Dependency Injection and Mock on Software and Testing

Author: Mengkeang Veng
Supervisor: Jonas Sjöström, Ph.D.

Department of Informatics and Media

June 2014
Abstract

Dependency Injection and Mock on Software and Testing

Software testing has been integrated within software development life cycle due to its importance in assuring software quality, software safety, and customers’ satisfaction. However, problems in software testing are prominent among software developers as system grows in size and complexity. Dependency injection becomes an appealing solution for developers with its practicality to improve software design, improve testability, and enable mock testing technique. The study aims to discover the extent to which the dependency injection facilitates software design and software testing. In addition, the effect of mock practice on testing is also assessed.

Metrics for investigation are defined and measured on various aspects of two systems. The two systems are selected and developed based on the same user requirements, development technologies, and methodologies. By comparing the two systems against the investigated metrics, we aim to reveal whether the dependency injection improve the code design. Then four test suites from both systems are evaluated in relation to testability.

The results demonstrate that the dependency injection does not seem to improve the code design if comparing on the selected metrics. Even though it does not score better, its effect is evident in other software aspects. The testability of the two systems is similar and suffers from the same problem. Meanwhile, mock helps assist software testing and improve testability. The effect of mock technique can be witnessed, especially when it is applied with other test techniques. Explanations and discussions on these findings are addressed in the paper.

Keywords: dependency injection, mock, software testing, testability, software metrics
Acknowledgements

I would like to thank Jonas Sjöström, Ph.D. for his permission to work with the Codisclose project. His supportive comments and feedback on the thesis work are valuable in addition to his time and effort to improve the overall work.

I really appreciate peer reviews from Jakob Malmstedt and Marcus Holmberg. Their input comments and critiques on the content, language-related issues, and project implementation help better the thesis work. My thanks to them all.

Another acknowledgement is given to Anna Borgström from the Language Workshop, Uppsala University. The consultation service is exceptional along with her feedback to identify various grammatical and sentence structure error in the paper.

This publication is part of my research work during the scholarship period fully supported by the Swedish Institute. I would also like to express my gratitude towards the Swedish Institute for funding my master’s degree study at Uppsala University.
# Contents

Abstract i  
Acknowledgements ii  
List of Figures v  
List of Tables vi  
List of Code Listings vii  

1 Introduction 1  
  1.1 Motivation ............................................................ 2  
  1.2 Goal and Objectives .................................................... 2  
  1.3 Methodology ............................................................. 3  
  1.4 Conceptual Framework .................................................. 4  
  1.5 Paper Structure .......................................................... 6  

2 Literature Review 7  
  2.1 Software Testing .......................................................... 7  
  2.2 Test Coverage and Test Adequacy ....................................... 10  
  2.3 Testability, Complexity, and Maintainability ......................... 14  
  2.4 Dependency Injection ..................................................... 16  
    2.4.1 Constructor Injection ............................................... 17  
    2.4.2 Property Injection .................................................. 18  
    2.4.3 Method Injection and Interface Injection .......................... 20  
    2.4.4 Dependency Injection Container .................................... 21  
    2.4.5 Benefits of Dependency Injection .................................. 23  
  2.5 Mock ................................................................. 25  
    2.5.1 Mock Pattern ...................................................... 25  
    2.5.2 Benefits of Mock ................................................... 28  

3 Metrics 30  
  3.1 Object-Oriented Notations .............................................. 30  
  3.2 Size-Related Metrics ................................................... 31  
  3.3 Cyclomatic Complexity ................................................ 32
## List of Figures

1.1 Conceptual Framework ................................................. 5

2.1 The V-Model (Ammann & Offutt, 2008, p. 8) ......................... 8

2.2 CollectionRepository UML Diagram ................................. 24

4.1 Codisclose Analysed Histogram ..................................... 41

4.2 Codisclose Process Overview ....................................... 42

4.3 Codisclose Database Diagram ....................................... 43

4.4 Codisclose and Test Project with Back-end Databases ............ 47

4.5 CodiscloseDI and Test Projects with Back-end Databases ....... 51
List of Tables

4.1 Codisclose Project Structure .................................. 45
4.2 CodiscloseDI Project Structure ............................... 49
4.3 Namespaces for Comparison on Various Metrics ............ 52
4.4 Test Project Namespaces for Comparison on Various Metrics 52

5.1 Size-Related Metrics on Namespace and Application Level ...... 55
5.2 Cyclomatic Complexity on Namespace and Application Level ...... 56
5.3 Maintainability Index on Namespace and Application Level ...... 57
5.4 Repeating and Unique Coupling on Namespace and Application Level 58
5.5 Max LCOM2 and Max LCOM3 on Namespace Level ............ 60
5.6 Test Suite Metrics of the Four Test Projects ............... 61
5.7 Code and Branch Coverage on CodiscloseDI from DIMockTest .. 62
5.8 Code and Branch Not Covered from CoUnitTest and DIUnitTest 64
5.9 Code and Branch Coverage on CodiscloseDI from DIMockUnitTest 65
5.10 Summary of Metrics on Codisclose and CodiscloseDI Project .... 66

A.1 Codisclose Project Metrics on Class, Namespace and Application Level 76
A.2 CodiscloseDI Project Metrics on Class, Namespace and Application Level 77

B.1 Codisclose Code and Branch Coverage from CoUnitTest .......... 80
B.2 CodiscloseDI Code and Branch Coverage from DIUnitTest .......... 81
B.3 CodiscloseDI Code and Branch Coverage from DIMockTest .......... 81
B.4 CodiscloseDI Code and Branch Coverage from DIMockUnitTest .... 82

C.1 Test Suite Metrics .................................................. 83
List of Code Listings

2.1 DeleteCollectionByID with Two Branches .................................. 10
2.2 SaveTextFiles with Six Basic Program Paths ............................... 11
2.3 Constructor Injection on CollectionRepository Class .................... 17
2.4 Property Injection on CollectionRepository with Default XMLContext .. 18
2.5 Method Injection and Interface Injection via BuildIndex Method ........ 20
2.6 Register Dependency Rules with Ninject .................................... 22
2.7 Resolve Dependency Rules with Ninject .................................... 22
2.8 Manual Registering and Resolving Dependencies ......................... 23
2.9 CollectionRepository Class With Its Two Methods ....................... 26
2.10 Unit Test on GetCollections with Stub ..................................... 27
2.11 Unit Test on DeleteCollectionByID with Mock .......................... 27
4.1 Typical Test Structure of Microsoft Unit Testing Framework ............ 46
Dedicated to my family for staying with me and encouraging me to complete the thesis work.
Chapter 1

Introduction

In today’s software development, software testing has been considered as one of the priority tasks that software developers have to perform. Not only does testing ensure software quality, but also it guarantees customers’ satisfaction and trust on the software. To achieve the promising software quality, satisfaction and trust, software developers have invested on software designs which are favourable to software testing. However, the challenges are still prominent as software grows in size and complexity. Most software has many components such as external database, web service, and file resources which interact with one another to perform the intended functionalities. Because there are interactions between various components, testing can prove to be a challenging and tedious work for developers to perform.

Due to difficulties in testing software whose components intertwine into complexity, testing techniques and methodologies have been formalised to alleviate the problems. One of the methodologies is Test Driven Development (TDD), a principle from Extreme Programming. The idea behind TDD is that instead of writing production code first, developers write test case first. Following TDD, developers are confident that various sections of the software are testable since TDD enforces developers to design manageable code unit (Beck, 2001). However, this does not imply that TDD ensures better code design. TDD gives the minimal mechanism in writing code that works.

Even though developers follow TDD principle, without design principles and patterns in the first place, software testing is still a challenge task. This brings us into the search of design pattern which eases software testing in general. One of the design patterns, which associates with software testing, is Dependency Injection. Dependency injection is claimed to increase testability of software because by following the dependency injection pattern, external resources such as database and web service can be mocked or faked.
to allow testing software components in isolation from one another. For this reason, we will dive into details on the dependency injection and mock in the study.

### 1.1 Motivation

The paper is motivated by the increasing need of software safety along with the impact on the environment that the software in question will be operated. Because of the great involvement of software in the society, business process, human life, and environment, any small error or bug can cause tremendous effect afterwards. It should be noted that software development is an error-prone process; thus, testing is an effective mean to detect any error or bugs and guarantee software quality.

Another motivation is related to time and effort in maintaining software. It is typical that software requirements change from time to time as business process evolves. Because of requirement changes, developers need to ensure that changing, removing or adding new components will work and operate smoothly as intended. Testing gives us solutions to detect any incompatible components when changes happen. Without testing, developers could not be certain that the software will work.

However, as testing becomes important in software quality, the cost related to testing covers about 25 per cent to 50 per cent of the whole system development cost (Jungmayer, 2004; Myers et al., 2011). As Freeman et al. (2013) suggested, time and effort are spent on tweaking existing code, locating bugs, and maintaining code to work with the changing requirements. For this reason, the demand on design patterns for testable code is increasing. One candidate among the other patterns is dependency injection. Proponents of the dependency injection argue that the pattern makes software easier to be tested in addition to low coupling, high cohesion, and maintainable code (Johnson & Foote, 1988; Martin, 1996; Prasanna, 2009; Seemann, 2012).

Despite the claim to endorse the benefits of the dependency injection on software design and software testing, there are a few research studies on the topic dependency injection in the literature. Consequently, we are motivated to fill the gap of knowledge on the topic and to measure the extent to which the dependency injection facilitates software design and testing in particular.

### 1.2 Goal and Objectives

As stated above, the dependency injection is claimed to improve software design and testability, the paper aims to discover whether the assertion holds and to measure the
extent of facilitation. Because the dependency injection enables mock practice to per-
form testing in isolation, measurement the effect of mock practice on software testing
will be included in the paper.

To reach the overall goal, we attempt to answer four research questions:

1. To what extent does dependency injection improve code structure design?
2. To what extent does dependency injection facilitate software testing?
3. To what extent does mock facilitate software testing?
4. To what extent do dependency injection and mock facilitate software testing?

The research questions assume that there is positive effect of the dependency injection
and mock on code structure design and software testing respectively. The actual result
will be illustrated with collected data.

1.3 Methodology

To answer the above research questions, the study will be implemented by following dif-
f erential research design. This research approach can be called “experiment” in which re-
searchers have control over or manipulate variables under study. In differential research,
variables are categorised into independent variable and dependent variable (Graziano
& Raulin, 2010; Oates, 2006). Researchers aim to control the independent variables
and to measure the effect on the dependent variables. For this reason, the independent
variable can be considered as “cause”, while the dependent variable is “effect” (Oates,
2006). Furthermore, the approach normally compares two similar groups: one with the
presence of the independent variables and one with the absence.

In relation to our study, the independent variables are the dependency injection and
mock, and the dependent variables are software and testing. To measure the extent of
facilitation of the independent variables on the dependent variables, the study involves
itself in comparing two similar software systems in which one system is designed based
on the dependency injection pattern, and the other follows an ad hoc design strategy.
Moreover, the two systems under experiment follow the exact user specifications with
similar software design methodology. This helps ensure that the comparison aims to
reveal the differential effect of the dependency injection pattern on the software system.

The first research question deals with measuring the differences on general code structure
design between the two design patterns. The measurement will be based on the common
software metrics available from the source code. The concept of system complexity and maintainability is closely related in defining the metrics. It should be noted that in a broad concept, the dependent variable “software” cannot be measured. The two characteristics (complexity and maintainability) are used to measure the term “software” instead.

Since another study goal is related with software testing and specifically, to answer research question two, three, and four, the two systems are going to be tested with various test suites based on different testing practices. The ad hoc designed system will have its own test suite. For the system with the dependency injection pattern, three test suites are prepared for testing with the inclusion of mock practices. Then the test suite of the ad hoc designed system is compared with the three test suites designed for the other system.

In other words, the comparison in the study is twofold. First, the two systems are compared to measure the extent of facilitation of the dependency injection on software complexity and maintainability. Second, the test suites from both systems are compared to measure the extent of the effect of the dependency injection and mock on software testing. Therefore, the comparisons of software testability and software testing are quantified based on various metrics of software measurement.

It should be noted that one system with the ad hoc design pattern has been further developed by a group of students from the Department of Informatics and Media, Uppsala University. The system contains its own test suite with the exception that the test suite covers a small part of the whole system only. As the result, more test cases will be added into the existing test suite to expand the system test coverage. The other system is redeveloped with the dependency injection pattern as the core of the structure design. After the new system is completely developed, the three test suites are created afterwards.

### 1.4 Conceptual Framework

Before we dive into details on the literature review in this paper, it is important to discuss the relationship of different concepts related to the overall study. The fundamental concepts in the study are depicted in figure 1.1 below.

Figure 1.1 illustrates the relationship among the key concepts. The relationship is simple in meaning: one concept has effect on other concepts as denoted by the arrow. Initially, structure design refers to the internal code structure of software system. This is the place where various system components are organised and where the interactions among
the components occur. To achieve one user requirement, for instance, to save input data from the user interface to the database, relevant classes will have to be designed and organised in a certain way to meet the requirement. When dealing with organisation and interactions among the components, design pattern plays very important role. Hence, if a system follows the dependency injection pattern, certain practice of component organisation and collaboration is applied.

It is generally accepted that any system adopting a specific design structure will have its associated complexity and maintainability. The idea is similar to the problem solving approach. To achieve one goal, there are many possible paths with their own cost. Thus, different system designs will have higher, lower or similar complexity and maintainability. In the case of software development, its complexity and maintainability can basically be measured by analysing the internal code structure.

Test suite is defined as the collection of test cases for the software system. Test suite is constructed to help reveal faults, assess system quality, and ensure the correctness of system functionalities. From the above conceptual model, the relationship of complexity and maintainability towards the test suite is that systems with different complexity and maintainability level require different number of test cases accordingly.

In a simple definition, testability refers to the ease with which a system can be tested. The relationship from the test suite to the testability is clearly witnessed as the test suite contains the number of test cases which determine whether the system is easy or difficult to be tested. Likewise, as complexity and maintainability affect the test suite, we assume that the two concepts also has connection to the testability. The discussion of the interrelationship among these concepts is reviewed in later chapters of the paper.
1.5 Paper Structure

To discover the answer to the research questions, the paper is structured into six chapters. Chapter 1 introduces the topic and relevant concepts. Research questions, overview of methodology, and conceptual framework are described as well.

Chapter 2 studies the concepts represented in the conceptual framework with background literature. The chapter restricts itself to discuss the concepts in abstraction. Details on implementation are given in the following chapter.

Chapter 3 continues the discussion from the literature review but to focus on details of implementation. The chapter is going to define metrics for the concepts in the framework from chapter 1.

Chapter 4 specifies the projects under experiment including their underlying components and how the components will be put together. The test suites for both projects are also defined. Then structure for comparisons of the projects and test suites are listed together with corresponding metrics defined in chapter 3.

Chapter 5 summarises the data collected from the projects and test suites. Explanations on various analyses are provided to justify the presented data.

Chapter 6 concludes the findings and tries to answer the research questions. Limitations and future work are discussed at the end of the chapter.
Chapter 2

Literature Review

This chapter will review the background literature on various components which share their connection within the study. We begin with software testing, test coverage and test adequacy. Then we will further discuss the concept of testability in relation with complexity and maintainability. Finally, dependency injection and mock are explained with code listing examples.

2.1 Software Testing

Software testing literature can be tracked back in the early 1970s as Hetzel & Hetzel (1988) organised the first conference on software testing topic. Since then software testing can be viewed as “destructive” rather than “constructive” (Hetzel & Hetzel, 1988; Myers et al., 2011). “Destructive” perspective of testing is that it aims to find error within the program, whereas “constructive” perspective aligns with developers’ urge to have a working and fault-free software. Then in the 1980s, software testing gained popularity which became part of the software engineering discipline, and the destructive view has shifted to include the idea of error prevention, maintainability and measuring software capability as well. Hetzel & Hetzel (1988) redefined the testing as the iterative process inside the software development life cycle.

In software testing literature, the two key terms: black-box testing and white-box testing are commonly mentioned. The two basic concepts are essential to understand other testing concepts in this literature review. Black-box testing treats the software under test as black box which we have no knowledge of the internal structure and algorithms of the software. In contrast, white-box testing assumes that we have knowledge of the internal structure and understand the detail process of the software. The white-box
testing is normally performed by the developers of the system, whereas the black-box testing can be exercised from the quality assurance team.

There are many different types of software testing practices. The testing methods to be applied on the software under test are determined based on the layers underneath software development. This notation of layers is best explained with the V-Model (Figure 2.1) of software testing paradigm (Ammann & Offutt, 2008).

![Figure 2.1: The V-Model (Ammann & Offutt, 2008, p. 8)](image)

We normally begin our test with unit test. Unit test by its nature assesses the smallest unit of the system under test. The unit in this sense refers to the methods of a class in object-oriented programming. For procedural programming, unit can be procedure or function. The aim of unit test is to test the unit in isolation. Therefore, we assume that unit testing will not involve other components. However, as already mentioned, in real application, connection to external resources cannot be avoided. Thus, to comply with unit testing principle, mock technique for unit testing allows us to test each unit in isolation. We will discuss further about mock in section 2.5.

The module test seems to blur with the unit test. Ammann & Offutt (2008) explained that with the rise of object-oriented programming, all units are wrapped inside a class (module for procedural programming). Testing all units is equivalent to testing the module as well. In addition, IEEE (1990) notes that the term module, component and unit are used interchangeably. However, MacConnell (2004) gave the distinction between the unit test and module test in terms of team collaboration. The unit test is performed by a single programmer who writes the unit. The module test is a class or package test
concerning many programmers or team. Thus, the module test will not involve with other classes or packages either. In this paper, we prefer to think of the module test and unit test as equivalent terms to avoid repeating the two terminologies together.

Unlike the unit test which tests each unit in isolation, integration test performs testing with the combination of different components together. For instance, if a component depends on another component, the two components in the integration test will be tested together to ensure the correctness of their behaviour. In the real application case, when a system connects to an external database, programmers have to connect to a real database instead of providing a mock object in replacement of the database.

From figure 2.1, the higher layer above the integration test is system test. In this case, we combine all components which build up the whole system into test targeting the system work flow. The system test ensures that the functionalities of the system meet the user requirements. The integration test and system test may take longer time than the unit test due to the connection to external resources. For this reason, the integration test and system test should be checked and executed less often than the unit test (Bender & McWherter, 2011).

Acceptance test involves itself with testing for user acceptance of the system. Normally, users of the real system are asked to perform the acceptance test to ensure the working order and the correctness of the system. It should be noted that the system test is performed by programmers to meet the user requirements, whereas the acceptance test indicates whether the end users satisfy with the system or not.

We consider the acceptance test as black-box testing because in this stage, the tests are not aware of any dependency in which the system has. Instead, the tests take into consideration the user requirements only. Conversely, the unit test, integration test, and system test are parts of the white-box testing. As developers perform the tests, they have the internal knowledge of the system under test.

In this project, we will perform our tests on three levels including the unit test, integration test, and system test on both systems since we will redevelop one system entirely and have the knowledge of the software internal structure. Additionally, we have full access to the source code of both programs allowing us to do comparison based on various software metrics available from the source code itself.
2.2 Test Coverage and Test Adequacy

In the software testing paradigm, developers need to consider test cases that they have to perform. The consideration on the number of test cases, i.e. test adequacy, of a certain software is determined by test criteria. To answer such question on the test criteria, Zhu et al. (1997) consolidated the literature on the topic for the last two decades. Here is a brief summary.

Zhu et al. (1997) explained four notions of the test adequacy criteria which still have been applied in modern testing techniques. These notations serve as the indicators of the adequacy of testing approach which developers or testers apply. The criteria include:

- **Statement Coverage**: testers aim to generate test cases to assert that every single statement of code is executed at least once during the test process. For this reason, code coverage can be used synonymously with statement coverage. Based on this criterion, testers would calculate the test adequacy relying on the percentage of the statements which the test cases cover.

