hen-Ming Lin and Jing-Yang Jou. On computing the minimum feedback vertex set of a directed graph by contraction operations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 295–307, March 2000.

[LOA00] Michelle Lee, Jeff Offutt, and Roger T. Alexander. Algorithmic analysis of the impacts of changes to object-oriented software. In *Proceedings of the Thirty Fourth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '00)*, pages 61–70, Santa Barbara CA, August 2000.

[MCL03] Brian A. Malloy, Peter J. Clarke, and Errol L. Lloyd. A parameterized cost model to order classes for class-based testing of C++ applications. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'03)*, pages 353–364, Denver, Colorado, 2003. IEEE Computer Society Press.

[Met07] What is metamodeling? http://www.metamodel.com/staticpages/index.php?page=20021010231056977, February 2007.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, New Jersey, 1997.

[Mye74] G. Myers. *Reliable Software Through Composite Design*. Mason and Lipscomb Publishers, New York NY, 1974.

[NIC06] Blair Neate, Warwick Irwin, and Neville Churcher. Coderank: A new family of software metrics. In *Australian Software Engineering Conference (ASWEC'06)*, pages 369–378. IEEE Computer Society, 2006.

[OA99] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, pages 416–429, Fort Collins, CO, October 1999. Springer-Verlag Lecture Notes in Computer Science Volume 1723.

[OAA00] Jeff Offutt, Aynur Abdurazik, and Roger T. Alexander. An analysis tool for coupling-based integration testing. In *The Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00)*, pages 172–178, Tokyo, Japan, September 2000. IEEE Computer Society Press.

[OAS07] Jeff Offutt, Aynur Abdurazik, and Stephen R. Schach. Coupling-based maintenance metrics for object-oriented software. *Accepted for publication, Software Quality Journal*, 2007.

[OAW+01] Jeff Offutt, Roger Alexander, Ye Wu, Quansheng Xiao, and Chuck Hutchinson. A fault model for subtype inheritance and polymorphism. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 84–93, Hong Kong China, November 2001. IEEE Computer Society Press.

[Obj05] Object Management Group. *UML Superstructure Specification, v2.0*, August 2005. October, 2005 http://www.omg.org/docs/formal/05-07-04.pdf.

[OHK93] Jeff Offutt, Mary Jean Harrold, and P. Kolte. A software metric system for module coupling. *The Journal of Systems and Software*, 20(3):295–308, March 1993.

[OLAA03] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *The Journal of Software Testing, Verification, and Reliability*, 13(1):25–53, March 2003.

[Par] Terence Parr. ANother Tool for Language Recognition. http://www.antlr.org.

[PK98] L. Prechelt and Ch. Krämer. Functionality versus practicality: Employing existing tools for recovering structural design patterns. *Journal of Universal Computer Science*, 4(12):866–882, December 1998.

[PRT00] Dewayne E. Perry, Alexander Romanovsky, and Anand Tripathi. Current trends in exception handling. *IEEE Transactions on Software Engineering*, 26(9):817—819, September 2000.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, July 2004.

[Roc] Daniel Rocacher. On fuzzy bags and their application to flexible querying. *Fuzzy Sets and Systems*, (1):93–110.

[RST+04] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'04)*, pages 432–448, Vancouver, British Columbia, Canada, October 2004. ACM Press.

[RT01] Barbara Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE 01)*, pages 210–220, Snowbird, Utah, June 2001.

[RW88] Charles Rich and Richard C. Waters. The programmer's apprentice: A research overview. *Computer*, 21(11):10–25, 1988.

[SB91] R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, February 1991.

[SG01] Gregory Swanson and Lee Globus. Software metrics visualization for a graphical programming environment. http://www.evaluationengineering.com/archive/articles/0301soft.htm, 2001.

[SMC74] W. Stevens, G. Meyers, and L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[Som95] I. Sommerville. *Software Engineering*. Addison-Wesley Publishing Company Inc., 5th edition, 1995.