- **Branch Coverage**: this adequacy criterion is calculated on the percentage coverage of the program execution paths. Control flow statement such as `if-then-else`, `switch case`, and `for` loop statement increases the execution branches. Testers would design test cases based on each branch of program conditions.

For example, given a piece of pseudo code to delete a collection by knowing its identity number within a database, there are two branches namely a case when the `collectionID` existed in the database and the other case when the `collectionID` does not exist. Listing 2.1 shows the pseudo code and its two executable branches.

```
public bool DeleteCollectionByID(int collectionID)
{
    if: collectionID does not exist, then return false.

    else:
        delete collection by collectionID.
        save changes to the database.
        return true.
}
```

**Listing 2.1**: DeleteCollectionByID with Two Branches

- **Path Coverage**: the criterion of test adequacy with path coverage is determined by the program flow from the entry point to the exit point. From the path coverage perspective,
a path is a segment of code including statements and control flows aiming to achieve program requirements. At the first glance, branch coverage and path coverage are similar concepts owing to the fact that control flow of the program determines the executable branches or paths. Listing 2.2 is the pseudo code to save text files from a given directory to a database.

```
public void SaveTextFiles(string directoryPath)
{
    if: directoryPath does not exists, then exit.
    else: fileNameList = retrieve all file names with extension .txt from directoryPath.

    foreach: fileName in fileNameList
    {
        if: fileName does not exist in the database, then
        read the content of fileName.
        save the content of fileName to database.
    }
}
```

**Listing 2.2:** SaveTextFiles with Six Basic Program Paths

Method `SaveTextFiles` consists of six basic paths determining the test cases: (1) when the `directoryPath` does not exist, (2) when there is no file with the extension `.txt` in `directoryPath` making `fileNameList` empty, (3) when there is one file and the file does not exist in the database, (4) when there is one txt file and the file already existed in the database, (5) when there is more than one file and at least one file does not exist in the database, and (6) when there is more than one file and all files already existed in the database. It should be noted that path (1), (2), (3), and (4) satisfy full statement and branch coverage already. Path (5) and (6) are additional test cases for path coverage criterion. We would conclude that the distinction between branch coverage and path coverage emerges with the existence of `loop` statement.

Applying path coverage encounters problems with infinite applicability, especially with the case of `loop` statement (Zhu et al., 1997). There can be infinite conditions to satisfy the `loop` condition; thus, we purposely defined only six basic paths of the `SaveTextFiles` because of such problem. Myers et al. (2011) have proposed a concept of multiple condition coverage stating that in the condition cases, there should be at least one test case satisfying the `true` evaluation and at least one satisfying the `false` evaluation. The six basic paths follow the multiple condition coverage to define finite applicability of the test criteria (Zhu et al., 1997).
Mutation Adequacy: from the destructive view of software testing (Hetzel & Hetzel, 1988; Myers et al., 2011), testing aims to detect faults inside the program. In mutation testing, artificial faults or mutants are injected into the system. Mutants can be created by altering the internal process or how the program does calculation. For example, in the original code which uses logic operator \texttt{and}, mutant can be created by changing logic operator \texttt{and} to operator \texttt{or}. Another similar approach is to swap the addition for subtraction and multiplication for division, and vice versa. The operators used in place of the original operators are called mutant operators. There are also mutant operators designed specifically for object-oriented programming. However, the discussion of mutation testing and mutant operators is beyond the scope of the paper.

In principle, we assume that the output from mutant program is equal to the one from the original program. If the output of the program after applying the mutant operators is different from the value of the original program, we conclude that the mutant is dead or killed by the test. Otherwise, the mutant is alive. Naturally, we aim to kill mutants as many as possible. Testers calculate the mutation adequacy, i.e. mutation score, by the ratio of the number of dead mutants over the subtraction between total number of mutants and total number of live mutants. The higher the mutation score is, the better the test adequacy is.

The notion of test adequacy serves at least two roles for software testing (Zhu et al., 1997). First, it helps determine the stopping rule for creating test cases. If testers follow statement coverage, test cases that exercise all statements inside a program are considered adequate. Second, it helps determine the test quality by quantifying into percentage that testers can assess whether the overall test is in an acceptable quality. For instance, if the test cases cover all branches in the program (100 per cent coverage), testers are confident about the completeness of the test.

One approach to categorise the test adequacy is based on information from the software requirements and the software internal structure itself. Zhu et al. (1997) categorised this perspective into three groups:

- Specification-based Criteria: as the name suggested, criteria are based on software specifications. The specifications will drive testers to design test cases to ensure that the requirements are met. In this notion, we consider that the specification-based criteria are similar to the concept of acceptance test.

- Program-based Criteria: following the white-box testing concept, program-based criteria will define test cases based on internal structure of the program. Likewise, testers who follow the program-based criteria would generate test cases by considering statement
coverage, branch coverage, and path coverage. It is advisable that both specification-based and program-based criteria should be taken into consideration. The specification-based criteria serve as correctness verification from the program-based criteria.

*Interface-based Criteria:* in some cases, testers can design test cases without basing on internal program information or specifications. The criteria are commonly applied to user interface data to verify whether the input values are correct or stay within the validation criteria which separate themselves from any structure or requirement of the software.

Finally, the test adequacy criteria can be categorised from the testing approach. Zhu et al. (1997) explained the groups as followings:

*Structural Testing:* this approach of testing aims to generate test cases from the information of the internal structure of the program. Again, the combination of the specification-based and program-based criteria is applied as discussed above. In addition to statement and branch coverage, data-flow criteria are applied in structural testing approach to determine the number of test cases as well. Data-flow criteria are used as complementary cases to branch coverage as data can change the control flow of the program internal logic.

An example of data-flow criteria can be witnessed from the discussion on the path coverage where we defined six basic paths of the `SaveTxtFiles` method (listing 2.2). The loop condition to traverse the list depends on the data gaining from `fileNameList`. The number of files existing in the list affects the flow path of the method.

*Fault-based Testing:* fault in this sense is the defect inside the software. Using this approach, testers would design test to decide the effectiveness of test cases of finding defects. Testers could possibly apply mutation adequacy as the criteria to design test cases. The mutation adequacy is discussed in the above section.

*Error-based Testing:* error in this case refers to the output defect of the program. Based on software specifications, the approach demands testers to test the program for vulnerable error points. To locate the vulnerable error points, the specification-based and program-based concepts can be utilised. This requires in depth understanding of the software process and internal structure. The main idea behind error-based testing is to provide input-output space which vulnerable error points could reside within the program under test. Two common approaches to define input-output space are random sampling and partition sampling.

To summarise, the test coverage and test adequacy criteria employ different perspectives that involve internal structure of the software, software specifications and input-output
space of the software. The criteria discussed in this section affect software testers in
designing the test cases for the system under test.

For the purpose of this study, we apply the structural testing approach by combining
the specification-based and program-based criteria. Specifically, statement and branch
coverage to satisfy full program code and flows will be our test adequacy criteria. As
discussed above, in the case of infinite applicability of control flow, multiple condition
coverage proposed by Myers et al. (2011) will be applied to form finite applicability of
the test suite. The test suites for both projects aim to reach as high adequacy as possible
in terms of statement and branch coverage.

2.3 Testability, Complexity, and Maintainability

IEEE (1990) defines testability as the ease of a system, component or requirement which
allows test criteria to be created and the ease of performing the test execution to verify
the satisfaction of the test criteria. From this definition, testability carries two parts.
The first part is about designing test criteria and the second part is about running the
test and test verification. Binder (1994) and Freedman (1991) suggested two facets of
testability: controllability and observability. The concept of controllability and observ-
ability are closely related with testability in the sense that the two concepts determine
the ease of creation, execution and verification.

Controllability is the degree to which software testers have control over the input value
of the system. Without full control of the input value, testers are not able to predict
the output value of the system. Low controllability can result in difficulty in setting up
test. Likewise, observability is the degree to which software testers are able to verify
the output of the program if any specific input value is supplied. Observability enables
testers to verify the correctness of the test creation and execution. In a short definition,
controllability refers to the input, whereas observability refers to the output.

Freedman (1991) attempted to measure testability by proposing metrics on controlla-
bility and observability. The approach from his study is to measure the input-output
space pairs (controllability and observability) of the program. In the case that the
input-output space is not controllable and observable, controllability extension and ob-
servability extension are applied. The two extensions are the procedures of modifying
the system components to become domain testable. Then the measurements of the two
facets are calculated from the input-output space partitioned within the whole domain
space. His argument was that a software with higher controllability and observability
should be easier to be tested.
Measuring the two metrics is a tedious work, particularly in the case that a method does not require an input and does not return an output. Another reason is that the procedure of converting such methods to domain testability is not applicable for our system projects due to the scope and time constraint. Additionally, the controllability and observability of the two systems are related to the external database. Since data inside the database are controllable and observable, comparing the two systems on such metrics does not give insight into the discrepancy between the two systems’ testability.

With a similar concept to Freedman (1991), Binder (1994) took into account various aspects of software including software specifications, source code implementation, test suite, and test tools. In this study, we will follow the concept from Binder (1994), but to focus on source code implementation only. The other aspects mentioned above are identical for both systems which we are going to examine.

Binder (1994) suggested different aspects of source code which contribute to software testability. They are complexity, encapsulation, inheritance, and polymorphism. Metrics responding with the four aspects were also proposed by Binder (1994). Other authors followed similar approach and explained that system complexity increases the testing efforts due to class collaborations. McCabe & Butler (1989) used the cyclomatic complexity of the system design to integrate with testing methodology. Cyclomatic complexity applies the concept of branch and path coverage of the test adequacy criteria to generate test cases. Thus, if the cyclomatic complexity increases, the number of test cases and testing effort increase.

Another concept related to complexity and testability should be discussed as well. Maintainability is defined as the ease of modifying software components to include or remove capability, to tweak performance, and to fix defect (IEEE, 1990; MacConnell, 2004). From this definition, maintainability is closely related to complexity. Naturally, a system with less complexity should be easier to maintain. Besides, both complexity and maintainability share many overlapping measurement metrics such as coupling, cohesion, and software size metrics.

The ISO 9126 (1991) has defined testability as one of the features under maintainability. Thus, studying testability is more or less related to maintainability or vice versa. In addition, Riaz et al. (2009) reviewed fifteen studies on software maintainability and asserted that maintainability is closely related to software complexity, and the prediction approach used by the reviewed studies is calculated with the input from complexity metrics. Riaz et al. (2009) further suggested that sub-characteristics of maintainability as defined by the ISO 9126 (1991), i.e. testability, can be predicted using similar metrics from complexity and maintainability. The study on measuring maintainability by Heitlager et al. (2007) also followed the model of the ISO 9126 (1991). Instead of studying
the maintainability directly, the authors decided to analyse various system components on the source code level by mapping them to the maintainability sub-characteristics. In order to measure system testability, the authors considered three factors: system complexity, unit test coverage, and unit test size.

In a study to predict the class testability in object-oriented system, Bruntink & van Deursen (2006) argued that the source code has effects on two factors of system testing. The two factors are test case generation factor and test case construction factor. Test case generation factor shares similarity to the concept of test adequacy criteria determining the number of test cases which have to be performed. Test case constructor factor is related to how to implement each test case. For example, if a class under test requires necessary data to work with, the input data must be initialised. Moreover, if the class has dependencies on other classes, such dependencies are also instantiated for the class under test to execute. In other words, test case constructor factor can be considered as the test effort which testers invest to create the test suite. From the two factors, complexity and maintainability related metrics of the source code are defined in their study as well.

From the above arguments, testability of a system can be measured based on the two facets: controllability and observability. Alternatively, complexity and maintainability from the source code can also be implemented to imply the testability due to the influences from the source code on software testing particularly test suite. If we take into account the definition of testability by the IEEE (1990), test suite and testability are closely related in terms of creating test cases, applying test criteria (statement and branch coverage) and executing the test cases to verify the result.

Because the study involves re-engineering a new system from another one, we have full access to the source code. Thus, the comparison of the two systems’ testability will be based on the consideration from complexity and maintainability in relation to the test effort to construct the test suite. Metrics related to complexity, maintainability, and test suite are further discussed in chapter 3.

2.4 Dependency Injection

Basically, dependency injection is a design practice by supplying services to the one who needs them. In technical concept, it is the practice of supplying service classes to another class which depends on them. The pattern has its own specific practices on how to inject the services, and it is best explained with coding examples in the next sections.
This design pattern is claimed to make code loose coupling; hence, the code is also extensible, maintainable, and testable (Johnson & Foote, 1988; Martin, 1996; Prasanna, 2009; Seemann, 2012). The dependency injection pattern bases mainly on the design principle “program to an interface, not an implementation” (Freeman, 2004). As Seemann (2012) pointed out this principle is not the result but the “premise” for the dependency injection concept. For this reason, the design using the dependency injection will normally utilise the abstraction (abstract class or interface).

There are many key terms which are in association with the dependency injection pattern in the literature. The term Dependency Inversion Principle was first appeared in Martin (1996) which shared the same principle as the dependency injection. The prominent work from Fowler (2004; 2005) suggested a different term as Inversion of Control, which he was inspired by the article from Johnson & Foote (1988). Fowler (2004) coined the term “dependency injection” to refer to the pattern instead of using the generic and often confusing term “Inversion of Control.”

The dependency injection pattern comes into at least two different forms: constructor injection and property injection (Fowler, 2004; Prasanna, 2009; Seemann, 2012). Fowler (2004) and Prasanna (2009) categorised another form as interface injection, while Seemann (2012) used method injection. The method injection and interface injection share similarity basing on the use of method to perform the injection. All the forms we are about to discuss are implemented in the new system. Understanding the patterns helps reviewers to validate how the new system is redeveloped and to assess the extent in which the dependency injection is applied.

2.4.1 Constructor Injection

Constructor injection is the simplest form of the dependency injection pattern. When a class requires a service or functionalities from another class, we simply supply the dependency on the constructor of the class which needs it. Listing 2.3 demonstrates the constructor injection.

```
public class CollectionRepository : ICollectionRepository
{
    private readonly ICodiscloseContext context;
    public CollectionRepository(ICodiscloseContext context)
    {
        this.context = context;
    }
    IQueryable<PdfCollection> GetCollections() {...}
```
Chapter 2. Literature Review

Listing 2.3: Constructor Injection on CollectionRepository Class

CollectionRepository class depends on ICodiscloseContext, which provides necessary services namely querying data, adding, deleting, updating and saving changes entity to the database. The ICodiscloseContext is injected via the constructor of CollectionRepository class. As already described that the dependency injection pattern complies with the principle “program to an interface, not an implementation”, we let CollectionRepository depend on an interface instead of real implementation. The interface in this sense can refer to abstract class or interface type in C#.

The problem with the constructor injection appears when any class has many dependencies. Injecting multiple dependencies in the constructor is the code smell and should be avoided (MacConnell, 2004; Seemann, 2012). Therefore, three or four dependencies should be the acceptable number. In case there are more than four dependencies, property injection can be the trade-off solution.

2.4.2 Property Injection

Property injection is very similar to the constructor injection form except that the injection happens via the setter property. The property injection assumes that the class that needs services already has “local default” implementation (Seemann, 2012). Thus, the dependency is optional while allowing users the ability to inject another different dependency. For example, the CollectionRepository class has a default data source from an XML file. Listing 2.4 demonstrates the concept of the property injection. The code is simplified without any exception checking for the sake of understanding only.

```csharp
1 public class CollectionRepository : ICollectionRepository {
2     private ICodiscloseContext context;
```
Chapter 2. Literature Review

```csharp
public CollectionRepository()
{
    this.context = new XMLContext();
}

public ICodiscloseContext Context { get; set; }

IQueryable<PdfCollection> GetCollections() {...}
PdfCollection GetCollectionByID(int collectionID) {...}

public class XMLContext : ICodiscloseContext {
    IQueryable<T> GetEntities<T>() {...}
    void Add<T>(T entity) {...}
    void Delete<T>(T entity) {...}
    void Update<T>(T entity) {...}
    void SaveChanges() {...}
}
```

Listing 2.4: Property Injection on CollectionRepository with Default XMLContext

Context property of CollectionRepository allows users to inject a different dependency which implements ICodiscloseContext. In the constructor of CollectionRepository, XMLContext is assumed to be the default data source.

However, the property injection has weaknesses which are worth discussing. One of them is that users who are not familiar with the internal structure of CollectionRepository may not know that CollectionRepository depends on ICodiscloseContext. There is no hint to the users to be aware of such dependency from the constructor of CollectionRepository. Another problem is that CollectionRepository allows a different dependency to be injected any time via the setter property. Thus, the users may inject a different dependency multiple times somewhere inside the program. From the debugging perspective, it is hard to track the behaviour of the program because we may not know which dependency is being used during the execution.

The local default is not necessarily needed, but it makes sense in the above situation. In case the users do not supply the dependency or do not know about the internal structure, CollectionRepository still works, at least with XMLContext. Because of such issue with the property injection, Beck (1997) recommended the constructor injection because programmers can see the dependencies during instantiation, and the object could not be instantiated without passing dependencies either.
2.4.3 Method Injection and Interface Injection

The two injection forms perform the dependency injection by passing dependency as the argument to a method. The main usage of the two forms is to provide dynamic dependency which can be obtained during the execution time. This happens when the dependency requires necessary information from the user interaction; for instance, select an item from a list of available actions. The good feature of the interface injection is that it enforces the class implementing its interface to supply the dependency. Listing 2.5 demonstrates the method injection and interface injection.

```java
public interface IRepositoryImportService
{
    void BuildIndex(int repositoryID, ISvnService svnService);
}

public class SvnImporter : IRepositoryImportService
{
    private readonly IRepositoryService repositoryService;
    private ISvnService svnService;

    public SvnImporter(IRepositoryService repositoryService)
    {
        this.repositoryService = repositoryService;
    }

    public void SetSvnService(ISvnService svnService)
    {
        this.svnService = svnService;
    }

    public void BuildIndex(int repositoryID, ISvnService svnService)
    {
        SetSvnService(svnService);
        ...
    }
}

public class SvnService : ISvnService
{
    public SvnService(string repositoryUri,
```
In listing 2.5, **IRepositoryImportService** sets the requirement on its method `BuildIndex` to supply **ISvnService**. **SvnImporter** which implements **IRepositoryImportService** depends on two services: **IRepositoryService** and **ISvnService**. One of the dependencies is injected via the constructor, while the other is injected via `BuildIndex` method. In this case, **ISvnService** is a dynamic dependency which is obtained during the execution of the program when end-users select an online code repository to get the required information, namely repository URI address, user name and password, which **SvnService** needs to operate with.

### 2.4.4 Dependency Injection Container

Looking at the dependency injection patterns above, you may have noticed that there is no concrete instance having been used. All the patterns involve the use of the **interface** or **abstract class**. Both abstract types cannot be instantiated by themselves; therefore, once the dependencies are in need to be used, concrete instantiations equivalent to the abstract types are required to pass in. The dependency injection container serves this purpose.

The dependency injection container has the ability to resolve the instantiation between concrete types and their abstract types. To be able to resolve this work, rules determining how the resolve should work are stored in the container. The algorithm for the containers to work is a three-step process (Seemann, 2012). First, the rules must be registered within the container. Second, by using the defined rules, the container resolves between abstract types and concrete types. Then the concrete types are implemented within the application. Third, the container releases the concrete types when they are no longer in use, i.e. disposing object and saving resources.

In the following example, **Ninject**, an open-source dependency injection container framework, is used to demonstrate how the rules are defined and how to resolve dependency between abstract types and concrete types. There are many other open-source containers available to choose, and the reason we choose Ninject is based entirely on previous knowledge and familiarity with the framework only.

In Ninject environment, the place where rules are registered is called **NinjectModule**. **NinjectModule** is an abstract class; thus, we need to inherit this class and override...
one method where we define the dependency rules. From listing 2.5, SvnImporter class implements an interface IRepositoryImportService and depends on two services: IRepositoryService and ISvnService. Listing 2.6 demonstrates how dependency rules are defined.

```csharp
public class CoreModule : NinjectModule
{
    public override void Load()
    {
        Bind<IRepositoryService>().To<RepositoryService>();
        Bind<IRepositoryImportService>().To<SvnImporter>();
    }
}
```

Listing 2.6: Register Dependency Rules with Ninject

The rules from listing 2.6 can be interpreted as: when an application needs to use IRepositoryService, it is resolved with RepositoryService. The same interpretation is applied to IRepositoryImportService and SvnImporter. There is no rule for ISvnService since it is dynamically passed during the program execution. From this notation, we would understand that registered rules in the container apply only to static resolve.

With the above defined rules, the container is able to resolve between abstract types and their equivalent concrete instances. Listing 2.7 demonstrates how rules can be resolved with Ninject framework.

```csharp
static class Program
{
    [STAThread]
    public static void Main()
    {
        var kernel = new StandardKernel(new CoreModule());
        var svnImporter = kernel.Get<IRepositoryImportService>();
        // svnImporter is ready for use.
        ...
    }
}
```

Listing 2.7: Resolve Dependency Rules with Ninject
Ninject provides a kernel whose role is to resolve the registered rules defined in the
\texttt{NinjectModule}. \texttt{StandardKernel} needs the rules to resolve, so \texttt{CoreModule} is passed
to its constructor. Then the kernel can give concrete instances accordingly. It should be
noted that the container is able to understand that \texttt{IRepositoryImportService} should
be resolved with \texttt{SvnImporter}, while \texttt{SvnImporter} depends on \texttt{IRepositoryService}.
The container will resolve \texttt{IRepositoryService} with \texttt{RepositoryService} before the
concrete type of \texttt{SvnImporter} is instantiated.

The dependency injection container is optional if we would like to do all the manual work
of registering, resolving, and releasing. The above two code listings can be done with
simple instantiation of objects by passing the concrete implementation to the correct
abstract types. However, the manual work becomes challenging when dealing with many
dependencies and in the case of requirement changes, more work needs to be done for
the maintenance of the system. Thus, the dependency injection container is normally
the preferred option. Listing \ref{lst:manual} demonstrates the manual registering and resolving
dependencies supposing that \texttt{RepositoryService} does not have any other dependency
and has the default constructor for object instantiation.

\begin{verbatim}
1 static class Program
2 {
 3   [STAThread]
4   public static void Main()
5   {
6     IRepositoryService service = new RepositoryService();
7     IRepositoryImportService importService =
8           new SvnImporter(service);
9
10     //importService is ready for use
11     ...
12   }
13 }
\end{verbatim}

\textbf{Listing 2.8:} Manual Registering and Resolving Dependencies

\subsection*{2.4.5 Benefits of Dependency Injection}

Proponents of the dependency injection pattern have argued over its effect on various
software quality attributes such as maintainability and testability. Prasanna (2009) and
Seemann (2012) explained that the dependency injection enables loose coupling code
and design. The reason is that the design bases on abstraction rather than concrete implementation. As seen in figure 2.2, CollectionRepository depends on the interface ICodiscloseContext. Thus, any implementation of ICodiscloseContext can be injected into CollectionRepository. From this design, CollectionRepository does not bind with any implementation, which makes it loose coupling.

Many authors has suggested coupling metrics as the prediction of complexity and maintainability (Briand et al., 1999; Dagpinar & Jahnke, 2003; Riaz et al., 2009). Since the dependency injection has the effect on class coupling, our study aims to measure the metrics between the two systems to witness to what extent the dependency injection enables loose coupling code.

Because of the loose coupling code, the design by the dependency injection improves testability (Prasanna, 2009; Seemann, 2012). This effect influences the controllability and observability for the test case. Again, from such design in figure 2.2, testers have choices to perform unit test on CollectionRepository. One option is to instantiate real implementation which is used in the main program. Another option is to implement the interface ICodiscloseContext as stand-in resource for unit test purpose. Final option is to use mocking framework to mock ICodiscloseContext without implementing the interface as in the second option. Since such flexibility gives testers control over the test options, statement and branch coverage should be improved. Mock will be discussed in section 2.5 below.

Finally, the dependency injection also increases extensibility and reusability. Loose coupling design makes the CollectionRepository reusable. In the case when there is a change to software specification affecting ICodiscloseContext, CollectionRepository will not be affected. Any new implementation will work with it as long as the implementation follows the contract imposed by the interface. For this reason, long-term maintainability is easier for the whole system as well. Another possible extensibility is
that the dependency injection enables other design patterns to be integrated with itself easily. One of them is the Decorator pattern. For further discussion on this topic, please consult the book on the dependency injection by Seemann (2012).

2.5 Mock

When applying unit testing, we would like to test the smallest unit in isolation. This implies that the unit test should test the methods within one class without talking to another class. The scenario is not practical in real world programming and system. The class and unit which programmers want to test will normally communicate with another component or service. This breaks the original goal of the unit test. Because of the challenge, mock plays an important role in separating the external component or service from the unit test.

What mock helps programmers is that instead of using an actual component or service, programmers can supply a mock object as a fake component or service to ensure the working of the test. By mocking the component or service, programmers decouple the dependency and test the unit in question in isolation. With such practice, test suite should be easier to be constructed and executed, which the study aims to explore.

The term mock is used in a general meaning which refers to a family of similar implementations to replace real external resources during unit testing (Bender & McWherter, 2011). Other similar terms are dummy, stub, fake, and mock (Bender & McWherter, 2011; Fowler, 2007; Seemann, 2007). While any attempt to distinguish these terminological words does not help but causes confusion, we group the above terms into two groups: stub and mock. Stub group includes dummy, fake and stub. The second group is mock itself. The stub group acts as stand-in resource instead of real resource. It provides only necessary data for the unit test. Fowler (2007) called this kind of test as state verification. The mock includes the same implementation as stub plus the behaviour verification (Fowler, 2007). Bender & McWherter (2011) explained that mock has the ability to verify which method, how frequent, and which order the method is called. Based on such information, mock can react accordingly.

2.5.1 Mock Pattern

Mackinnon et al. (2001) has concluded the mock pattern by the following five-step procedure. The procedure includes:

1. Create mock object
2. Set up mock object state

3. Set up mock object expectation

4. Supply mock object to domain code

5. Verify behaviour on mock object

From the above differentiation between stub and mock, step one to four are common procedures for stub, while step one to five represent mock by including behaviour verification. In the following examples, we are going to demonstrate the stub and mock procedures by using Moq, a mocking library for the .Net Framework. The unit test implementation is conducted using Microsoft unit testing framework. Further explanation will be provided along with code listings.

Listing 2.9 illustrates the internal structure of CollectionRepository class. Two methods: GetCollections and DeleteCollectionByID will be unit tested as shown in listing 2.10 and listing 2.11 respectively.

```csharp
public class CollectionRepository : ICollectionRepository
{
    private readonly ICodiscloseContext context;
    public CollectionRepository(ICodiscloseContext context)
    {
        this.context = context;
    }

    public IQueryable<PdfCollection> GetCollections()
    {
        return context.GetEntities<PdfCollection>();
    }

    public bool DeleteCollectionByID(int collectionID)
    {
        var collection = this.GetCollections().FirstOrDefault
        (c => c.id == collectionID);
        if (collection == null) return false;
        context.Delete<PdfCollection>(collection);
        context.SaveChanges();
        return true;
    }
}
```
Listing 2.9: `CollectionRepository` Class With Its Two Methods

```csharp
[TestMethod]
public void GetCollections_Test()
{
    var context = new Mock<ICodiscloseContext>();
    var repository = new CollectionRepository(context.Object);

    var list = new List<PdfCollection>();
    list.Add(new PdfCollection() { id = 1, name = "test1" });
    list.Add(new PdfCollection() { id = 2, name = "test2" });
    list.Add(new PdfCollection() { id = 3, name = "test3" });
    var mockList = list.AsQueryable<PdfCollection>();
    context.Setup(c => c.GetEntities<PdfCollection>())
        .Returns(mockList);

    Assert.AreEqual(3, repository.GetCollections().Count());
}
```

Listing 2.10: Unit Test on GetCollections with Stub

In listing 2.9, `CollectionRepository` depends on `ICodiscloseContext`, which provides service for retrieving data from the database, deleting records and saving changes to the database. We perform stub procedure to test `GetCollections` method in listing 2.10. First, we setup mock object on `ICodiscloseContext` and inject it to `CollectionRepository` (line 4-5). We prepare mock data by adding items manually. Then we instruct mock `context` that whenever it calls method `GetEntities<PdfCollection>`, return `mockList` instead (line 12-13). Finally, we assert if `GetCollections` method returns a list of collection with three items as we have setup with `mockList`.

```csharp
[TestMethod]
public void DeleteCollectionByID_Test()
{
    var context = new Mock<ICodiscloseContext>();
    var repository = new CollectionRepository(context.Object);

```
In listing 2.11, we perform the similar procedure as in listing 2.10. From listing 2.9, `DeleteCollectionByID` method calls two methods: `Delete<PdfCollection>` and `SaveChanges` of `context`. Thus, we further setup to verify on call behaviour of the two methods on mock `context` (line 16-17). After executing line 19, we verify that both methods: `Delete<PdfCollection>` and `SaveChanges` are called once only. Finally, we assert whether delete collection method returns value `true`. It is important to remember that the purpose of mock is to verify the behaviour of the method under test. It does not perform real data deletion during the test execution.

### 2.5.2 Benefits of Mock

Besides acting as stand-in resource of real object, mock provides us with many other benefits as well. Bender & McWherter (2011) and Mackinnon et al. (2001) gave several reasons to use mock as followings:

In general terms, mock enables fast execution on unit level because of the decoupling with external resources. Mock also enables testers to reproduce error during the test easily. For instance, network error would happen when there is a problem with hardware and software configuration. Producing such error in unit test may cost more time and effort.
Developers normally work as a team to create a system. In the case that many developers work on the same database involving changing state of the data, unit tests from one developer may fail because the other developers could have altered the data. Mock ensures that unit test is localised because the external resources are replaced with in-memory data required for the test. Then controllability and observability are ensured.

From developers’ perspective, they should be able to control and predict the result of the unit test. When they supply input by controlling the variation value, the expected result should be asserted accordingly. However, with the involvement of external resources, controlling and predicting result from the unit test can be a challenge because the chances of data changes are high and unexpected. Mock provides developers with controllability and predictability of the input data and the expected results in a consistent way.
Chapter 3

Metrics

This chapter aims to define relevant notations and metrics with operational definition. The notations are mainly about object-oriented features such as class, field, and method. The metrics in our study will be the combination of complexity and maintainability metrics with their preliminary concepts discussed in chapter 2. Additionally, metrics related to test suite are also defined in the chapter.

3.1 Object-Oriented Notations

To have unambiguous definition, we would like to borrow notations from Briand et al. (1999). The simplified version of the notation can be found in Bruntink & van Deursen (2006). We define the definition in relation to the features of C# programming language.

*System*: an object-oriented system is the collection of types. There is at least one type within the system.

*Namespace*: a collection of types within the system. Namespace can be considered as a division inside the system.

*Type*: a type can be a class, an abstract class, an interface, an enumeration value and *struct*. This definition is mainly applied to C# programming language feature.

*Class*: class is the collection of methods, fields, and properties. At least one of the three components of class must be defined within class body. A class denoted by $c$ can have relationship as below:

- *Children($c$)* are the classes that inherit directly from $c$. 
• *Parent*(c) is the class that c inherits directly from.

• *Descendants*(c) are classes that inherit directly and indirectly from *Children*(c).

• *Ancestors*(c) are classes from which *Parent*(c) inherits directly and indirectly.

**Method**: methods are parts of a class. Methods include:

• inherited methods from *Ancestors*(c).

• newly declared methods within a class body including abstract methods, overridden methods, and methods implemented from an interface. Class constructor(s) and property *get* and *set* belong to this category. Non-overridden methods are excluded and are part of the inherited methods.

• methods declared within third party assemblies are not taken into account.

**Method Invocation**: method m invokes another method n if and only if within m body, method n is called.

**Field**: fields are declared within a class body but outside any method body. Fields can be declared as readonly, constant, enumeration value, primitive type, and any other type. Fields defined in third party assemblies are excluded.

**Field Reference**: field reference happens when in the body of method m, there is a reference to that field.

### 3.2 Size-Related Metrics

Metrics to measure system complexity can be found from the system size. The idea is that the larger the system is, the more complex it is to maintain, test and debug. Chidamber & Kemerer (1994) proposed the metrics as followings:

**Number of Lines of Code** (NbLOC): it counts the number of lines of code. Variations exist for the metrics as well. It can be code statements without comments or code statements and comments together. In our study, we count the lines of code of the intermediate language (IL). Thus, the code with or without comments are measured fairly. Another important point is that the tool used to calculate the NbLOC excludes the generated code by the Visual Studio.

\[
\text{NbLOC} = \text{Number of Lines of IL Code}
\]
**Number of Types** (NbTypes): it counts the number of types at the namespace level and system level. Types follow the definition from the section 3.1.

**Number of Methods** (NbMethods): it counts the number of methods within a class. Methods of a class follow the definition from the section 3.1. Methods which are declared in the abstract class or interface type are excluded.

**Number of Fields** (NbFields): it counts the number of fields within a class. Fields of a class follow the definition from the section 3.1.

### 3.3 Cyclomatic Complexity

**Weighted Methods Per Class** (WMC) aiming to measure class complexity is first introduced by Chidamber & Kemerer (1994). WMC is the total of each method complexity within a class. In their paper, Chidamber & Kemerer (1994) gave value one to every method. In other words, regardless of method implementation and internal structure, each method has the same complexity. Therefore, WMC is equal to the number of methods within the class.

**Cyclomatic Complexity** (CC) measures the complexity of a method as well. However, McCabe & Butler (1989) considered the complexity based on the control flow statements within the method implementation. Control flow statements increase the number of paths that a program can execute. The higher the number of paths is, the higher the complexity is. Control flow statements in C# include if, while, for, foreach, case, default, catch, and logical operators (and and or). Cyclomatic complexity of a method will be the value one plus the number of encounters with the above control flow statements.

In this study, we would like to adopt the CC concept on the method complexity. The CC on the class level will be the sum of each method CC, and the CC on the namespace level is the total number of each class CC residing within the namespace. System or application level follows the same principle.

### 3.4 Maintainability Index

To measure or predict the maintainability of a software system, different numbers of metrics have been applied in the literature. The studies by Coleman et al. (1994) and Welker et al. (1997) indicated that certain numbers of metrics are better predictive to the system maintainability. The predictive metrics include Halstead’s volume, cyclomatic
complexity, number of lines of code, and number of lines of comment. They applied
the set of selected metrics for maintainability on eight programs obtained from Hewlett-
Packard. The programs’ size ranges from 1,000 to 10,000 lines of code. From the above
predictive metrics, maintainability index (MI) is evolved. There are different versions in
calculating MI introduced by Coleman et al. (1994) and Welker et al. (1997) which will
be illustrated as below:

\[
MI_1 = 171 - 5.2 \times \ln(Vol) - 0.23 \times CC - 16.2 \times \ln(LOC) \\
+ 50 \times \sin(\sqrt{2.4 \times NbLinesOfComment})
\]

\(Vol\) is the Halstead’s volume; \(CC\) is the cyclomatic complexity value; \(LOC\) is the num-
ber of lines of code; and \(NbLinesOfComment\) is the number of lines of comment. The
Halstead’s volume is calculated with the involvement of operators plus the number of
operands available from the source code (details on how to calculate Halstead’s volume
can be found in Kan (2002)). From the formula, \(NbLinesOfComment\) can increase the
value of MI up to 50 units. In the case of system which does not have considerable
comments, the above formula can be simplified to omit the involvement of \(NbLinesOf-
Comment\) as following:

\[
MI_2 = 171 - 5.2 \times \ln(Vol) - 0.23 \times CC - 16.2 \times \ln(LOC)
\]

For our study, the tool used to calculate the MI value follows similar formula from MI2
with minor alterations to limit the range value between 0 and 100 (Conor, 2007):

\[
MI_3 = \text{Max}(0, (171 - 5.2 \times \ln(Vol)) - 0.23 \times CC \\
- 16.2 \times \ln(NbLOC)) \times 100/171)
\]

The \(NbLOC\) is the number of lines of IL code. Thus, it is reasonable to exclude the
number of lines of comment from the formula since IL code does not include comment
lines. It should be noted that \(MI_1\) value can range over 171 due to the additional 50
value from the number of lines of comment, while \(MI_2\) value peaks at 171. On the other
hand, \(MI_3\) value will fall in between 0 and 100. The higher MI value indicates that the
system is easier to maintain for future changes.
Chapter 3. *Metrics*

Reviewing the formulas to calculate MI explains the connection between the two concepts of complexity and maintainability as we have discussed in chapter 2. Specifically, the metrics for complexity including cyclomatic complexity and number of lines of code are the cores of the MI formula. MI metrics is considered to be a good indicator for our study as well since the dependency injection pattern is claimed to affect system long-term maintainability. We follow the formula defined in $MI_3$. At the class level, the MI will be the average of the methods’ MI value; likewise, the namespace level MI is the average of its classes’ MI value.

### 3.5 Coupling Metrics

Coupling aims to measure the strength of dependency between methods and fields of different classes within a system (MacConnell, 2004). Low coupling between classes is desirable, and it enables reusability. Since low coupling indicates low association between classes, it improves maintainability, and changes will be easier to make. High coupling is also a sign of fault-proneness, and testing such components needs to be rigorous which in turn increases test cost (Basili *et al.*, 1996; MacConnell, 2004).

A consolidation study on coupling measurement by Briand *et al.* (1999) discussed various aspects of coupling. One of the aspects include the direction of coupling with import and export service between classes. A client class depends on a server class which provides necessary services for the client class to function. This aspect can be witnessed in the metrics *Coupling Between Objects* (CBO) proposed by Chidamber & Kemerer (1991) as the metrics counts the number of associated classes with both directions of import and export service.

Similarly, Martin & Martin (2006) defined two metrics based on such direction of coupling. The two metrics are *Afferent Coupling* (Ca) and *Efferent Coupling* (Ce). Martin & Martin (2006) measure Ca of a method of a class as the number of methods of other classes which invokes it directly. Ce of a method of a class is the number of methods of other classes which it invokes directly.

From the above definition of Ca and Ce, coupling happens when there is method invocation between classes. However, coupling can happen in various situations such as field reference, parameters, return values from a method, and class instantiation (MacConnell, 2004).

In relation to software testing, Ce coupling is taken into account because test case of such method needs to be constructed by considering the involvement of other classes. Associated classes have to be initialised before the tested method can be executed. For
this reason, we would like to define the coupling metrics for the study by following the
definition of coupling from MacConnell (2004) and considering only one direction as
in Ce. We also distinguish between unique coupling and repeating coupling between
classes. The unique coupling counts the associated classes in distinction. Even though
many methods of a class depend on several other methods and fields from another
class, the unique coupling will be counted as one. Unique coupling also applies to the
namespace and application level. In contrast, the repeating coupling does not consider
unique count. Coupling from abstract methods and interface types is ignored in the
calculation.

\[
\text{Repeating Coupling (Coupling)} = \text{Count of coupling classes}
\]

\[
\text{Unique Coupling (UCoupling)} = \text{Distinct count of coupling classes}
\]

In a study by Bruntink & van Deursen (2006), the authors used Fan Out (FOUT) metrics
to measure coupling and testability. FOUT is one direction of CBO which counts the
coupling of a class that uses the other classes. FOUT is similar to the above coupling
definition with the exception that FOUT considers class coupling on method invocation
and field reference only. Another common metrics for coupling is Response for Class
(RFC). RFC of a class is the count of methods within the class plus the number of
methods from other classes that are invoked within that class body. RFC and coupling
share similarity in a sense that RFC counts the coupling on the method, while coupling
counts the dependency on the class level. With the testing perspective, instantiating a
class in the test case is required before using its methods. For this reason, we believe
RFC is a repetition of class coupling, which we will exclude from the study.