[SS] Jason McC. Smith and P. David Stotts. SPQR: Flexible automated design pattern extraction from source code. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 03)*, pages 215–224. IEEE Computer Society.

[SS02] Jason McC. Smith and David Stotts. Elemental design patterns: A formal semantics for composition of OO software architecture. In *27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, pages 183–190, October 5-6 2002.

[Sta97] Richard P. Stanley. *ENUMERATIVE COMBINATORICS, volume 1.* Cambridge University Press, 1997.

[Tar72] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[TCS05] Nikolaos Tsantalis, Alexander Chatzigeorgiou, and George Stephanides. Predicting the probability of change in object-oriented systems. *IEEE Transactions on Software Engineering*, 31(7):601–614, July 2005.

[TCSH06] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, 2006.

[TD97] Kuo-Chung Tai and F.J. Daniels. Test order for inter-class integration testing of object-oriented software. In *The Twenty-First Annual International Computer Software and Applications Conference (COMPSAC '97)*, pages 602–607, Santa Barbara CA, 1997. IEEE Computer Society.

[TJJM00] Yves Le Traon, Thierry Jéron, Jean-Marc Jézéquel, and Pierre Morel. Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability*, pages 12–25, March 2000.

[TZ81] D. A. Troy and S. H. Zweben. Measuring the quality of structured designs. *The Journal of Systems and Software*, 2:112–120, 1981.

[Vok] Marek Vokác. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Transactions on Software Engineering*, (12):904–917.

[Vok06] Marek Vokác. An efficient tool for recovering design patterns from C++ code. *Journal of Object Technology*, 5(1):139–157, Jan/Feb 2006.

[War99] Ian Warren. *The Renaissance of Legacy Systems.* Springer-Verlag, London, 1999.

[Wen03] Lothar Wendehals. Improving design pattern instance recognition by dynamic analysis. In *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA'03)*, Portland, USA, May 2003.

[Wey88] E. J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, September 1988.

[WK00] F. George Wilkie and Barbara A. Kitchenham. Coupling measures and change ripples in C++ application software. *The Journal of Systems and Software*, 52(2-3):157–164, 2000.

[YCM78] S. S. Yau, J. S. Collofello, and T. M. MacGregor. Ripple effect analysis of software maintenance. In *Proceedings of IEEE 2nd Annual International Computer Software and Applications Conference (COMPSAC '78)*, pages 60–65, Santa Barbara CA, 1978. IEEE Computer Society.

[Zei89] Steven J. Zeil. Perturbation techniques for detecting domain errors. *IEEE Transactions on Software Engineering*, 15:737–746, June 1989.

Appendix A: Unified Framework for Coupling - Definitions of Terms

We use the following 17 definitions from the unified framework for coupling measurement by Briand, Daly, and Wüst [BDW99].

DEFINITION 1 System, Classes, Inheritance and other Relationships.

An object-oriented system consists of a set of classes, C. There can exist inheritance relationships between classes such that for each class $c \in C$ let

- *Parents(c)* $\subset C$ *be the set of parent classes of class c.*
- *Children(c)* $\subset C$ *be the set of children classes of class c.*
- *Ancestors(c)* $\subset C$ *be the set of ancestor classes of class c.*
- *Descendants(c)* $\subset C$ *be the set of descendent classes of class c.*

DEFINITION 1-1 System, Classes, Interfaces, Abstract Classes, Regular Classes, Stable Classes, and Unstable Classes.

An object-oriented system consists of a set of classes, C.

- $IN \subset C$ *be the set of Interfaces.*
- $AC \subset C$ *be the set of abstract classes.*
- $RC \subset C$ *be the set of regular classes.*
- $C = IN \cup AC \cup RC$, *and* $IN \cap AC \cap RC = \emptyset$.

- *$SC \subset C$ be the set of stable classes.*
- *$UC \subset C$ be the set of unstable classes.*
- *$C = SC \cup UC$.*

DEFINITION 1-2 UML Relationships.