### 3.6 Cohesion Metrics

Cohesion is the degree of methods within a class performing related tasks with a shared
common purpose (MacConnell, 2004). The idea to increase cohesion of a class is to
design methods to access fields declared within such class. The assumption is that
methods which work with the same class data share common goal and function. In a
non-cohesive class, each method works with different data on different purposes.

In contrast to coupling, high cohesion is desirable due to the same effect on maintainabil-
ity, change, and complexity as in the case of low coupling. High cohesive class indicates
close relation between methods and field data. For low cohesive class, unrelated methods
should be placed in different classes to increase cohesion.
Different authors have proposed similar measurements for cohesion. One of the most common used metrics is *Lack of Cohesion of Methods* (LCOM). LCOM is the inversion of cohesion; thus, low LCOM value is preferable indicating high cohesion. There are different definitions and how to calculate LCOM proposed in the literature. Chidamber & Kemerer (1991; 1994) defined LCOM with the following formula (we would like to call it LCOM1):

\[
LCOM_1 = \begin{cases} 
P - Q & \text{if } P > Q, \\
0 & \text{otherwise.}
\end{cases}
\]

The LCOM1 takes each pair of methods in a class into consideration. If a pair of methods does not share at least one common field data, increase \( P \) by one. Conversely, if a pair of methods shares at least one common field data, increase \( Q \) by one. In the case, \( P \) is less than \( Q \), the LCOM1 value is set to zero, which is the maximum cohesion. Since LCOM1 assesses \( P \) and \( Q \) value based on field data, classes with only property access `get` and `set` will receive high value of LCOM1 (Gupta, 1997). Thus, LCOM1 is not a good indicator for such classes. Additionally, Chidamber & Kemerer (1991; 1994) did not state whether inherited methods and fields are counted in the formula (Briand et al., 1998).

Two similar metrics with the same name were proposed later in the literature. The two metrics aim to tackle the drawbacks we have mentioned in LCOM1. We would like to called them LCOM2 and LCOM3 respectively. LCOM2 is calculated from the following formula:

\[
LCOM_2 = \begin{cases} 
0 & \text{if } M = 0 \text{ or } F = 0, \\
1 - \frac{\text{Sum}(MF)}{M \ast F} & \text{otherwise.}
\end{cases}
\]

Where \( M \) is the number of methods of a class; \( F \) is the number of fields of a class; \( MF \) is the number of methods accessing a field data; and \( \text{Sum}(MF) \) is the total of \( MF \) for all fields in a class. The value ranges from zero to one. Value zero denotes the maximum cohesion, and value one denotes the minimum cohesion.

LCOM3 is proposed by Sellers (1996) and yields value between zero and two. If the value is greater than one, it represents a serious lack of cohesion of a particular class. Therefore, value below one falls in an acceptable range. The following formula is to calculate LCOM3:
Chapter 3. Metrics

\[ LCOM3 = \begin{cases} 
0 & \text{if } M = 1 \text{ or } F = 0, \\
(M - \text{Sum}(MF)/F)/(M - 1) & \text{otherwise.}
\end{cases} \]

The zero case of LCOM2 and LCOM3 poses problem in calculating the formula. Actually, it is an abnormal case for a class which does not have any method and field (LCOM2 = 0). However, there is still a scenario when such case happens. For example, a class has only public access fields without any property \texttt{get} and \texttt{set}, yet the LCOM2 is set to zero meaning perfect cohesive class. Likewise, in the case of LCOM3, a class with only static methods and without any field data is problematic in the analysis as LCOM3 will be zero.

Another problem with LCOM2 and LCOM3 as indicated by Briand et al. (1998) is the access method (property accessors) and constructor of the class. These two types of method can increase and decrease LCOM value. For instance, the access method normally has access to only one field data. While the LCOM value is calculated based on the number of methods accessing all fields in the class, LCOM value for such kind of class will increase presenting non-cohesive class. Similarly, the constructor normally initialises every field of a class. Consequently, the constructor can lower the LCOM value substantially. However, the above arguments have been alleviated with the presence of automated tools such as NDepend. Classes with property method \texttt{get} and \texttt{set} and/or without constructor have the LCOM value zero.

There are other cohesion metrics available from the literature. However, due to the lack of automated and reliable tools in calculating such metrics, we will not include those metrics in the discussion and study. Instead, we would like to implement LCOM2 and LCOM3 in this paper.

3.7 Test Suite Metrics

As we have discussed in section 2.2 on test adequacy criteria, code and branch coverage can be served as the stopping rules when defining the number of test cases for a specific system. In addition to the adequacy criteria, code and branch coverage have been widely applied to predict quality and failure of the system. The idea is that the code section covered by the test cases is executed at least once. Thus, the execution of the code section is able to detect fault or error leading to better quality.

However, some authors doubt the above claim. Nagappan et al. (2010) explained that the code covered by the test cases does not guarantee flawless code because the main
Purpose of code coverage is to measure the percentage of code exercised by the test case. Any test case without flaw detection purpose can achieve similar coverage. Additionally, Nagappan et al. (2010) continued to argue that the input-output value range in real usage from users can be very different from testers’ perspective because testers aim to produce passed test cases. Therefore, unexpected user input may be able to cause error in the system.

Despite the above doubts from Nagappan et al. (2010), code coverage is still widely used as the assessment on system quality, fault detection, and testability in particular (Bruntink & van Deursen, 2006; Heitlager et al., 2007). Our study will adopt the notation of test adequacy based on code and branch coverage. We have tried to minimise the above challenge by introducing structural testing approach together with specification-based and program-based criteria. Again, specification-based criteria take into account the user input-output range value to the program internal structure. The problem with specification-based testing is that we are not able to accommodate full domain input-output space. We adopt the multiple condition coverage to reach finite applicability instead. For the review on multiple condition coverage and finite applicability on test suite, please consult section 2.2 on test coverage and test adequacy. Moreover, the main goal of the study does not aim to assess the quality of the two systems. We aim to assess the effect of the dependency injection pattern and mock on software testing. For this reason, code and branch coverage are the main indicators to witness the differences between the systems.

We define the two following formulas for code and branch coverage. The code covered by the test suite is the physical line of source code, i.e. not the IL code. In addition, the tool to measure code coverage excludes the Visual Studio generated code from the analysis. Branch coverage analysis will follow the same criteria in the calculation by ignoring the control flow branches caused by the generated code.

\[
\text{Code Coverage} = \frac{\text{Code Covered by Test Suite}}{	ext{Total Code Lines}}
\]

\[
\text{Branch Coverage} = \frac{\text{Branch Covered by Test Suite}}{\text{Total Branches}}
\]

In the studies by Bruntink & van Deursen (2006) and Heitlager et al. (2007), the authors aimed to measure testability based on the test suite. They defined two metrics particularly Number of Lines of Code (NbLOC) of the test suite and Number of Test Cases (NbTestCases). NbTestCases is calculated on the number of invocations of \texttt{assert} statement within the test cases. If a test case contains two or three \texttt{assert} statements, NbTestCases will be two or three accordingly.
We would like adopt the two metrics in the study. However, we alter the definition of the two metrics as followings to suit our need. First, NbLOC will be counted by the number of IL code as defined in above section. Second, instead of counting the number of invocations of \texttt{assert} statement within the test cases, we count the number of test cases directly. The reason is that in some test cases of our test projects, there is no \texttt{assert} statement due to the purpose of the test to catch known exceptions from the program execution. Thus, in this definition, even though there are more than one \texttt{assert} statement within a test case, we count it as one.

Our study will implement the test cases using the Microsoft unit testing framework. The framework denotes each test case with the attribute \texttt{[TestMethod]}. As the name suggested, the attribute is marked on any method of a test class. Thus, NbTestCases will be counted on the number of methods with the attribute \texttt{[TestMethod]}.

\[
\text{NbTestCases} = \text{Count(TestCase)}
\]
Chapter 4

Project Implementation

The chapter will describe in details the projects under study and necessary tools which are implemented within the entire study. First, overview of the Codisclose project will be discussed along with its components, database schema, and process overview. Next, we introduce the revision of the Codisclose project with the name CodiscloseDI in which the dependency injection is applied on every feature of the application. Additionally, four test projects for both applications are constructed for comparison.

4.1 Overview of Codisclose

Codisclose was originally developed and initiated by Jonas Sjöström, Ph.D. from the Department of Informatics and Media, Uppsala University. Later, a group of students during the course on Agile Methods continued the development. The Codisclose aims to depict two activities of software development during the project life cycle. One activity is the discussion among project stakeholders to define user requirements and raise any issue regarding the project implementation. The other is from the developers’ activities in coding. The two activities are illustrated in the analysed histogram (figure 4.1). The solid line represents the activity of developers’ coding, whereas the dotted line illustrates the discussion activity among stakeholders.

The Codisclose requires two data sources to produce the graph in figure 4.1. One source of data is from the project source code stored in an online repository. Currently, it supports only Subversion repository. Another data source is the written documents from the project meeting. The documents can be in the formats of Microsoft Word Document (.docx and .doc), PDF file (.pdf), and text file (.txt).
4.2 Codisclose Process

To produce the output as in figure 4.1, the Codisclose follows five process steps as depicted in figure 4.2. The followings are the details of each process step:

Setup: as the Codisclose needs the two data sources, users are required to set up the location of the sources. For the online repository, credential information such as user name, password and address of the repository (URI) is required. Document source is stored in any folder on the local machine. The Codisclose defines the folder as Collection. Thus, the users need to specify the location where the Codisclose can retrieve the documents.

In addition to the above two setups, The users are required to specify the inquiry for the analysis against the two data sources. The inquiry contains keywords to be analysed with the sources. Since there are two data sources, two keyword types are defined: repository keyword and document keyword. The users have the ability to define as many keywords as they would like.

Synchronise: the Codisclose has to synchronise data from the two sources into a local database. For the repository source, the program imports each code file inside the repository which developers have committed. Any files which have been synchronised
previously will not be imported in the next synchronisation. The same procedure is applied to local documents. The program reads the content of files from the specified collection and store them in the database. Any documents already imported will not be included in later synchronisation. The users have the option not to perform synchronisation because previous stored data in the database can be used for the analysis.

*Analyse:* the inquiry keywords which have been defined in the setup step are used in the analysis. The inquiry keywords for the repository are matched against the repository data source. The program counts the number of code files whose file name matches at least one inquiry keyword. Similarly, inquiry keywords for the document are matched with the content of each document inside the specified collection. In other words, the program counts the frequency of keywords occurring within the document content. Both analyses are grouped by the file-creation date or file last-modified date.

*Display:* the analysed data from the previous step are used to draw the histogram similar to figure 4.1. Again, the solid line is drawn from the analysed data from repository keywords. The dotted line is from the analysed data from document keywords.

*Export:* the users have the options to export both analysed data and the histogram for their use. The analysed data can be exported as comma-separated values (CSV) or copied to the clipboard. Likewise, the graph can be saved as image file format or stored in the clipboard.

### 4.3 Database Schema

As already mentioned, the Codisclose synchronises the two data sources and stores them into a database. The database schema along with the entity relationship is illustrated in figure 4.3. The description of each entity is listed as followings:

- *PdfCollections* stores the location of the collection which the users specify. Initially, the Codisclose only worked with PDF document. The name of this entity is not changed even though the current version works with three different file formats.

- *PdfDocuments* stores the content of files locating inside the collection. The same reason applies to *PdfDocuments* as in the case of *PdfCollections*. 

![Figure 4.2: Codisclose Process Overview](image)
4.4 Codisclose Project

The Codisclose has been developed using Visual Studio 2012 with C# as the main programming language. The user interface is built upon Windows form application. The database is stored using Microsoft SQL Server Express on the local machine. For unit, integration, and system testing project, Microsoft unit testing framework is implemented. Another supplementary tool is AnkhSvn client for working with Subversion repository.
Other external libraries that the Codisclose depends on include SharpSvn for retrieving information from the Subversion repository, iTextSharp for reading PDF file content, and Code\textunderscore 7248\textunderscore word\_reader for reading Microsoft Word Document file content.

It should be noted that the Codisclose project needs to store data to the local database. The local database follows the database schema from figure 4.3. However, the Codisclose project does not enforce relationship constraint on each entity. Without relationship constraint between entities, it allows each entity to be independent. For instance, from figure 4.3, PdfCollections has an one-to-many relationship with PdfDocuments. Relationship constraint states that every item in PdfDocuments must have a reference to an existing item in PdfCollections. Thus, without setting relationship constraint, the above rule is not necessary for items in PdfDocuments to reference to an existing item in PdfCollections.

This is important to state the above point because it affects how the test cases need to be set up. In other words, without relationship constraint, test cases that involve PdfDocuments can be prepared without initialising PdfCollections. It reduces the amount of work to prepare pre-existing data for test to work with. Thus, it affects the test size, i.e. NbLOC metrics on test suite, as we have discussed in section 3.7.

Following the overall process of the Codisclose, we organise the project in the similar fashion. Table 4.1 gives details on the project structure. We will later compare all the metrics we have presented in chapter 3 based solely on the project structure, i.e. namespace level and application level.

The Codisclose project follows the design pattern of using static method and singleton in classes throughout the whole project. This practice makes the implementation of each functionality easy and understandable. When one function needs to be invoked in another place, the class which provides the functionality does not need to be instantiated. The function can be called using this pattern ClassName.MethodName. Therefore, classes in the Codisclose project can be seen as a bundle of methods. The data which the classes need to communicate resides within the back-end database.

While the practice of using static method and singleton pattern is convenient to developers who design the system, it poses problems in measuring cohesion metrics. Since cohesion measures the number of methods accessing the local field data, classes without local field data in the case of the Codisclose project can distort the cohesion value.
### Chapter 4. Project Implementation

#### Table 4.1: Codisclose Project Structure

<table>
<thead>
<tr>
<th>Structure</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program.cs</td>
<td>File</td>
<td>Entry point to show main form to users. It is part of the Codisclose namespace.</td>
</tr>
<tr>
<td>frmMain.cs</td>
<td>File</td>
<td>Contain most of the functionalities such as setting up collections, repositories, inquiries and inquiry keywords; triggering synchronisation; and invoking analysis. The functionalities are invoked by user interaction with the form. It is part of the Codisclose namespace.</td>
</tr>
<tr>
<td>Synchronise</td>
<td>Namespace</td>
<td>Contain synchronisation-related classes.</td>
</tr>
<tr>
<td>Analyse</td>
<td>Namespace</td>
<td>Contain analysis-related classes.</td>
</tr>
<tr>
<td>frmChart.cs</td>
<td>File</td>
<td>Display a histogram on the user interface. It also contains functionalities to save and copy graph and data to the clipboard as well as CSV file. The functionalities are triggered by user interaction with the form. It is part of the Codisclose namespace.</td>
</tr>
<tr>
<td>Export</td>
<td>Namespace</td>
<td>Contain export-related classes.</td>
</tr>
<tr>
<td>CRUD</td>
<td>Namespace</td>
<td>Contain a class with logic to create, read, update and delete operation on the database. It manipulates the functionalities provided by DataContext.</td>
</tr>
<tr>
<td>DataContext</td>
<td>File</td>
<td>Contain classes representing entities (tables) of the database and provide functionalities to perform query to the database. They are generated by Visual Studio and parts of the Codisclose namespace.</td>
</tr>
<tr>
<td>Others</td>
<td>File</td>
<td>Contain Visual Studio generated code files. They are parts of the Codisclose namespace. They will be excluded from the comparison and analysis.</td>
</tr>
</tbody>
</table>

#### 4.5 Codisclose Test Project

Codisclose test project aims to test the functionalities from the Codisclose project. The test follows the structural testing approach and targets adequacy on code and branch coverage. It contains all the test cases in three levels: unit test, integration test and system test. Every test case in the test project follows the mechanisms from Microsoft unit testing framework. In the test suite, the three levels of testing are combined together, and unit test has only a small proportion. Thus, we would like to use “integration test” to refer to the three levels together.

Classes and their methods from the Codisclose project will have their corresponding test classes and test methods in the test project. To reach high code and branch coverage on the Codisclose project, one method of a class can have more than one test method.
Frequently, the test methods aim for passed test, failed test, and other possible execution paths introduced by the control flow statements. Listing 4.1 illustrates a typical test class from the Microsoft unit testing framework.

```csharp
[TestClass]
public class SvnImporterTest
{
    [ClassInitialize]
    public static void Initialize(TestContext context)
    {
        ...
    }

    [ClassCleanup]
    public static void Cleanup()
    {
        ...
    }

    [TestMethod]
    public void BuildIndex_Test()
    {
        ...
    }

    [TestMethod]
    public void BuildIndex_Exception_Test()
    {
        ...
    }
}
```

Listing 4.1: Typical Test Structure of Microsoft Unit Testing Framework

Attribute `[ClassInitialize]` represents a method which will be executed at the beginning of the test, whereas `[ClassCleanup]` characterises a method which will be run at the end of the test. The purpose of the two attributes are self-explanatory: initialising necessary data and cleaning up the data after use. As we defined in section 3.7, test case is marked with the attribute `[TestMethod]`. Thus, the two methods with the attribute `[ClassInitialize]` and `[ClassCleanup]` are not counted in the metrics \(Nb\text{TestCases}\); however, they are counted in the \(Nb\text{LOC}\) metrics because they contribute to the size of the test suite.

The test project also includes the user interface test cases for `frmMain.cs` and `frmChart.cs`. It is worth mentioning that it is uncommon to test user interface using unit testing framework. However, we thrive to achieve such work to reach as high as code and branch coverage of test adequacy criteria. In addition, the two classes share more than half of the entire source code, and the export feature can only be tested with
the involvement of user interface elements. Microsoft also provides user interface testing framework, but the practice is depended on the monitor screen pixel. Thus, this is device-dependent. To run the user interface test, one must have the same monitor screen resolution and dimensions as the one in which test cases have been developed.

To test the user interface of the Codisclose application, we have removed some user interface elements which need interaction from the users. The elements are message box showing information and dialogue box asking for further actions from users such as deleting items, saving files, and locating folder location. In the test cases, user interaction simulation is triggered with in the test code to ensure the correctness of application logic.

The test project for the Codisclose is organised in a similar structure as presented in table 4.1. Each class in a certain namespace from the Codisclose project will have its corresponding test cases in the test project with the name of the namespace plus “Test”. For example, the Synchronise namespace will appear as SynchroniseTest namespace. In the case of frmMain.cs and frmChart.cs, we will put test files under two namespaces instead because the two form classes contain many functionalities. Namespace MainFormTest and ChartFormTest are for frmMain.cs and frmChart.cs respectively.

Regarding the database issue, test project works with a clone of the original database which the Codisclose operates with (figure 4.4). Thus, test project does not depend on the data from the original database, and any error happening within the test cases will not affect the data of the original database. With such implementation, the caution of breaking the original program is alleviated. This point can be a compromise with mock practice in performing unit test in isolation without affecting other parts of the program.
4.6 CodiscloseDI Project

The main purpose of the whole study is to prepare two similar systems with the same user requirements for the comparison on various aspects including complexity, maintainability, testability and test suites. The new system has been developed by refactoring the Codisclose project. The new project is identical to the Codisclose in terms of functionalities and project structure. The new project is named CodiscloseDI.

One of the major differences between the two projects is the internal design structure. The CodiscloseDI implements each functionality with the dependency injection pattern. The project structure of the CodiscloseDI looks similar to the one in the Codisclose. The differences include additional classes and namespaces to follow the dependency injection practices. Table 4.2 summarises the new project structure.

Having seen in table 4.1 and table 4.2, the CodiscloseDI project introduces two additional namespaces: DataRepository and Service. The two new namespaces have the same functionalities as the ones in CRUD from the Codisclose project. CRUD contains only one class which has many responsibilities to perform queries to every entity of the database. Thus, the redesign in the CodiscloseDI is to split one class with all functionalities into manageable classes. Another reason to have two namespaces is to follow good practice of the dependency injection pattern to form modular design (Bender & McWherter, 2011; Prasanna, 2009; Seemann, 2012). The two namespaces create an intermediate layer between application logic and database logic. The Service is responsible for providing its functionalities to the application layer, while the DataRepository works with the back-end layer.

Another point is that the classes from the Service namespace combine relevant functionalities of different classes in the DataRepository. This helps reduce the number of dependencies which will be used in the application layer, especially the Windows form. All classes within the DataRepository are used inside the Service namespace.

The CodiscloseDI project has been redesigned using the same technologies as the ones in the Codisclose project. An additional library which the new project includes is dependency injection framework or container. The new project borrows the utilities provided by Ninject, an open-source dependency injection container for the .Net Framework. Our decision to select Ninject is based on knowledge and familiarity with the framework itself. Any other dependency injection framework will perform the same functionalities. Please refer to section 2.4.4 for the explanation on how the dependency injection container works.
Table 4.2: CodiscloseDI Project Structure

<table>
<thead>
<tr>
<th>Structure</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program.cs</td>
<td>File</td>
<td>Entry point to show main form to users. It is part of CodiscloseDI namespace.</td>
</tr>
<tr>
<td>CoreModule.cs</td>
<td>File</td>
<td>Contain dependency rules for the application. The rules are used to create concrete instances of the dependencies. It is part of the CodiscloseDI namespace.</td>
</tr>
<tr>
<td>frmMain.cs</td>
<td>File</td>
<td>Contain most of the functionalities such as setting up collections, repositories, inquiries and inquiry keywords; triggering synchronisation; and invoking analysis. The functionalities are invoked by user interaction with the form. It is part of the CodiscloseDI namespace.</td>
</tr>
<tr>
<td>Synchronise</td>
<td>Namespace</td>
<td>Contain synchronisation-related classes.</td>
</tr>
<tr>
<td>Analyse</td>
<td>Namespace</td>
<td>Contain analysis-related classes.</td>
</tr>
<tr>
<td>frmChart.cs</td>
<td>File</td>
<td>Display a histogram on the user interface. It also contains functionalities to save and copy graph and data to the clipboard as well as CSV file. The functionalities are triggered by user interaction with the form. It is part of the CodiscloseDI namespace.</td>
</tr>
<tr>
<td>Export</td>
<td>Namespace</td>
<td>Contain export-related classes.</td>
</tr>
<tr>
<td>DataRepository</td>
<td>Namespace</td>
<td>Contain classes with logic to create, read, update and delete operation on the database. They depend on the functionalities provided by DataContext.</td>
</tr>
<tr>
<td>Service</td>
<td>Namespace</td>
<td>Contain classes with logic to create, read, update and delete operation on the database. They depend on the functionalities provided by classes from the DataRepository namespace.</td>
</tr>
<tr>
<td>DataContext</td>
<td>File</td>
<td>Contain classes representing entities (tables) of the database and provide functionalities to perform query to the database. They are generated by Visual Studio and parts of the CodiscloseDI namespace.</td>
</tr>
<tr>
<td>Others</td>
<td>File</td>
<td>Contain Visual Studio generated code files. They are parts of the CodiscloseDI namespace. They will be excluded from the comparison and analysis.</td>
</tr>
</tbody>
</table>
From table 4.2, `CoreModule.cs` is the location where all dependency rules are stored. Recalling from section 2.4.4, the dependency rules are required by the container to resolve instances between abstract types and concrete types. Since all dependency rules are defined in `CoreModule.cs`, this class also appears to be the core of the dependency injection pattern and the CodiscloseDI project.

### 4.7 CodiscloseDI Test Projects

The CodiscloseDI contains three test projects with the following names: `CodiscloseDIMockTest`, `CodiscloseDIUnitTest` and `CodiscloseDIMockAndUnitTest`. The first test project will utilise the mock practices in testing. All the namespaces except `Service` in the CodiscloseDI project will be tested. The second test project tests all components in the CodiscloseDI project without involving any mock practices. If the components need to interact with other external resources, the test cases have to initialise the required resources. The third project as the name suggested applies both technologies from the previous two test projects. As mentioned in the above section, the `Service` namespace is the replication of `DataRepository`; therefore, the `Service` namespace does not have any test cases.

The mock test project depends on the functionalities and features of the external mocking library. `Moq` is an open-source mocking framework for the .Net Framework. It provides both stub and mock behaviours which are necessary for our test cases. The selection of Moq is based on our previous knowledge and familiarity with the library itself. There are other mocking frameworks available to use, and they provide similar functionalities as the ones from Moq. Examples on how Moq is implemented on the test cases can be found in section 2.5.1.

The three test projects follow the same principle applied in the Codisclose test project. The structure of the namespaces inside test projects are prepared to match the namespaces from the CodiscloseDI project. Visual elements requiring user interaction are removed from the application for testing. Test cases are created to reach high code and branch coverage.

The two test projects excluding `CodiscloseDIMockTest` will access the same back-end database for their test cases. Following the same technique as the Codisclose test project, the test database is the clone of the original database eliminating the risk of breaking and introducing data inconsistency in the original database. The mock test project interacts with stub data or stand-in data created for the test purpose. Figure 4.5 illustrates the back-end databases for the CodiscloseDI and its test projects.
4.8 Project Comparisons

Our comparisons on both projects will target the namespace level and entire project level. Since the two projects are structured similarly based on application specifications, the comparison is relatively straightforward for both projects. Additionally, the design pattern of both projects can be considered as consistence throughout the entire applications. The Codisclose is designed implementing static method pattern for all functionalities, whereas the CodiscloseDI manipulates the dependency injection pattern in each specification as well. For this reason, it allows us to measure the extent of differences between the two patterns.

Table 4.3 summarises the corresponding namespaces for the comparison from the two projects. All namespaces will be measured against the metrics we have discussed in chapter 3. It should be noted that CRUD namespace has its two counterparts: Service and DataRepository from the CodiscloseDI.

Likewise, the test suites from both projects will be compared on the basis of namespaces against the test suite metrics as discussed in chapter 3. Table 4.4 summarises the namespaces and metrics for the study.

To calculate the metrics we have described in chapter 3, various code metrics tools are required. This study uses four different metrics calculation tools as below:
### Table 4.3: Namespaces for Comparison on Various Metrics

<table>
<thead>
<tr>
<th>Codisclose Project</th>
<th>CodiscloseDI Project</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codisclose</td>
<td>CodiscloseDI</td>
<td>Size-related metrics,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cyclomatic complexity,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Maintainability index,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Coupling metrics,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cohesion metrics</td>
</tr>
<tr>
<td>Synchronise</td>
<td>Synchronise</td>
<td></td>
</tr>
<tr>
<td>Analyse</td>
<td>Analyse</td>
<td></td>
</tr>
<tr>
<td>Export</td>
<td>Export</td>
<td></td>
</tr>
<tr>
<td>CRUD</td>
<td>DataRepository</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Service</td>
<td></td>
</tr>
</tbody>
</table>

### Table 4.4: Test Project Namespaces for Comparison on Various Metrics

<table>
<thead>
<tr>
<th>Codisclose Test Project</th>
<th>CodiscloseDI Test Projects</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>MainFormTest</td>
<td>MainFormTest</td>
<td>Code coverage,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Branch coverage,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NbLOC,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NbTestCases</td>
</tr>
<tr>
<td>SynchroniseTest</td>
<td>SynchroniseTest</td>
<td></td>
</tr>
<tr>
<td>AnalyseTest</td>
<td>AnalyseTest</td>
<td></td>
</tr>
<tr>
<td>ChartFormTest</td>
<td>ChartFormTest</td>
<td></td>
</tr>
<tr>
<td>ExportTest</td>
<td>ExportTest</td>
<td></td>
</tr>
<tr>
<td>CRUDTest</td>
<td>DataRepositoryTest</td>
<td></td>
</tr>
</tbody>
</table>

- **Code Metrics Values** is available from the Visual Studio 2012. The tool calculates five different metrics. Four of them are NbLOC (IL Code), Cyclomatic complexity, Maintainability index and Class coupling (unique coupling). The tool allows us to get the metrics value from the method level to the application level. Our study captures the four metrics from this built-in tool.

- **Code Coverage Analysis** is available from the Visual Studio 2012. The tool measures the number and percentage of code coverage by the test suite. Code coverage analysis is retrieved from the tool. However, it analyses only the code sections written by developers only.

- **NDepend** is a commercial add-on tool for Visual Studio. The tool can calculate numerous metrics from the method level to the application level. These include coupling (repeating coupling), cohesion, cyclomatic complexity and other size-related metrics. However, repeating coupling, cohesion, and size-related metrics are needed from the NDepend. In the study, we use the trial version of NDepend v5.
- *NCover* is a commercial add-on tool for Visual Studio. The tool measures the branch coverage exercised by the test suite. The branch coverage analysis is compiled from the NCover support. We use the trial version of NCover v4.5.
Chapter 5

Data Analysis

The purpose of the chapter is to compare the studied projects including the test suites with quantified measurements of the selected metrics. The chapter will discuss various aspects of the comparisons and give plausible explanations behind the discrepancies as well. Differentiations of the Codisclose and CodiscloseDI project are analysed following by their test suite comparisons. Finally, testability of both systems is discussed.

5.1 Codisclose and CodiscloseDI

5.1.1 Size-Related Metrics

The first comparison will be on the size-related metrics. The metrics give the overall description and basis of both projects. Table 5.1 lists the metrics on the namespace level and application level. NbMethods excludes the methods from interface and abstract class (abstract type for both terms).

Considering the overall summary of the two projects, we noted that CodiscloseDI introduces a larger number of types, fields, and methods (table 5.1). This results from the modularisation of the dependency injection pattern to form small units in the code base. The increase in the number of types results from introducing the abstract types within the whole CodiscloseDI project.

An important point to notice is the CRUD, DataRepository, and Service namespace at the bottom of the table 5.1. The CRUD has one class which performs all the functionalities to the underlying database. In the CodiscloseDI project, the functionalities are divided into smaller units with 17 types residing within the namespace (nine of them are abstract types). For this reason, the DataRepository has a higher number of fields and
methods. As mentioned earlier, the Service namespace combines similar functionalities from the DataRepository and results in a smaller number of types. The divisions within the DataRepository and the introduction of the Service namespace have repercussion on other metrics such as cyclomatic complexity, maintainability, and coupling due to the increase of NbMethods, NbFields, and NbLOC.

Even though the CodiscloseDI project introduces more number of fields, methods, and an additional namespace, the value of NbLOC metrics is still comparable with its counterpart. Modularisation of the dependency injection pattern helps reduce the repeating pieces of source code as witnessing the increase of methods.

### 5.1.2 Cyclomatic Complexity

As discussed in section 3.3, CC measures the complexity of a method. It is natural to assume that the CC value from the CodiscloseDI is higher than the one from the Codisclose owing to the fact that the CodiscloseDI introduces more methods. Table 5.2 includes the CC and CC’ (Without Abstract) metrics for the comparison.

From the CC definition, every method has at least CC value of one. As the dependency injection pattern encourages modularisation, there is an increase in the number of methods. Literally, the total CC value of the CodiscloseDI is much higher than the one from the Codisclose. However, the CodiscloseDI project contains a number of abstract types in each of its namespaces. Even though the abstract type does not have NbLOC value
Table 5.2: Cyclomatic Complexity on Namespace and Application Level

<table>
<thead>
<tr>
<th>Namespace</th>
<th>CC</th>
<th>CC' (Without Abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codisclose Overall</td>
<td>540</td>
<td>540</td>
</tr>
<tr>
<td>CodiscloseDI Overall</td>
<td>711</td>
<td>600</td>
</tr>
<tr>
<td>Codisclose</td>
<td>352</td>
<td>352</td>
</tr>
<tr>
<td>CodiscloseDI</td>
<td>355</td>
<td>348</td>
</tr>
<tr>
<td>Codisclose.Synchronise</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>CodiscloseDI.Synchronise</td>
<td>50</td>
<td>47</td>
</tr>
<tr>
<td>Codisclose.Analyse</td>
<td>46</td>
<td>46</td>
</tr>
<tr>
<td>CodiscloseDI.Analyse</td>
<td>36</td>
<td>34</td>
</tr>
<tr>
<td>Codisclose.Export</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>CodiscloseDI.Export</td>
<td>51</td>
<td>45</td>
</tr>
<tr>
<td>Codisclose.CRUD</td>
<td>76</td>
<td>76</td>
</tr>
<tr>
<td>CodiscloseDI.DataRepository</td>
<td>129</td>
<td>78</td>
</tr>
<tr>
<td>CodiscloseDI.Service</td>
<td>90</td>
<td>48</td>
</tr>
</tbody>
</table>

(abstract class with only abstract methods and interface type), it still has its associated CC value. If removing the CC value of those abstract types from the analysis, the total CC' value of the CodiscloseDI decreases to 600. Regarding the difference of NbMethods between the two projects, the higher CC' of 60 points on the application level is justified.

Among all classes in both systems, there are only two classes which have the highest CC value. They are the Windows form classes (frmMain.cs and frmChart.cs) holding user interface elements. The CC of frmMain.cs is 133 and 148, while frmChart.cs has 74 and 55 for the Codisclose and CodiscloseDI respectively (see appendix A for details). It is important to identify the two classes because they contain error-prone code and interact with nearly all system components. In sections on test suites, we will revisit the two Windows form classes for further discussion.

In a general sense, the dependency injection pattern encourages the usage of interface and abstract class to fulfill its benefits. By adding more abstract types, additional increase in complexity is expected. However, the existence of supplementary abstract types in the project has its own influence. The next section will demonstrate the effect of abstract types on the system maintainability index.
5.1.3 Maintainability Index

MI is a good indicator to compare both projects. As already seen from the MI formula, three values of Halstead’s volume, CC, and NbLOC are included to calculate the MI value. MI scores on the namespace and application level are summarised in table 5.3. MI at the namespace level is the average of MI values from its classes. This also applies to the MI point at the application level.

From the discussion in section 3.4, MI value peaks at 100. If considering the value of 50 is the threshold value, both projects’ MI values fall into the acceptable range, and the CodiscloseDI project achieves a better value. Additionally, every namespace (except the CodiscloseDI namespace) of the CodiscloseDI system has a higher MI than the one from the Codisclose.

We already discussed that the CC value of the CodiscloseDI is much higher if we consider the involvement of abstract types in the analysis. In contrast to CC metrics, MI value from the CodiscloseDI is better by including the abstract types. The reason is that the abstract type has the value NbLOC of zero making the MI value equal to 100. When the average of MI on the namespace and application level is calculated, the value tends to have a better score.

However, if we remove the MI value of abstract types from the CodiscloseDI, the overall and each namespace MI’ values are comparable to the values of the Codisclose project as demonstrated in table 5.3 at the third column. For this reason, we believe the dependency injection pattern helps increase the system maintainability with the practice
Table 5.4: Repeating and Unique Coupling on Namespace and Application Level

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Coupling</th>
<th>UCoupling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codisclose Overall</td>
<td>604</td>
<td>224</td>
</tr>
<tr>
<td>CodiscloseDI Overall</td>
<td>991</td>
<td>258</td>
</tr>
<tr>
<td>Codisclose</td>
<td>299</td>
<td>136</td>
</tr>
<tr>
<td>CodiscloseDI</td>
<td>357</td>
<td>177</td>
</tr>
<tr>
<td>Codisclose.Synchronise</td>
<td>100</td>
<td>42</td>
</tr>
<tr>
<td>CodiscloseDI.Synchronise</td>
<td>129</td>
<td>52</td>
</tr>
<tr>
<td>Codisclose.Analyse</td>
<td>92</td>
<td>35</td>
</tr>
<tr>
<td>CodiscloseDI.Analyse</td>
<td>83</td>
<td>34</td>
</tr>
<tr>
<td>Codisclose.Export</td>
<td>26</td>
<td>14</td>
</tr>
<tr>
<td>CodiscloseDI.Export</td>
<td>49</td>
<td>20</td>
</tr>
<tr>
<td>Codisclose.CRUD</td>
<td>87</td>
<td>67</td>
</tr>
<tr>
<td>CodiscloseDI.DataRepository</td>
<td>302</td>
<td>64</td>
</tr>
<tr>
<td>CodiscloseDI.Service</td>
<td>71</td>
<td>26</td>
</tr>
</tbody>
</table>

of using abstract type. Without abstract type, the dependency injection performs comparably to the ad hoc design.

If we reviewed the NbLOC and CC of the Codisclose project, its MI score should have been better than the CodiscloseDI’s MI point since its NbLOC and CC are lower than the ones from the CodiscloseDI. The possible reason is that the tool calculating the MI value does not disclose the Halstead’s volume. We are left with the final result from the calculation. The plausible explanation is that Halstead’s volume of the Codisclose is higher. Another thing is the MI formula itself. The use of natural logarithms (ln) and the trimming of the original MI value from 171 to 100 make the difference relatively low; hence, comparable even though there are differences in the NbLOC and CC between both projects.

5.1.4 Coupling Metrics

We calculate the coupling between classes based on the criteria we have discussed in section 3.5. Table 5.4 provides the coupling measurements in terms of repeating coupling and unique coupling. The unique coupling at the application level is not the sum of unique coupling from its namespaces. Instead, it filters to form further distinct count.

Again, due to the large number of types, fields, and methods in the CodiscloseDI project, the number of repeating couplings is much higher in comparison to the one in the Codisclose system. Even though we exclude the number of couplings from the Service namespace, the huge difference of the coupling still holds. The plausible explanation
is because of the modularisation from the dependency injection pattern to form small units inside the code structure.

However, if we consider the unique count on coupling, the difference between the two systems becomes comparable in spite of the modularisation code structure into small units and the introduction of new abstract types within the CodiscloseDI. This is contradictory to the wide discrepancy from the repeating coupling. It is natural for a system with a higher number of methods and fields to have strong coupling. Using unique coupling removes the strong influence of the difference in numbers of methods and fields between similar systems such as the Codisclose and CodiscloseDI. Another point is that the CodiscloseDI has 20 extra types over the Codisclose project (table 5.1). These additional types more or less assist the increasing number of repeating coupling and unique coupling for the CodiscloseDI project. Moreover, the supplementary Service namespace plays its role in rising the coupling as well. By excluding the coupling from the Service namespace, the discrepancy between the two projects on unique coupling is even smaller.

### 5.1.5 Cohesion Metrics

Before discussing the cohesion metrics, it is better to provide a short review on LCOM. LCOM2 value ranges between zero and one, and LCOM3 value is between zero and two. LCOM3 value above one denotes serious lack of cohesion inside a class. Another important point is that LCOM is the inversion of cohesion. The smaller value of LCOM is desirable.

For LCOM metrics, analysis with maximum value makes more sense than the measurement of average on the namespace level. The problem is that within each namespace, there are several classes with different LCOM value. We cannot perform mean on LCOM value of all classes since this practice makes the LCOM value on the namespace much lower than it should be. Moreover, classes with only constructor and property `get` and `set` have the LCOM value of zero decreasing the average LCOM even further.

Table 5.5 presents the highest LCOM2 and LCOM3 value on the namespace level. The number in the parenthesis after the namespace denotes the number of types excluding the abstract ones from each namespace of the CodiscloseDI project.