An object-oriented system consists of a set of classes, C. There can exist relationships between classes such that for each class $c \in C$ let

- *Compositions(c) $\subset C$ be the set of classes that have composition relationship with class c.*
- *Aggregations(c) $\subset C$ be the set of classes that have aggregation relationship with class c.*
- *Associations(c) $\subset C$ be the set of classes that have association relationship with class c.*
- *Dependencies(c) $\subset C$ be the set of classes that have dependency relationship with class c.*
- *Exceptions(c) $\subset C$ be the set of exception handler classes that handles exceptions thrown by class c.*
- *Externals(c) $\subset C$ be the set of classes that shares external media with class c.*

DEFINITION 2 Methods of a Class.

For each class $c \in C$ let $M(c)$ be the set of methods of class c.

DEFINITION 3 Declared and Implemented Methods.

For each class $c \in C$, let

- *$M_D(c) \subseteq M(c)$ be the set of methods declared in c, i.e., methods that c inherits but does not override or virtual methods of c.*
- *$M_I(c) \subseteq M(c)$ be the set of methods implemented in c, i.e., methods that c inherits but overrides or non-virtual non-inherited methods of c.*
- *where $M(c) = M_D(c) \cup M_I(c)$ and $M_D(c) \cap M_I(c) = \phi$.*

DEFINITION 4 Inherited, Overriding, and New Methods.

For each class $c \in C$, let

- *$M_{INH}(c) \subseteq M(c)$ be the set of inherited methods of c.*
- *$M_{OVR}(c) \subseteq M(c)$ be the set of overriding methods of c.*
- *$M_{NEW}(c) \subseteq M(c)$ be the set of non-inherited, non-overriding methods of c.*

DEFINITION 5 $M(C)$. The Set of all Methods.

$M(C)$ is the set of all methods in the system and is represented as $M(C) = \cup_{c \in C} M(c)$.

DEFINITION 6 Parameters.

For each method $m \in M(C)$, let Par(m) be the set of parameters of method m.

DEFINITION 7 SIM(m). The Set of Statically Invoked Methods of m.

Let $c \in C$, $m \in M_I(c)$, and $m' \in M(C)$. Then $m' \in SIM(m) \Leftrightarrow \exists d \in C$ such that $m' \in M(d)$ and the body of m has a method invocation where m' is invoked for an object of static type class d.

DEFINITION 7-1 SIM(c,d). The Set of Statically Invoked Methods of d by c.

Let $c \in C, d \in C$, and $m \in M_I(d)$. Then $m \in SIM(c,d) \Leftrightarrow \exists m \in C(d)$ such that the body of c has a method invocation where m is invoked for an object of static type class d.

DEFINITION 8 NSI(m,m'). The Number of Static Invocations of m' by m.

Let $c \in C$, $m \in M_I(c)$, and $m' \in SIM(m)$. $NSI(m,m')$ is the number of method invocations in m where m' is invoked for an object of static type class d and $m' \in M(d)$.

DEFINITION 8-1 NSI(m,m'). The Number of Static Invocations of m' by m.

Let $c \in C$, $m \in M_I(c)$, and $m' \in SIM(m)$. $NSI(m,m')$ is the number of method invocations in m where m' is invoked for an object of static type class d and $m' \in M(d)$.

DEFINITION 9 PIM(m). The Set of Polymorphically Invoked Methods of m.

Let $c \in C$, $m \in M_I(c)$, and $m' \in M(C)$. Then $m'PIM(m) \Leftrightarrow \exists d \in C$ such that $m' \in M(d)$ and the body of m has a method invocation where m' may, because of polymorphism and dynamic binding, be invoked for an object of dynamic type d.

DEFINITION 10 NPI(m,m'). The Number of Polymorphic Invocations of m' by m.

Let $c \in C$, $m \in M_I(c)$, and $m' \in PIM(m)$. $NPI(m, m')$ is the number of method invocations in m where m' can be invoked for an object of dynamic type class d and $m' \in M(d)$.