The Codisclose project manipulates the pattern of `static` method in nearly all the functionalities. As we have mentioned in the section 3.6, LCOM value for a class with only `static` members will have the value of zero as in the case of the Analyse namespace. This prevents us from making reasonable comparison between the projects. If comparing
### Table 5.5: Max LCOM2 and Max LCOM3 on Namespace Level

<table>
<thead>
<tr>
<th>Namespace (NbTypes)</th>
<th>Max LCOM2</th>
<th>Max LCOM3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codisclose (18)</td>
<td>0.95</td>
<td>0.97</td>
</tr>
<tr>
<td>CodiscloseDI (16)</td>
<td>0.96</td>
<td>0.97</td>
</tr>
<tr>
<td>Codisclose.Synchronise (5)</td>
<td>0.56</td>
<td>0.7</td>
</tr>
<tr>
<td>CodiscloseDI.Synchronise (6)</td>
<td>0.63</td>
<td>0.73</td>
</tr>
<tr>
<td>Codisclose.Analyse (4)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CodiscloseDI.Analyse (2)</td>
<td>0.33</td>
<td>0.5</td>
</tr>
<tr>
<td>Codisclose.Export (2)</td>
<td>0.57</td>
<td>0.67</td>
</tr>
<tr>
<td>CodiscloseDI.Export (4)</td>
<td>0.67</td>
<td>0.8</td>
</tr>
<tr>
<td>Codisclose.CRUD (1)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CodiscloseDI.DataAccess (8)</td>
<td>0.5</td>
<td>0.57</td>
</tr>
<tr>
<td>CodiscloseDI.Service (4)</td>
<td>0.62</td>
<td>0.67</td>
</tr>
</tbody>
</table>

The Synchronise and Export between the two projects, LCOM2 and LCOM3 from the CodiscloseDI project appears to have higher value, but still in comparable range.

The highest value of LCOM2 and LCOM3 can be found in the Codisclose and CodiscloseDI namespace. Both namespaces contain two Windows form classes. As you might have known, the Windows form class contains all the visual element declarations to render different controls to users. All event handlers on the Windows controls reside in such class as well. For this reason, the Windows form class is inherently lack of cohesion. The dependency injection on the form class does not help improve the cohesion either.

Another namespace worth discussing is CRUD. It contains only one class `DataAccess` which implements the singleton pattern and `static` method. Although it is a very large class with many couplings and methods, the LCOM2 and LCOM3 is zero. This is due to the fact that LCOM calculates the number of methods accessing the fields, in this case only one field. Its counterparts which divide the functionalities into smaller units have higher LCOM2 and LCOM3 value denoting inferior cohesion.

LCOM3 has the maximum value of two, and value one represents a problematic class. Considering the LCOM2 and LCOM3 of the form classes, we found that the validity of LCOM3 is questionable. The two Windows form classes of both projects have the LCOM2 value closely to one (the maximum value) denoting serious lack of cohesion, while the LCOM3 is comparable to LCOM2 (less than the threshold value one). Thus, LCOM3 does not appear to be a good indicator of cohesion. Because there is no threshold value for LCOM2, it is hard to decide whether the classes using the dependency injection are lack of cohesion. The documentation from the NDepend tool sets the threshold value of LCOM2 to 0.8 (NDepend, n.d.). If we follow the same threshold value on LCOM2,
Table 5.6: Test Suite Metrics of the Four Test Projects

<table>
<thead>
<tr>
<th>Project Name</th>
<th>NbLOC</th>
<th>NbTestCases</th>
<th>% Code Coverage</th>
<th>% Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoUnitTest</td>
<td>1325</td>
<td>207</td>
<td>98.07%</td>
<td>97.75%</td>
</tr>
<tr>
<td>DIUnitTest</td>
<td>1351</td>
<td>237</td>
<td>98.17%</td>
<td>97.70%</td>
</tr>
<tr>
<td>DIMockTest</td>
<td>2286</td>
<td>259</td>
<td>92.67%</td>
<td>90.29%</td>
</tr>
<tr>
<td>DIMockUnitTest</td>
<td>1718</td>
<td>264</td>
<td>99.64%</td>
<td>99.27%</td>
</tr>
</tbody>
</table>

all classes using the dependency injection pattern fall into the acceptable range (except the Windows form classes).

5.2 Codisclose and CodiscloseDI Test Projects

First, we compare the fundamental features of the four test projects in terms of the test suite metrics from chapter 3. Table 5.6 summaries the metrics from the four test projects at the application level. We have shortened the name of the test projects as CoUnitTest for CodiscloseUnitTest, DIUnitTest for CodiscloseDIUnitTest, DIMockTest for CodiscloseDIMockTest, and DIMockUnitTest for CodiscloseDIMockAndUnitTest.

It should be noted that the percentage code and branch coverage are measured from the Codisclose and CodiscloseDI project. We have adjusted the score on these two metrics because the tool calculating the code coverage does not analyse the entity model classes which represent the database tables. The reason of the exclusion is due to the fact that Visual Studio generated the entity code. The tool measures only the code written by developers. Another important point is that the test suites do not aim to test the generated code either. Test cases were designed to cover the written code only. In the case of branch coverage, the NCover tool includes the entity model classes into the analysis.

Referencing the table 5.6 on NbLOC, we noticed that the DIMockTest project which employs solely the mock practice is the largest test suite in terms of size. It produces 2286 lines of IL code and second highest in NbTestCases. The reason is that mock requires initialisation setup, stub data setup, and behaviour verification on the methods under test. The other test suites without depending on the mock practice achieve smaller size. Furthermore, the mock test suite effectiveness is the lowest among the other. It covers only 92.67 per cent and 90.29 per cent on code and branch coverage respectively.
Table 5.7: Code and Branch Coverage on CodiscloseDI from DIMockTest

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Code Covered</th>
<th>Total Code</th>
<th>% Code Coverage</th>
<th>Branch Covered</th>
<th>Total Branch</th>
<th>% Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodiscloseDI Overall</td>
<td>2580</td>
<td>2784</td>
<td>92.67%</td>
<td>865</td>
<td>958</td>
<td>90.29%</td>
</tr>
<tr>
<td>CodiscloseDI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>frmMain.cs</td>
<td>1882</td>
<td>1933</td>
<td>97.36%</td>
<td>473</td>
<td>511</td>
<td>92.56%</td>
</tr>
<tr>
<td>frmChart.cs</td>
<td>397</td>
<td>397</td>
<td>99.52%</td>
<td>99</td>
<td>101</td>
<td>98.02%</td>
</tr>
<tr>
<td>Synchronise</td>
<td>226</td>
<td>226</td>
<td>100%</td>
<td>83</td>
<td>85</td>
<td>97.65%</td>
</tr>
<tr>
<td>Analyse</td>
<td>72</td>
<td>72</td>
<td>100%</td>
<td>63</td>
<td>63</td>
<td>100%</td>
</tr>
<tr>
<td>Export</td>
<td>108</td>
<td>108</td>
<td>100%</td>
<td>90</td>
<td>90</td>
<td>100%</td>
</tr>
<tr>
<td>DataRepository</td>
<td>292</td>
<td>292</td>
<td>100%</td>
<td>156</td>
<td>156</td>
<td>100%</td>
</tr>
<tr>
<td>Service</td>
<td>0</td>
<td>153</td>
<td>0%</td>
<td>0</td>
<td>53</td>
<td>0%</td>
</tr>
</tbody>
</table>

The CoUnitTest is the smallest test suite in terms of NbLOC and NbTestCases. The test suite (DIUnitTest) for the CodiscloseDI project containing more test cases than its counterpart does not mean the CodiscloseDI is harder to be tested. It contributes to the increasing number of methods inside the project. Noticeably, the effectiveness of the CoUnitTest is as high as and comparable to the DIUnitTest project which applies the same technique in constructing the test suite. If comparing the ratio of NbTestCases in relation to code and branch coverage, the CoUnitTest achieves better result and requires less effort to produce such effective test suite.

The test suite which incorporates the mock and integration test together performs the best on the percentage of code and branch coverage. It nearly completes 100 per cent in both coverages. Nonetheless, it requires the highest NbTestCases to reach such promising result. We shall dive into further details to see what contributes to test suite effectiveness of each test project.

5.2.1 DIMockTest

Table 5.7 gives the overview of the test suite effectiveness of the DIMockTest project. As mentioned in the above section, code coverage has been adjusted to ignore the Visual Studio generated code. The Code Covered column is the number of physical lines of code.

The reason leading the DIMockTest project to have the lowest code and branch coverage stems from the Service namespace. The test suite does not test the Service namespace.
because all functionalities in the namespace are identical to the ones in the DataRepository. If we exclude the Service namespace code and branches from the overall scores, the test suite achieves similar coverages to the other test suites.

Recall that the test project includes two test namespaces: MainFormTest and ChartFormTest. The namespaces aim to test the user interface forms targeting \texttt{frmMain.cs} and \texttt{frmChart.cs} respectively. By testing only two classes in the CodiscloseDI, the test suite already achieve more than half of the total code and total branches. The two form classes contain the interactions in the whole systems and are the most error-prone sections. Therefore, testing the two classes requires a great deal of effort and test cases to reach better coverage.

If we look into the \texttt{frmMain.cs} coverage alone, the test cases achieve nearly full code coverage (only three lines of code cannot be covered). Similarly, the test cases for \texttt{frmChart.cs} cover 100 per cent lines of code. In the case of branch coverage, there are two remaining branches from both classes that cannot be covered by the test suite. However, this represents the facilitation provided by the mock practice to cover nearly full code and branch coverage.

The Synchronise namespace are fully code covered, but two branches are left uncovered. The explanation is that mock cannot produce stub data to cover the two branches. Technically, the data needed to cover the branches are related to the online repository. Since the third-party library \textit{SharpSvn} does not expose any public constructor allowing mock to initialise the object, the data is retrieved originally from the online repository. Controlling the incoming data is challenging just to cover the two branches and would require more time and effort.

### 5.2.2 CoUnitTest and DIUnitTest

Having been seen in table 5.6, the percentage of code and branch coverage of the CoUnitTest and DIUnitTest are comparable. We shall discuss the not-covered sections of the two test suites. Table 5.8 below summarises the scores.

The sections that the test suites cannot cover reside within the Codisclose and CodiscloseDI namespace, specifically the \texttt{frmMain.cs} and \texttt{frmChart.cs}. As discussed earlier, the two Window form classes are the most problematic and complex within the whole system. For both test suites, the lines and branches of code which are left by the test suites are parts of the exception handling. The test suites have been constructed to catch known exceptions, producing
them appears to be a challenging task and may take more effort from the developers or testers.

From the above discussion on the DIMockTest, the test suite is able to cover nearly perfect code and branch coverage for `frmMain.cs` accounting for the facilitation of mock to produce exception easily within the test cases. Nevertheless, testing other system functionalities can be accomplished with integration test and system test technique. The evidence can be witnessed when discussing the effectiveness of the DIMockUnitTest in the next section.

Regarding the CRUD namespace of the Codisclose system and the Synchronise of the CodiscloseDI, the uncovered code and branches are related to online repository. We think it is not the problem of the test cases, but the data retrieved from the repository. This is the same case as in the above discussion with the mock practice.

### 5.2.3 DIMockUnitTest

Table 5.9 summarises the code and branch coverage by the test suite from the DIMockUnitTest project. The DIMockUnitTest employs two techniques in creating test cases: integration test and mock test.
Table 5.9: Code and Branch Coverage on CodiscloseDI from DIMockUnitTest

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Code Covered</th>
<th>Total Code</th>
<th>% Code Coverage</th>
<th>Branch Covered</th>
<th>Total Branch</th>
<th>% Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodiscloseDI Overall</td>
<td>2774</td>
<td>2784</td>
<td>99.64%</td>
<td>951</td>
<td>958</td>
<td>99.27%</td>
</tr>
<tr>
<td>CodiscloseDI</td>
<td>1923</td>
<td>1933</td>
<td>99.48%</td>
<td>506</td>
<td>511</td>
<td>99.02%</td>
</tr>
<tr>
<td>frmMain.cs</td>
<td>1450</td>
<td>1453</td>
<td>99.52%</td>
<td>258</td>
<td>260</td>
<td>99.23%</td>
</tr>
<tr>
<td>frmChart.cs</td>
<td>397</td>
<td>397</td>
<td>100%</td>
<td>99</td>
<td>101</td>
<td>98.02%</td>
</tr>
<tr>
<td>Synchronise</td>
<td>226</td>
<td>226</td>
<td>100%</td>
<td>83</td>
<td>85</td>
<td>97.65%</td>
</tr>
<tr>
<td>Analyse</td>
<td>72</td>
<td>72</td>
<td>100%</td>
<td>63</td>
<td>63</td>
<td>100%</td>
</tr>
<tr>
<td>Export</td>
<td>108</td>
<td>108</td>
<td>100%</td>
<td>90</td>
<td>90</td>
<td>100%</td>
</tr>
<tr>
<td>DataRepository</td>
<td>292</td>
<td>292</td>
<td>100%</td>
<td>156</td>
<td>156</td>
<td>100%</td>
</tr>
<tr>
<td>Service</td>
<td>153</td>
<td>153</td>
<td>100%</td>
<td>53</td>
<td>53</td>
<td>100%</td>
</tr>
</tbody>
</table>

If comparing with the test effectiveness of the DIMockTest and DIMockUnitTest (referencing table 5.7 and 5.9 together), we noticed that three sections do not change any value for both code and branch coverage, namely frmMain.cs, frmChart.cs and the Synchronise namespace. The increase in test effectiveness is due to the interaction of components within the frmMain.cs when performing the integration test by the test suite.

It is wise to detail how the test cases in the DIMockUnitTest are constructed. First, all test cases from the DIUnitTest are reinstated within the DIMockUnitTest. Second, the sections of code which are left without coverage are tested using mock technique. Hence, the increasing number of NbTestCases on the DIMockUnitTest is due to the cooperation with mock test cases.

The selected mock test cases for the DIMockUnitTest are to cover the exception handling code which the integration test cannot achieve. The exact result appears to suggest that the mock technique improves the test effectiveness when dealing with exception handling section. For normal cases, the integration test technique is effective to cover most parts of the system.

5.3 Testability

According to the IEEE (1990), testability is defined as the ease of a system, component or requirement which allows test criteria to be created and the ease of performing the test execution to verify the satisfaction of the test criteria. The above definition gives direction in our discussion when relevant metrics are available in hands.
Table 5.10: Summary of Metrics on Codisclose and CodiscloseDI Project

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Codisclose</th>
<th>CodiscloseDI</th>
</tr>
</thead>
<tbody>
<tr>
<td>NbTypes</td>
<td>30</td>
<td>59</td>
</tr>
<tr>
<td>NbFields</td>
<td>203</td>
<td>233</td>
</tr>
<tr>
<td>NbMethods</td>
<td>345</td>
<td>417</td>
</tr>
<tr>
<td>NbLOC</td>
<td>2538</td>
<td>2558</td>
</tr>
<tr>
<td>CC$'$ (Without Abstract)</td>
<td>540</td>
<td>600</td>
</tr>
<tr>
<td>MI$'$ (Without Abstract)</td>
<td>73</td>
<td>73</td>
</tr>
<tr>
<td>Coupling</td>
<td>604</td>
<td>912</td>
</tr>
<tr>
<td>UCoupling</td>
<td>224</td>
<td>258</td>
</tr>
<tr>
<td>Max LCOM2</td>
<td>0.95</td>
<td>0.96</td>
</tr>
<tr>
<td>Max LCOM3</td>
<td>0.97</td>
<td>0.97</td>
</tr>
<tr>
<td>NbLOC (Test Suite)</td>
<td>1325</td>
<td>1351</td>
</tr>
<tr>
<td>NbTestCases</td>
<td>207</td>
<td>237</td>
</tr>
<tr>
<td>Code Not Covered (Total Code)</td>
<td>53 (2749)</td>
<td>51 (2784)</td>
</tr>
<tr>
<td>% Code Coverage</td>
<td>98.07%</td>
<td>98.17%</td>
</tr>
<tr>
<td>Branch Not Covered (Total Branch)</td>
<td>20 (890)</td>
<td>22 (958)</td>
</tr>
<tr>
<td>% Branch Coverage</td>
<td>97.75%</td>
<td>97.70%</td>
</tr>
</tbody>
</table>

Table 5.10 summarises all the metrics of both projects. The metrics related to the test suites are selected from the CoUnitTest and DIUnitTest. All metrics for the CodiscloseDI project have a higher score than the ones from its counterpart. The metrics on complexity and maintainability reflect the larger number on branches of the program execution of the CodiscloseDI. Additionally, the size-related and both coupling metrics from the CodiscloseDI have significant effect on the test suite. Particularly, higher NbTestCases results from the increasing number on size-related and coupling metrics. The higher number on NbTestCases implies that the CodiscloseDI project needs more testing effort from testers to maintain the whole system.

Regarding the test effectiveness, both test suites achieve a similar scale on percentage code and branch coverage. From the earlier discussion, both test suites cannot achieve full coverage owing to the unknown exception handling and uncontrollable incoming data from the online repository. This appears that both systems’ testability suffers from the same problem. Therefore, the dependency injection does not seem to improve the testability of the software system. In contrast, it demands testers to include more test cases to help achieve a better coverage.

The testability can be improved with the coordination from mock facilities helping cover
the exception handling code as we witness in the DIMockUnitTest test effectiveness. When dealing with third-party libraries which limit the ability to create stub data, mock faces the challenge to improve the testability as well. The integration test appears to be effective to cover a great deal of code sections and branch executions.
Chapter 6

Conclusions

The final chapter will try to conclude on what has been done during the whole study. These include answering the research questions, comparing the result from another research study with a similar purpose, discussing the limitations, and giving directions for future work.

6.1 Dependency Injection and Code Structure Design

The study was conducted to answer four research questions as presented in chapter 1. This section aims to answer the first question: **To what extent does dependency injection improve code structure design?**

From the data analysis chapter, the effect of the dependency injection on the code structure design can be witnessed from the maintainability metrics. However, the maintainability is better when including the abstract types’ maintainability score into the analysis (table 5.3).

By reviewing the object-oriented feature of the abstract type, the better value for MI is justified. The power of the abstract type is to enable polymorphism which impacts how systems are maintained when changes happen. The abstract type defines a set of rules (methods) to perform a specific functionality without stating how such functionality will actually work. The real implementation is provided by the class implementing or inheriting the abstract type. Since there is no limit on the number of classes allowing to implement or inherit an abstract type, the changing scenarios are easier to tackle. New changing requirements can be redefined in other separate classes. The system using the abstract type can operate with the new classes seamlessly as long as the classes comply with the abstract type’s rules.
The dependency injection pattern takes the advantages of the abstract type to another level. The pattern enables the switching of different classes which follow the rules of the abstract type easily as witnessing in the dependency injection forms (constructor, property, and method/interface injection). The container for resolving the abstract types and their concrete instances makes the system maintenance even simpler and reduces developers' time and effort.

In relation to cohesion, the study appears to lack valid comparison. The Codisclose system was designed by using static method pattern making most of the components to have perfect cohesion. The problem is obvious in the CRUD namespace in which one class has the perfect cohesion in spite of overwhelming functionalities and strong coupling. If we look into other namespaces, the cohesion of the system with the dependency injection tends to be a bit lower than its counterpart (because of higher LCOM value). As discussed in section 5.1.5, the cohesion from both systems falls within the acceptable threshold score defined by the NDepend tool.

One noticeable effect from the dependency injection which later influences the rest of the system is the introduction of new types, additional field members, and methods. The effect is the main factor to add up the number of coupling in the project, especially on the repeating coupling. To fully reach its full potential, the pattern applies the practice of using abstract types rather than real implementations. Concurrently, it adds more types into the system. In addition, modularisation on system functionalities further increases system size. If we review how coupling happens, the surge in coupling score is naturally expected. Therefore, the claim to suggest that the dependency injection helps make the code loose coupling appears to be contradictory.