DEFINITION 11 Declared and Implemented Attributes.

For each class $c \in C$, let $A(c)$ be the set of attributes of class c. $A(c) = A_D(c) \cup A_I(c)$ where

- *$A_D(c)$ is the set of attributes declared in class c (i.e., inherited attributes).*
- *$A_I(c)$ is the set of attributes implemented in class c (i.e., non-inherited attributes).*

DEFINITION 12 $A(C)$. The Set of all Attributes.

$A(C)$ is the set of all attributes in the system and is represented as $A(C) = \cup_{c \in C} A(c)$.

DEFINITION 13 $AR(m)$.

For each $m \in M(C)$ let $AR(m)$ be the set of attributes referenced by method m.

DEFINITION 14 Basic Types and User-Defined Types.

- *BT is the set of built-in types provided by the programming language (e.g., integer, real, character, string).*
- *UDT is the set of user-defined types (e.g., records, enumerations, but not classes).*

- *CT is the set of built-in or user-defined collection types (e.g., array, list, set, hashtable).*

DEFINITION 15 T The Set of Available Types.

The set $\mathbf{T}$ *of available types in system is* $\mathbf{T} = BT \cup UDT \cup C$.

DEFINITION 16 Types of Attributes and Parameters.

For each attribute $a \in A(C)$ *the type attribute* $\mathbf{a}$ *is denoted by* $T(a) \in \mathbf{T}$. *For each method* $m \in M(C)$ *and each parameter* $v \in Par(m)$ *the type of parameter* v *is denoted by* $T(v) \in \mathbf{T}$.

DEFINITION 17 Uses.

Let $c \in C$, $d \in C$. $uses(c, d) \Leftrightarrow (\exists m \in M_I(c) : \exists m' \in M_I(d) : m' \in PIM(m)) \vee (\exists m \in M_I(c) : \exists a \in A_I(d) : a \in AR(m))$.

A class c uses a class d if a method implemented in class c references a method or an attribute implemented in class d.

DEFINITION 18 External Media.

For each class $c \in C$, *let* $E(c)$ *be the set of external files and devices that used by class* c.

DEFINITION 19 $D(C)$. The Set of all External Media (Device and Files).

$D(C)$ *is the set of all external files and devices in the system and is represented as* $D(C) = \cup_{c \in C} E(c)$.

DEFINITION 19 $E(C)$. The Set of Exceptions.

$E(C)$ is the set of classes in the system.

DEFINITION 19 $E(c)$. Exceptions.

$E(c)$ is the set of exceptions that generated by class c.

DEFINITION 20 Constructors of a Class.

For each class $c \in C$ let $CR(c)$ be the set of constructors of class c.

DEFINITION 21 Inherited, Overriding, and New Constructors.

For each class $c \in C$, let

- *$CR_{INH}(c) \subseteq CR(c)$ be the set of inherited constructors of c.*
- *$CR_{OVR}(c) \subseteq CR(c)$ be the set of overriding constructors of c.*
- *$CR_{NEW}(c) \subseteq CR(c)$ be the set of non-inherited, non-overriding constructors of c.*

DEFINITION 22 $NI(m, m', v)$. Number of invocation of m' by m through variable v.

Let $c \in C$, $m \in M_I(c)$, and $m' \in M(d)$. $NI(m, m', v)$ is the number of method invocations in m where m' is invoked for an object of static type class d through variable v where $v \in V_{as}$.

DEFINITION 23 $NDI(m, m')$. Number of direct invocation of m' by m.

Let $c \in C$, $m \in M_I(c)$, and $m' \in M(d)$. $NDI(m, m')$ is the number of direct invocations of m' in m.

Appendix B: Sample Implementation of Design Patterns

This appendix presents sample implementation of the Adapter, Composite, and Observer patterns.