It appears that the dependency injection enables loose coupling design, not loose coupling code. This sounds confusing and counter-intuitive since loose coupling design should lead to loose coupling code. However, the metrics on coupling does not suggest such connection. Loose coupling design is the result of applying abstract types when constructing system components. The maintainability index reveals the effect of the abstract types on system maintainability allowing developers to deal with changes easier.

Regarding the effect on extensibility and maintainability, we would argue that the dependency injection does assist for the betterment. However, if we dive into the practice of the pattern, the assistance seems to stem from the usage of abstract types instead. For instance, the system follows the pattern without applying the abstract types on the injection pattern, the extensibility and maintainability will be comparable to the ad hoc design as the maintainability index without the abstract types gives similar overall score in the case of the Codisclose and CodiscloseDI.
Even though using the dependency injection is very likely to introduce new members to the system, which increases the score of various metrics, this does not necessarily mean that the dependency injection is not an appealing solution to software design. If we consider the loose coupling design which the pattern offers in return, the problem of rising score on various metrics can be alleviated. For instance, the CodiscloseDI project has been designed by including two similar namespaces: DataRepository and Service. The Service namespace interacts with other business-logic classes; the DataRepository works with the database. In the case the database is moved to cloud-based service, changes will only happen at the DataRepository, while the rest of the application is untouched. The simplification for long-term maintenance makes the pattern a good investment.

6.2 Dependency Injection, Mock, and Software Testing

The other research questions are related to software testing, so the discussion to answer them will be combined together under this section. The questions are as the followings:

**To what extent does dependency injection facilitate software testing?**

The testability of the system using the dependency injection does not seem to outclass its counterpart. As we have demonstrated from the test suites of both studied projects, the test suites are not able to cover the code and branch sections with the same limitation (unknown exception handling).

Because of the additional methods from the system using the dependency injection, more test cases need to be prepared to have better coverage, i.e. more effort in preparing and maintaining the test suite. What the dependency injection offers in return is testing options for testers. With the practice of using abstract types, developers can use mock, stub or even implement their own fake implementation for the purpose of testing. In contrast, the ad hoc design system has one option available: integration test. In the case the integration test is not feasible, the problem may stem from the internal structure design requiring further refactoring. For this reason, the dependency injection can be considered to facilitate software testing in terms of providing test technique alternatives.

**To what extent does mock facilitate software testing?**

Mock has been applied on two test suites of the system with the dependency injection. The result suggests the improvement on code and branch coverage if we compare with the test suites without using mock practices. The improvement mainly covers the code sections which cannot be easily covered by the integration testing approach, in our
specific case, the unknown exception handling. Along with its benefits to improve the
code and branch coverage, mock technique requires tight control over test data and
method call behaviours. As a result, mock test produces the highest number of lines of
code in comparison with the other test suites (table 5.6). This demands more effort in
producing and maintaining the test suite in the long run.

Another factor regarding mock and testing is that in our study, the test suites are con-
structed by a single developer. This helps reduce the problems with inconsistent test
data in contrast to the development in team setting. In team development environment,
mock will further assist to alleviate such problems with the ability to create in-memory
data. During the Codisclose development in team environment, the problem of in-
consistent data with the database was resolved with the creation of database on each
developer’s local machine. This technique proved to be effective, and the testing was
constructed without any problem. Such kind of problem is easily solved due to the ease
of creating the database. In the case when required testing environment is difficult to
setup in the first phase of development, mock becomes handy and facilitates the testing
procedure to another level.

Using mock practice in isolation enables true unit testing; nevertheless, it cannot ensure
the correct interactions among system components. If we recall the test effectiveness
from the DIMockTest (table 5.7), the test suite does not cover the Service namespace.
This bases on the assumption that the same functionalities from the DataRepository
namespace will act as expected when incorporating with several components. In order
to further assure the error-free interactions, only the integration test is able to detect
that.

To what extent do dependency injection and mock facilitate software testing?

In the previous discussions, test suites using the integration test cannot cover the un-
known exception handling code, while mock test is able to cover the exception handling
sections with ease. However, mock test poses risk because the interaction among com-
ponents cannot be ensured. For this reason, it appears that the two test techniques are
complementary each other. The two test techniques can be applied together owing to
the facilitation from the dependency injection to provide alternatives for testing options.

If we consider the effect of the two techniques together, the test suite is able to reach
the highest code and branch coverage (table 5.6). Still, the rise in percentage coverage
is minor if we compare to the one from the integration test only. The rise also comes
with cost because more test cases are added to cover the exception handling parts.

In short, the dependency injection enables testing options, and mock supplements the
other testing techniques to reach a better coverage. Since the effect of increasing the
coverage comes with additional test cases, the choice is left to the testers’ decision whether the investment is worth the effort.

6.3 Implications for Practice

The introduction in chapter 1 stated the involvement of Test Driven Development (TDD) in promoting testable code. Since TDD is one of the principles in Extreme Programming (XP), it also complies with the agile principles including embracing changing requirements, providing functional software, and satisfying customers’ needs (Alliance, n.d.).

In practice, the TDD principles normally come along with the dependency injection and mock. The three entities supplement each other to reach their full potentials. As TDD promotes testable code, it does not ensure good design. The dependency injection promotes good software design for maintainability and enables mock technique. Mock, in return, improves software testability. Software development practitioners, especially who follow XP methodology, will benefit from the contributions of the three components as well as conform with the agile principles.

From our discussion on the effect of the abstract type, the dependency injection utilises the power of the abstract type in its practice. Thus, those practitioners who do not follow any specific methodology can take the benefit of the abstract type into their consideration for future software development, but it does not necessarily mean that the abstract type improves the software design remarkably. It depends on the practitioners’ ability and creativity to manipulate the available arsenal.

Software metrics will remain a necessary tool to assist developers in gaining insights into their software. However, their practicality and meaning still depend on various interpretations. Specifically, system with the dependency injection will tend to have high score on coupling and complexity metrics because of its practices to add new types and method members and how the metrics are calculated. However, considering its effect on long-term maintainability, we believe the dependency injection will remain an appealing solution in software design.

6.4 Related Study

There are very few studies on the topic of dependency injection in the literature. Among the found studies, only one research study assessed the effect of the dependency injection on system maintainability owing to the similar claims we have mentioned in the paper. The study by Razina & Janzen (2007) will be discussed in relation to our result.
Razina & Janzen (2007) selected 40 projects from open-source code repositories in their study. 20 projects were designed using the dependency injection pattern, while the other were developed using an ad hoc design pattern. All the projects were written in Java using Spring as the dependency injection container. In addition to the above selection criteria, each counterpart project was chosen on the similar development technologies. However, the selected 40 projects do not based on the same user requirements. In other words, they are 40 unique projects.

To measure the effect of the dependency injection on the maintainability, the authors defined three metrics including Coupling Between Objects (CBO), Response for Class (RFC), and LCOM. The CBO, RFC, and LCOM score at the application level were calculated and divided into two data sets. Then the $t$-test was run to find whether there is statistically significant difference between the dependency injection and ad hoc design. Their result suggested that there is no trend of using the dependency injection to reduce the number of coupling and to improve system cohesion.

Their result appears to strengthen our discussion on coupling, cohesion, and maintainability. However, their experiment design and procedure are questionable. The selected projects are very different from one another and have no common user requirements. There is no explanation on the reason why the metrics scores of the two project sets are combined together for the $t$-test analysis. We cannot be sure even if their finding was statistically significant.

Another important point is that the reason why there is no differential effect may result from the large difference between each investigated project. For this reason, projects with similar size and user requirements should be selected to ensure validity and reliability.

### 6.5 Limitations

Similar to any other research study, our work has its own unavoidable limitations. First, our comparison is based on the descriptive statistics. In differential research design, complex inferential statistical tests should be performed to find whether the difference is statistically significant. The $t$-test or the analysis of variance (ANOVA) are the two common tests to find mean difference of two or more experiment groups. However, there is a problem with the sample size of the two projects under our experiment. Recall that the Codisclose has 30 types in total, whereas the CodiscloseDI has 59 types in which 20 are abstract types. The difference is almost double between the two projects. When applying the $t$-test or ANOVA on the namespace level, the sample size becomes too small to ensure validity.
There are also sophisticated techniques to deal with uneven sample sizes when comparing two means or variances. The discussion on such advanced topic in statistics is beyond our knowledge. Instead, the investigated data from the metrics appear to confirm our conclusion. Reviewing the table 5.10 on all metrics of the two projects, we can assume that the dependency injection has no effect on the structure code design and test suites since the score from the metrics are higher for the CodiscloseDI. Thus, the thought of performing statistical tests was abandoned.

Second, since the two systems are very different in terms of size-related metrics and an additional namespace, the other investigated metrics appear to show the superiority of the ad hoc designed system. This will definitely skew the result in favour of one than the other. Actually, we had considered on the point before implementing the project. There was a tug of war when deciding to introduce additional namespace. The class within the CRUD namespace has a great deal of functionalities, while the philosophy of the dependency injection and software design principle discourage such practice. If the dependency injection was designed exactly the same as in the CRUD namespace, this does not represent the practicality of the dependency injection pattern. For the realistic reason, we decided to introduce the new namespace.

Third, as with any differential research, generalisation from the study result remains in question and still needs further investigation with other systems. The result represents a case of a specific project. In addition, the Codisclose project can be considered as a relatively small project with about 2500 lines of code. With more complex system, the effect of the dependency injection might be discovered because the practice of the dependency injection plus the functionalities from its container simplifies the software maintenance effort and enables mock testing technique. Nonetheless, we are confident that the result of the study also contributes to knowledge bank on the dependency injection and its effect on code structure design and software testing.

Another point is the technique to measure test suite effectiveness. In the study, code and branch coverage are selected as the test criteria adequacy. From chapter 2, one of the purposes of software testing is to detect faults. Following this goal, mutation testing is the most suitable technique. Frankly, we also thought about applying mutation testing, but the lack of automated tool for generating mutants (artificial faults) and the intensive demand on time and resources to perform mutation testing bar our decision to include the technique. Furthermore, mutation testing is a broad topic requiring an in-depth study which is not suitable for our study purpose and scope.

Finally, the metrics for comparison were calculated at a specific point (end of development). This suggests that the metrics indicate the state of system quality of that particular point only. To truly measure software maintainability and testing, ongoing
assessment on the metrics should be conducted rigorously during the system development or when there are changes in user requirements. Then metrics on time, effort, and cost should be defined and measured.

### 6.6 Future Work

For future similar studies, the discussion on the study limitations should be taken into consideration. Regarding the mutation testing literature, there are a few research studies on the mutant testing using C# programming language. The series of studies on the mutant testing with C# projects were conducted by Anna Derezińska. One of the most recent studies (Derezińska & Szustek, 2012) documented the performance and quality of the mutation testing on many C# projects. Her research studies can give practical information on the test including its risks and strengths.

Additionally, the test suites from our studies can also be used to for the mutation testing topic since the four test suites have high code and branch coverage. One of the strengths of our test suites is the inclusion of user interface test cases. It is uncommon to have test suites for the user interface in addition to the test cases with the external database and online repository. We believe applying such test suites in the mutation testing may contribute to the knowledge on the topic of mutation testing to another level.
Appendix A

Codisclose and CodiscloseDI

Project Metrics

Typ. = Types;  
Meth. = Methods;  
Fie. = Fields;  
LOC = Lines of code;  
CC = Cyclomatic complexity;  
MI = Maintainability index;  
Coup. = Repeating coupling;  
UCoup. = Unique coupling;  
LC2 = LCOM2 (Lack of cohesion of methods);  
LC3 = LCOM3.

<p>| Table A.1: Codisclose Project Metrics on Class, Namespace and Application Level |
|---------------------------------|-------|-------|-------|-------|-------|-------|-------|-------|</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Typ.</th>
<th>Fie.</th>
<th>Meth.</th>
<th>LOC</th>
<th>CC</th>
<th>MI</th>
<th>Coup.</th>
<th>UCoup.</th>
<th>LC2</th>
<th>LC3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td>30</td>
<td>203</td>
<td>345</td>
<td>2538</td>
<td>540</td>
<td>73</td>
<td>604</td>
<td>224</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Codisclose</td>
<td>18</td>
<td>179</td>
<td>237</td>
<td>1765</td>
<td>352</td>
<td>88</td>
<td>299</td>
<td>136</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BackgroundWorker</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DoWorkArgument</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ChangePath</td>
<td>1</td>
<td>11</td>
<td>23</td>
<td>23</td>
<td>92</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CodiscloseContext</td>
<td>1</td>
<td>8</td>
<td>18</td>
<td>18</td>
<td>92</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>frmChart</td>
<td>1</td>
<td>30</td>
<td>38</td>
<td>391</td>
<td>74</td>
<td>60</td>
<td>100</td>
<td>71</td>
<td>0.95</td>
<td>0.97</td>
</tr>
<tr>
<td>frmMain</td>
<td>1</td>
<td>86</td>
<td>54</td>
<td>1217</td>
<td>133</td>
<td>50</td>
<td>107</td>
<td>82</td>
<td>0.95</td>
<td>0.97</td>
</tr>
<tr>
<td>Inquiry</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>93</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>InquiryKeyword</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>93</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>KeywordType</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>94</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PdfCollection</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>94</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PdfDocument</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>93</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PdfQuantifier</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>93</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Continued on next page
## Appendix A. Codisclose and CodiscloseDI Metrics

### Table A.1 – continued from previous page

<table>
<thead>
<tr>
<th>Name</th>
<th>Typ.</th>
<th>Fie.</th>
<th>Meth.</th>
<th>LOC</th>
<th>CC</th>
<th>MI</th>
<th>Coup.</th>
<th>UCoup.</th>
<th>LC2</th>
<th>LC3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDFRev QuantifierList</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>93</td>
<td>6</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Program</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>81</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RCAQuantifier</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>93</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RDAQuantifier</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>93</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Repository</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td>11</td>
<td>93</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Revision</td>
<td>1</td>
<td>6</td>
<td>13</td>
<td>13</td>
<td>93</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RevisionQuantifier</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>93</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Synchronise</strong></td>
<td>5</td>
<td>5</td>
<td>16</td>
<td>118</td>
<td>35</td>
<td>66</td>
<td>100</td>
<td>42</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DocImporter</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>20</td>
<td>6</td>
<td>67</td>
<td>15</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PdflImport</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>31</td>
<td>8</td>
<td>60</td>
<td>21</td>
<td>14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SubversionUtility</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>38</td>
<td>9</td>
<td>66</td>
<td>33</td>
<td>20</td>
<td>0.56</td>
<td>0.7</td>
</tr>
<tr>
<td>SvnImporter</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>5</td>
<td>72</td>
<td>15</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TxtImporter</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>21</td>
<td>7</td>
<td>66</td>
<td>16</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Analyse</strong></td>
<td>4</td>
<td>13</td>
<td>25</td>
<td>110</td>
<td>46</td>
<td>67</td>
<td>92</td>
<td>35</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LocalDocument</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>25</td>
<td>17</td>
<td>65</td>
<td>23</td>
<td>14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Analyzer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LocalRepository</td>
<td>1</td>
<td>7</td>
<td>9</td>
<td>69</td>
<td>17</td>
<td>44</td>
<td>45</td>
<td>24</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Analyzer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RelativeCodeAnalysis</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td>80</td>
<td>12</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RelativeDocument</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td>80</td>
<td>12</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Analysis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Export</strong></td>
<td>2</td>
<td>5</td>
<td>16</td>
<td>49</td>
<td>31</td>
<td>81</td>
<td>26</td>
<td>14</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CsvExport &lt;T&gt;</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>40</td>
<td>22</td>
<td>68</td>
<td>22</td>
<td>14</td>
<td>0.57</td>
<td>0.67</td>
</tr>
<tr>
<td>ExportChartData</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>94</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>CRUD</strong></td>
<td>1</td>
<td>1</td>
<td>51</td>
<td>496</td>
<td>76</td>
<td>61</td>
<td>87</td>
<td>67</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DataAccess</td>
<td>1</td>
<td>1</td>
<td>51</td>
<td>496</td>
<td>76</td>
<td>61</td>
<td>87</td>
<td>67</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table A.2: CodiscloseDI Project Metrics on Class, Namespace and Application Level

<table>
<thead>
<tr>
<th>Name</th>
<th>Typ.</th>
<th>Fie.</th>
<th>Meth.</th>
<th>LOC</th>
<th>CC</th>
<th>MI</th>
<th>Coup.</th>
<th>UCoupl.</th>
<th>LC2</th>
<th>LC3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Overall</strong></td>
<td>59</td>
<td>236</td>
<td>515</td>
<td>2558</td>
<td>711</td>
<td>84</td>
<td>991</td>
<td>258</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CodiscloseDI</td>
<td>17</td>
<td>182</td>
<td>250</td>
<td>1709</td>
<td>355</td>
<td>85</td>
<td>357</td>
<td>177</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BackgroundWorker</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>94</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DoWorkArgument</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Name</th>
<th>Typ.</th>
<th>Fie.</th>
<th>Meth.</th>
<th>LOC</th>
<th>CC</th>
<th>MI</th>
<th>Coup.</th>
<th>UCoup.</th>
<th>LC2</th>
<th>LC3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChangePath</td>
<td>1</td>
<td>11</td>
<td>23</td>
<td>23</td>
<td>92</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>CodiscloseDIContext</td>
<td>1</td>
<td>8</td>
<td>22</td>
<td>22</td>
<td>91</td>
<td>20</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>CoreModule</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>21</td>
<td>2</td>
<td>59</td>
<td>45</td>
<td>41</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>frmChart</td>
<td>1</td>
<td>29</td>
<td>32</td>
<td>325</td>
<td>55</td>
<td>60</td>
<td>91</td>
<td>64</td>
<td>0.94</td>
<td>0.97</td>
</tr>
<tr>
<td>frmMain</td>
<td>1</td>
<td>94</td>
<td>32</td>
<td>1206</td>
<td>148</td>
<td>53</td>
<td>117</td>
<td>90</td>
<td>0.96</td>
<td>0.97</td>
</tr>
<tr>
<td>Inquiry</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>93</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>InquiryKeyword</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>93</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>KeywordType</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>94</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>PdfCollection</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>94</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>PdfDocument</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>93</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Program</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>71</td>
<td>14</td>
<td>8</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Quantifier</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>93</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>QuantifierList</td>
<td>1</td>
<td>4</td>
<td>11</td>
<td>21</td>
<td>15</td>
<td>80</td>
<td>13</td>
<td>7</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Quantifiers*</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>100</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Repository</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>93</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Revision</td>
<td>1</td>
<td>6</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>93</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Synchronise</td>
<td>8</td>
<td>10</td>
<td>26</td>
<td>165</td>
<td>50</td>
<td>75</td>
<td>129</td>
<td>52</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AbDocument</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>83</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IRepoistory</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>100</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ISvnService*</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>100</td>
<td>5</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PdImporter</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>31</td>
<td>6</td>
<td>60</td>
<td>20</td>
<td>13</td>
<td>0.33</td>
<td>0.5</td>
</tr>
<tr>
<td>SvnImporter</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>45</td>
<td>15</td>
<td>61</td>
<td>29</td>
<td>18</td>
<td>0.5</td>
<td>0.62</td>
</tr>
<tr>
<td>SvnService</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>41</td>
<td>12</td>
<td>69</td>
<td>34</td>
<td>22</td>
<td>0.63</td>
<td>0.73</td>
</tr>
<tr>
<td>TxtImporter</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>22</td>
<td>6</td>
<td>66</td>
<td>15</td>
<td>8</td>
<td>0.33</td>
<td>0.5</td>
</tr>
<tr>
<td>WordImporter</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>19</td>
<td>4</td>
<td>62</td>
<td>15</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Analyse</td>
<td>4</td>
<td>15</td>
<td>18</td>
<td>99</td>
<td>36</td>
<td>76</td>
<td>83</td>
<td>34</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DocumentAnalyzer</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>25</td>
<td>16</td>
<td>62</td>
<td>28</td>
<td>18</td>
<td>0.33</td>
<td>0.5</td>
</tr>
<tr>
<td>IDocumentAnalyzer*</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>100</td>
<td>3</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IRevisionAnalyzer*</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>100</td>
<td>3</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RevisionAnalyzer</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>74</td>
<td>18</td>
<td>43</td>
<td>49</td>
<td>29</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Export</td>
<td>5</td>
<td>13</td>
<td>28</td>
<td>79</td>
<td>51</td>
<td>87</td>
<td>49</td>
<td>20</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ChartData*</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>100</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ChartQuantifier</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>29</td>
<td>14</td>
<td>72</td>
<td>15</td>
<td>8</td>
<td>0.67</td>
<td>0.8</td>
</tr>
<tr>
<td>ChartType</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>2</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CsvExport&lt;T&gt;</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>40</td>
<td>22</td>
<td>68</td>
<td>22</td>
<td>14</td>
<td>0.57</td>
<td>0.67</td>
</tr>
</tbody>
</table>