B.1 Adapter Pattern Structure Java Sample Implementation

The following structural code demonstrates the Adapter pattern which maps the interface of one class onto another so that they can work together. These incompatible classes may come from different libraries or frameworks.

```
// Adapter pattern -- Structural example

  class Client  {
    public static void main(String args[])
    {
      // Create adapter and place a request
      Target target = new Adapter();
      target.request();
    }
  }
  // "Target"
```

```
public interface Target
{
  public void request();
}

// "Adapter"

class Adapter implements Target
{
  private Adaptee adaptee = new Adaptee();

  public void request()
  {
    // Possibly do some other work
    // and then call specificRequest()
    adaptee.specificRequest();
  }
}

// "Adaptee"

class Adaptee
{
  public void specificRequest()
  {
    System.out.println("Called specificRequest().");
  }
```

```
  }

Output from running Client class:
Called specificRequest().
```

B.2 Composite Pattern Structure Java Sample Implementation

The following structural code demonstrates the Composite pattern which allows the creation of a tree structure in which individual nodes are accessed uniformly whether they are leaf nodes or branch (composite) nodes.

```
// Composite pattern -- Structural example

  // Client

  public class Client
  {
     Component comp;
     public Client( Component comp )
     {
        this.comp = comp;
     }

    public void printout()
    {
        comp.print();
    }
```

```
}

// "Component"

public abstract class Component
{
  public void add(Component c)
  {
     throw new UnsupportedOperationException();
  }
  public void remove(Component c)
  {
     throw new UnsupportedOperationException();
  }
  public void print()
  {
     throw new UnsupportedOperationException();
  }
}

// "Composite"

class Composite extends Component
{
  String name;
  private ArrayList children = new ArrayList();

  // Constructor
```

```
  public Composite(String name)
  {
    this.name = name;
  }

  public void add(Component component)
  {
    children.add(component);
  }

  public void remove(Component component)
  {
    children.remove(component);
  }

  public void print()
  {
    System.out.println("Composite");
   }
  }
}

// "Leaf"

public class Leaf extends Component
{
  String name;
```

```
  // Constructor
  public Leaf(String name)
  {
     this.name = name;
  }

  public void print()
  {
    System.out.println("Leaf");
  }
}
```

B.3 Observer Pattern Structure Java Sample Implementation

The following structural code demonstrates the Observer pattern in which registered objects are notified of a state change and updated accordingly.

```
class MainApp
{
  pubilc static void main(String args[])
  {
    // Configure Observer pattern
    ConcreteSubject s = new ConcreteSubject();

    s.registerObserver(new ConcreteObserver(s,"X"));
```

```
        s.registerObserver(new ConcreteObserver(s,"Y"));
        s.registerObserver(new ConcreteObserver(s,"Z"));

        // Change subject and notify observers
        s.setState("ABC");
        s.notify();

    }
  }

public interface Subject {
        public void registerObserver(Observer o);
        public void removeObserver(Observer o);
        public void notifyObservers();
}

public interface Observer {
        public void update();
}

public class ConcreteSubject implements Subject {
        private ArrayList observers;
        private String state;

        public ConceretSubject() {
                observers = new ArrayList();
        }
```

```
public void registerObserver(Observer o) {
        observers.add(o);
}

public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
                observers.remove(i);
        }
}

public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
                Observer observer = (Observer)observers.get(i);
                observer.update();
        }
}

public void setState( String newState )
{
   state = newState;
}
public String getState()
{
   return state;
}
```

```
}

public class ConcreteObserver implements Observer{
        ...
        private ConcreteSubject concreteSubject;
        String observerState;
        String name;

        public ConcreteObserver(ConcreteSubject concreteSubject,
                                                    String name) {
                this.concreteSubject = concreteSubject;
                this.name = name;
                concreteSubject.registerObserver(this);
        }

        public void update() {

           observerState = concreteSubject.getState();

           System.out.println( "Observer " + name +
                           "'s new state is: " + observerState );

        }

}
```