Continued on next page
### Table A.2 – continued from previous page

<table>
<thead>
<tr>
<th>Name</th>
<th>Typ.</th>
<th>Fie.</th>
<th>Meth.</th>
<th>LOC</th>
<th>CC</th>
<th>MI</th>
<th>Coup.</th>
<th>UCoup.</th>
<th>LC2</th>
<th>LC3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExportChartData</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>9</td>
<td>94</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DataRepository</td>
<td>17</td>
<td>8</td>
<td>105</td>
<td>405</td>
<td>129</td>
<td>84</td>
<td>302</td>
<td>64</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ChangePathRepository</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>23</td>
<td>7</td>
<td>70</td>
<td>31</td>
<td>17</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CollectionRepository</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>47</td>
<td>9</td>
<td>67</td>
<td>31</td>
<td>17</td>
<td>0.43</td>
<td>0.5</td>
</tr>
<tr>
<td>DocumentRepository</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>63</td>
<td>12</td>
<td>65</td>
<td>37</td>
<td>22</td>
<td>0.5</td>
<td>0.57</td>
</tr>
<tr>
<td>IChangePath</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>100</td>
<td>4</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IChangePath*</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>100</td>
<td>3</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ICodiscloseDIContext*</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>100</td>
<td>5</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ICollectionRepository*</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>100</td>
<td>5</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IDocumentRepository*</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>100</td>
<td>4</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>InquiryRepository</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>51</td>
<td>9</td>
<td>66</td>
<td>30</td>
<td>16</td>
<td>0.29</td>
<td>0.33</td>
</tr>
<tr>
<td>InquireRepository*</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>100</td>
<td>4</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IKeywordRepository*</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>100</td>
<td>5</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IKeywordRepository*</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>100</td>
<td>5</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IKeywordTypeRepository*</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>100</td>
<td>5</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ILocalRepository*</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>100</td>
<td>5</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>InquiryRepository</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>51</td>
<td>9</td>
<td>66</td>
<td>30</td>
<td>16</td>
<td>0.29</td>
<td>0.33</td>
</tr>
<tr>
<td>IKeywordRepository*</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>100</td>
<td>4</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IRevisionRepository*</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>100</td>
<td>4</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>KeywordRepository</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>76</td>
<td>12</td>
<td>62</td>
<td>36</td>
<td>21</td>
<td>0.25</td>
<td>0.29</td>
</tr>
<tr>
<td>KeywordTypeRepository</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>45</td>
<td>9</td>
<td>66</td>
<td>31</td>
<td>16</td>
<td>0.33</td>
<td>0.4</td>
</tr>
<tr>
<td>LocalRepository</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>47</td>
<td>9</td>
<td>68</td>
<td>31</td>
<td>17</td>
<td>0.43</td>
<td>0.5</td>
</tr>
<tr>
<td>RevisionRepository</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>53</td>
<td>11</td>
<td>65</td>
<td>35</td>
<td>20</td>
<td>0.43</td>
<td>0.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Service</th>
<th>8</th>
<th>8</th>
<th>88</th>
<th>101</th>
<th>90</th>
<th>92</th>
<th>71</th>
<th>26</th>
<th>-</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>CollectionService</td>
<td>1</td>
<td>2</td>
<td>13</td>
<td>27</td>
<td>13</td>
<td>83</td>
<td>11</td>
<td>6</td>
<td>0.46</td>
<td>0.5</td>
</tr>
<tr>
<td>ICollectionService*</td>
<td>1</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>12</td>
<td>100</td>
<td>6</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>InquryService*</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>100</td>
<td>4</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IKeywordService*</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>11</td>
<td>100</td>
<td>6</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>InquiryService</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>84</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IRepositoryService*</td>
<td>1</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>15</td>
<td>100</td>
<td>7</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>KeywordService</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>27</td>
<td>12</td>
<td>83</td>
<td>11</td>
<td>6</td>
<td>0.46</td>
<td>0.5</td>
</tr>
<tr>
<td>RepositoryService</td>
<td>1</td>
<td>3</td>
<td>16</td>
<td>37</td>
<td>18</td>
<td>82</td>
<td>18</td>
<td>13</td>
<td>0.62</td>
<td>0.67</td>
</tr>
</tbody>
</table>

* abstract class/interface
Appendix B

Code and Branch Coverage

\[ \text{CNCov} = \text{Code not covered}; \quad \text{CCov} = \text{Code covered}; \]
\[ \text{TCode} = \text{Total code}; \quad \%\text{CCov} = \%\text{Code coverage}; \]
\[ \text{BNCov} = \text{Branch not covered}; \quad \text{BCov} = \text{Branch covered}; \]
\[ \text{TBr} = \text{Total branch}; \quad \%\text{BCov} = \%\text{Branch coverage}. \]

Table B.1: Codisclose Code and Branch Coverage from CoUnitTest

<table>
<thead>
<tr>
<th>Name</th>
<th>CNCov</th>
<th>CCov</th>
<th>TCode</th>
<th>%CCov</th>
<th>BNCov</th>
<th>Bcov</th>
<th>TBr</th>
<th>%BCov</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td>53</td>
<td>2696</td>
<td>2749</td>
<td>98.07%</td>
<td>20</td>
<td>870</td>
<td>890</td>
<td>97.75%</td>
</tr>
<tr>
<td>Codisclose</td>
<td>41</td>
<td>1958</td>
<td>1999</td>
<td>97.95%</td>
<td>18</td>
<td>518</td>
<td>536</td>
<td>96.64%</td>
</tr>
<tr>
<td>frmChart</td>
<td>0</td>
<td>475</td>
<td>475</td>
<td>100%</td>
<td>2</td>
<td>143</td>
<td>145</td>
<td>98.62%</td>
</tr>
<tr>
<td>frmMain</td>
<td>36</td>
<td>1445</td>
<td>1481</td>
<td>97.57%</td>
<td>15</td>
<td>231</td>
<td>246</td>
<td>93.90%</td>
</tr>
<tr>
<td>Program</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>Synchronise</td>
<td>0</td>
<td>168</td>
<td>168</td>
<td>100%</td>
<td>0</td>
<td>62</td>
<td>62</td>
<td>100%</td>
</tr>
<tr>
<td>Analyse</td>
<td>0</td>
<td>90</td>
<td>90</td>
<td>100%</td>
<td>0</td>
<td>80</td>
<td>80</td>
<td>100%</td>
</tr>
<tr>
<td>Export</td>
<td>0</td>
<td>64</td>
<td>64</td>
<td>100%</td>
<td>0</td>
<td>64</td>
<td>64</td>
<td>100%</td>
</tr>
<tr>
<td>CRUD</td>
<td>12</td>
<td>416</td>
<td>428</td>
<td>97.20%</td>
<td>2</td>
<td>146</td>
<td>148</td>
<td>98.65%</td>
</tr>
<tr>
<td>DataAccess</td>
<td>12</td>
<td>416</td>
<td>428</td>
<td>97.20%</td>
<td>2</td>
<td>146</td>
<td>148</td>
<td>98.65%</td>
</tr>
</tbody>
</table>

Note: Classes with full coverage are excluded.
### Table B.2: CodisposeDI Code and Branch Coverage from DIUnitTest

<table>
<thead>
<tr>
<th>Name</th>
<th>CNCov</th>
<th>CCov</th>
<th>TCode</th>
<th>%CCov</th>
<th>BNCov</th>
<th>BCov</th>
<th>TBr</th>
<th>%BCov</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td>51</td>
<td>2733</td>
<td>2784</td>
<td>98.17%</td>
<td>22</td>
<td>936</td>
<td>958</td>
<td>97.70%</td>
</tr>
<tr>
<td>CodisposeDI</td>
<td>51</td>
<td>1882</td>
<td>1933</td>
<td>97.36%</td>
<td>20</td>
<td>491</td>
<td>511</td>
<td>96.09%</td>
</tr>
<tr>
<td>frmChart</td>
<td>0</td>
<td>397</td>
<td>397</td>
<td>100%</td>
<td>2</td>
<td>99</td>
<td>101</td>
<td>98.02%</td>
</tr>
<tr>
<td>frmMain</td>
<td>44</td>
<td>1409</td>
<td>1453</td>
<td>96.97%</td>
<td>17</td>
<td>243</td>
<td>260</td>
<td>93.46%</td>
</tr>
<tr>
<td>Program</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>Synchronise</td>
<td>0</td>
<td>226</td>
<td>226</td>
<td>100%</td>
<td>2</td>
<td>83</td>
<td>85</td>
<td>97.65%</td>
</tr>
<tr>
<td>SvnImporter</td>
<td>0</td>
<td>62</td>
<td>62</td>
<td>100%</td>
<td>2</td>
<td>31</td>
<td>33</td>
<td>93.94%</td>
</tr>
<tr>
<td>Analyse</td>
<td>0</td>
<td>72</td>
<td>72</td>
<td>100%</td>
<td>0</td>
<td>63</td>
<td>63</td>
<td>100%</td>
</tr>
<tr>
<td>Export</td>
<td>0</td>
<td>108</td>
<td>108</td>
<td>100%</td>
<td>0</td>
<td>90</td>
<td>90</td>
<td>100%</td>
</tr>
<tr>
<td>DataRepository</td>
<td>0</td>
<td>292</td>
<td>292</td>
<td>100%</td>
<td>0</td>
<td>156</td>
<td>156</td>
<td>100%</td>
</tr>
<tr>
<td>Service</td>
<td>0</td>
<td>153</td>
<td>153</td>
<td>100%</td>
<td>0</td>
<td>53</td>
<td>53</td>
<td>100%</td>
</tr>
</tbody>
</table>

*Note: Classes with full coverage are excluded.*

### Table B.3: CodisposeDI Code and Branch Coverage from DIMockTest

<table>
<thead>
<tr>
<th>Name</th>
<th>CNCov</th>
<th>CCov</th>
<th>TCode</th>
<th>%CCov</th>
<th>BNCov</th>
<th>BCov</th>
<th>TBr</th>
<th>%BCov</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td>204</td>
<td>2580</td>
<td>2784</td>
<td>92.67%</td>
<td>93</td>
<td>865</td>
<td>958</td>
<td>90.29%</td>
</tr>
<tr>
<td>CodisposeDI</td>
<td>51</td>
<td>1882</td>
<td>1933</td>
<td>97.36%</td>
<td>38</td>
<td>473</td>
<td>511</td>
<td>92.56%</td>
</tr>
<tr>
<td>ChangePath</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6</td>
<td>17</td>
<td>23</td>
<td>73.91%</td>
</tr>
<tr>
<td>CodisposeDIcontext</td>
<td>19</td>
<td>0</td>
<td>19</td>
<td>0%</td>
<td>22</td>
<td>0</td>
<td>22</td>
<td>0%</td>
</tr>
<tr>
<td>CoreModule</td>
<td>22</td>
<td>0</td>
<td>22</td>
<td>0%</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>frmChart</td>
<td>0</td>
<td>397</td>
<td>397</td>
<td>100%</td>
<td>2</td>
<td>99</td>
<td>101</td>
<td>98.02%</td>
</tr>
<tr>
<td>frmMain</td>
<td>3</td>
<td>1450</td>
<td>1453</td>
<td>99.79%</td>
<td>2</td>
<td>258</td>
<td>260</td>
<td>99.23%</td>
</tr>
<tr>
<td>KeywordType</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>85.71%</td>
</tr>
<tr>
<td>Program</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>Revision</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>11</td>
<td>13</td>
<td>84.62%</td>
</tr>
<tr>
<td>Synchronise</td>
<td>0</td>
<td>226</td>
<td>226</td>
<td>100%</td>
<td>2</td>
<td>83</td>
<td>85</td>
<td>97.65%</td>
</tr>
<tr>
<td>SvnImporter</td>
<td>0</td>
<td>62</td>
<td>62</td>
<td>100%</td>
<td>2</td>
<td>31</td>
<td>33</td>
<td>93.94%</td>
</tr>
<tr>
<td>Analyse</td>
<td>0</td>
<td>72</td>
<td>72</td>
<td>100%</td>
<td>0</td>
<td>63</td>
<td>63</td>
<td>100%</td>
</tr>
<tr>
<td>Export</td>
<td>0</td>
<td>108</td>
<td>108</td>
<td>100%</td>
<td>0</td>
<td>90</td>
<td>90</td>
<td>100%</td>
</tr>
<tr>
<td>DataRepository</td>
<td>0</td>
<td>292</td>
<td>292</td>
<td>100%</td>
<td>0</td>
<td>156</td>
<td>156</td>
<td>100%</td>
</tr>
<tr>
<td>Service</td>
<td>153</td>
<td>0</td>
<td>153</td>
<td>0%</td>
<td>53</td>
<td>0</td>
<td>53</td>
<td>0%</td>
</tr>
<tr>
<td>CollectionService</td>
<td>41</td>
<td>0</td>
<td>41</td>
<td>0%</td>
<td>13</td>
<td>0</td>
<td>13</td>
<td>0%</td>
</tr>
</tbody>
</table>

*Continued on next page*
### Table B.3 – continued from previous page

<table>
<thead>
<tr>
<th>Name</th>
<th>CNCov</th>
<th>CCov</th>
<th>TCode</th>
<th>%CCov</th>
<th>BNCov</th>
<th>BCov</th>
<th>TBr</th>
<th>%BCov</th>
</tr>
</thead>
<tbody>
<tr>
<td>InquiryService</td>
<td>16</td>
<td>0</td>
<td>16</td>
<td>0%</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>0%</td>
</tr>
<tr>
<td>KeywordService</td>
<td>40</td>
<td>0</td>
<td>40</td>
<td>0%</td>
<td>12</td>
<td>0</td>
<td>12</td>
<td>0%</td>
</tr>
<tr>
<td>RepositoryService</td>
<td>56</td>
<td>0</td>
<td>56</td>
<td>0%</td>
<td>23</td>
<td>0</td>
<td>23</td>
<td>0%</td>
</tr>
</tbody>
</table>

Note: Classes with full coverage are excluded.

### Table B.4: CodiscloseDI Code and Branch Coverage from DIMockUnitTest

<table>
<thead>
<tr>
<th>Name</th>
<th>CNCov</th>
<th>CCov</th>
<th>TCode</th>
<th>%CCov</th>
<th>BNCov</th>
<th>BCov</th>
<th>TBr</th>
<th>%BCov</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td>10</td>
<td>2774</td>
<td>2784</td>
<td>99.64%</td>
<td>7</td>
<td>951</td>
<td>958</td>
<td>99.27%</td>
</tr>
<tr>
<td>CodiscloseDI</td>
<td>10</td>
<td>1923</td>
<td>1933</td>
<td>99.48%</td>
<td>5</td>
<td>506</td>
<td>511</td>
<td>99.02%</td>
</tr>
<tr>
<td>frmChart</td>
<td>0</td>
<td>397</td>
<td>397</td>
<td>100%</td>
<td>2</td>
<td>99</td>
<td>101</td>
<td>98.02%</td>
</tr>
<tr>
<td>frmMain</td>
<td>3</td>
<td>1450</td>
<td>1453</td>
<td>99.79%</td>
<td>2</td>
<td>258</td>
<td>260</td>
<td>99.23%</td>
</tr>
<tr>
<td>Program</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>Synchronise</td>
<td>0</td>
<td>226</td>
<td>226</td>
<td>100%</td>
<td>2</td>
<td>83</td>
<td>85</td>
<td>97.65%</td>
</tr>
<tr>
<td>SvnImporter</td>
<td>0</td>
<td>62</td>
<td>62</td>
<td>95.16%</td>
<td>2</td>
<td>31</td>
<td>33</td>
<td>93.94%</td>
</tr>
<tr>
<td>Analyse</td>
<td>0</td>
<td>72</td>
<td>72</td>
<td>100%</td>
<td>0</td>
<td>63</td>
<td>63</td>
<td>100%</td>
</tr>
<tr>
<td>Export</td>
<td>0</td>
<td>108</td>
<td>108</td>
<td>100%</td>
<td>0</td>
<td>90</td>
<td>90</td>
<td>100%</td>
</tr>
<tr>
<td>DataRepository</td>
<td>0</td>
<td>292</td>
<td>292</td>
<td>100%</td>
<td>0</td>
<td>156</td>
<td>156</td>
<td>100%</td>
</tr>
<tr>
<td>Service</td>
<td>0</td>
<td>153</td>
<td>153</td>
<td>100%</td>
<td>0</td>
<td>53</td>
<td>53</td>
<td>100%</td>
</tr>
</tbody>
</table>

Note: Classes with full coverage are excluded.
## Appendix C

### Test Suite Metrics

<table>
<thead>
<tr>
<th>Name</th>
<th>CoUT</th>
<th>DIUT</th>
<th>DIMT</th>
<th>DIMUT</th>
<th>CoUT</th>
<th>DIUT</th>
<th>DIMT</th>
<th>DIMUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td>1324</td>
<td>1351</td>
<td>2286</td>
<td>1718</td>
<td>207</td>
<td>237</td>
<td>259</td>
<td>264</td>
</tr>
<tr>
<td>MainFormTest</td>
<td>470</td>
<td>492</td>
<td>789</td>
<td>792</td>
<td>76</td>
<td>73</td>
<td>90</td>
<td>92</td>
</tr>
<tr>
<td>SynchroniseTest</td>
<td>134</td>
<td>155</td>
<td>242</td>
<td>214</td>
<td>17</td>
<td>30</td>
<td>37</td>
<td>36</td>
</tr>
<tr>
<td>AnalyseTest</td>
<td>165</td>
<td>133</td>
<td>285</td>
<td>141</td>
<td>12</td>
<td>10</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>ChartFormTest</td>
<td>220</td>
<td>195</td>
<td>195</td>
<td>195</td>
<td>29</td>
<td>29</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>ExportTest</td>
<td>49</td>
<td>87</td>
<td>87</td>
<td>87</td>
<td>12</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>CRUDTest/DataRepositoryTest</td>
<td>286</td>
<td>289</td>
<td>688</td>
<td>289</td>
<td>61</td>
<td>78</td>
<td>74</td>
<td>78</td>
</tr>
</tbody>
</table>

CoUT = CodiscloseUnitTest  
DIUT = CodiscloseDIUnitTest  
DIMT = CodiscloseDIMockTest  
DIMUT = CodiscloseDIMockAndUnitTest
Bibliography


Beck, Kent. 2001. Aim, fire. *IEEE SOFTWARE.*

Bender, James, & McWherter, Jeff. 2011. *Professional test driven development with C#: developing real world applications with TDD.* Wrox Press Ltd.


Freeman, Elisabeth. 2004. *Head first design patterns.* " O’Reilly Media, Inc.”.


Seemann, Mark. 2007. Unit testing-exploring the continuum of test doubles. MSDN Magazine-Louisville, 87–100.


