Tero Haukilehto

Isolated unit tests in .Net

Thesis

Spring 2013

School of Technology

Information Technology

SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Thesis Abstract

Faculty: School of Technology

Degree programme: Information Technology

Specialisation: Data Network Technology

Author: Tero Haukilehto

Title of thesis: Isolated unit tests in .Net

Supervisor: Hilkka Niemelä

Year: 2013 Pages: 60 Number of appendices: 1

In this thesis isolation in unit testing is studied to get a precise picture of the isolation frameworks available for .Net environment. At the beginning testing is discussed in theory with the benefits and the problems it may have been linked with. The theory includes software development in general in connection with testing.

Theory of isolation is also described before the actual isolation frameworks are represented. Common frameworks are described in more detail and comparable information and coding examples are shown. Because the purpose of this thesis is to report of usable isolation frameworks in unit testing, the focus is on doing unit testing in practice. As a result an isolation framework can be recommended for the use of ABB's software development.

Keywords: unit testing, software development, isolation frameworks

SEINÄJOEN AMMATTIKORKEAKOULU

Opinnäytetyön tiivistelmä

Koulutusyksikkö: Tekniikan yksikkö

Koulutusohjelma: Tietotekniikka

Suuntautumisvaihtoehto: Tietoverkkotekniikka

Tekijä: Tero Haukilehto

Työn nimi: Isolated unit tests in .Net

Ohjaaja: Hilkka Niemelä

Vuosi: 2013 Sivumäärä: 60 Liitteiden lukumäärä: 1

Tässä työssä tutkitaan ohjelmistotestausta sekä erityisesti yksikkötestausta ja siinä käytettäviä eristystekniikoita. Työn tarkoituksena on selvittää, mikä .Net-eristystekniikoista sopii ABB:n käyttöön.

Opinnäytetyössä käydään läpi testausta teoriassa, käsitellään testauksen tärkeyttä ja syitä, jotka vähentävät sitä. Testausta lähestytään ohjelmistokehityksessä käytettävien mallien avulla, joihin sen eri strategiat ja testaustasot ovat yhteydessä. Testauksen yleisestä teoriasta päästään itse eristystekniikoihin, joilla testausta voidaan helpottaa ja nopeuttaa.

Eristystekniikoita tutkitaan niiden käyttöasteen mukaan. Valittujen tekniikoiden ominaisuuksia vertaillaan ja niillä toteutetaan esimerkkitestejä. Esimerkkien pohjalta muodostetaan kuva tekniikoiden käytettävyydestä ja lopulta pystytään suosittelemaan tekniikkaa ABB:n sovelluskehityksen käyttöön.

Asiasanat: Sovelluskehitys, yksikkötestaus, eristystekniikat.

TABLE OF CONTENTS

Thesis Abstract....................1
Opinnäytetyön tiivistelmä....................2
TABLE OF CONTENTS....................3
Tables and figures....................5
Abbreviations and terms....................7
1 INTRODUCTION....................9
1.1 Background....................9
1.2 Objectives....................9
1.3 Structure....................10
1.4 Company introduction....................10
2 SOFTWARE TESTING....................12
2.1 Importance of testing....................12
2.2 Reasons for omitting tests....................13
2.3 V-model....................14
2.4 Scrum....................16
2.5 Testing strategies....................18
2.5.1 Functional (Black-Box) testing.....................18
2.5.2 Structural (White-Box) testing....................19
2.5.3 Gray-Box testing....................20
2.6 Testing levels....................21
2.6.1 Unit (module) testing....................21
2.6.2 Integration testing....................23
2.6.3 System testing....................24
2.6.4 Acceptance testing....................25
2.7 Isolation....................26
3 ISOLATION FRAMEWORKS....................28
3.1 Isolation frameworks used today....................28
3.2 Mostly used isolation frameworks....................29
3.3 Moles framework....................30
3.3.1 Installing Moles framework in Microsoft Visual Studio 2010....................31

3.3.2 Mole code example 32
3.3.3 Stub coding example 39
3.4 Fakes Framework 44
3.4.1 Main differences between Fakes and Moles frameworks 45
3.4.2 Fakes example 47
3.5 Moq Framework 48
3.6 FakeItEasy 50
3.6.1 FakeItEasy installation via NuGet 51
3.6.2 FakeItEasy example 53
4 RESULTS 54
5 SUMMARY 56
BIBLIOGRAPHY 57
APPENDICES 61

Tables and figures

Figure 1. V-model in software testing (Haikala & Mikkonen 2011). 14
Figure 2. Scrum Framework. 16
Figure 3. Black-Box testing. (Burnstein 2003, 65). 19
Figure 4. Levels of testing (Burnstein 203, 134). 21
Figure 5. Module hierarchy (Burnstein 2003, 154). 24
Figure 6. Test doubles (Meszaros 2009). 27
Figure 7. Isolation frameworks used in .Net 2010 (Osherove 2010). 28
Figure 8. Isolation frameworks used in .Net 2012. (Osherove 2012). 29
Figure 9. Tools bar and Extension Manager. 31
Figure 10. Extension Manager. 32
Figure 11. Mole example Library event class. 33
Figure 12. Mole example Sql layer class. 34
Figure 13. Solution explorer after creating a unit test project. 35
Figure 14. Adding Reference to the project. 35
Figure 15. Adding Moles Assembly. 36
Figure 16. Solution Explorer after adding the assembly. 36
Figure 17. Moles example .moles xml-file. 36
Figure 18. Fake types in Class View. 37
Figure 19. Mole example unit test class. 38
Figure 20. Run All Tests in Solution. 39
Figure 21. Mole example Test Results. 39
Figure 22. Stub example Sql Layer class. 40
Figure 23. Stub example Library event class. 41
Figure 24. Stub example Save event class. 42
Figure 25. Solution Explorer after adding moles assembly and reference. 43
Figure 26. Stub example unit test class. 44
Figure 27. Fakes example unit test class. 48
Figure 28. MOQ example unit test class. 50
Figure 29. Nuget Package Manager Console. 52
Figure 30. Installing FakeItEasy package. 52
Figure 31. FakeItEasy unit test class. 53

Table 1. Main differences between Fakes and Moles frameworks........................ 46
Table 2. Moq features compared to Moles features.. 49

Abbreviations and terms

.Net Framework	Software component library that consists of the common language runtime and the .NET class library (MSDN).
Acceptance test	A test that determines if the system meets its acceptance criteria and can be accepted by the customer (Farrell-Vinay 2008, 470).
Black-box testing	Functional testing without using any knowledge of the system's construction (Farrell-Vinay 2008, 473).
C#	Object-oriented programming language for building applications in .Net Framework (MSDN 2012).
Configurable test double	Reusable test double with configurable values to be returned or expected at runtime (Meszaros, 2009).
DOC	A component in a software that has dependencies (Meszaros, 2009).
Dummy object	A test double that does not have behavior with no inputs or output handling (Meszaros, 2009).
Fake object	A test double type to run unrunnable tests by using the indirect outputs (Meszaros, 2009).
Integration testing	Test level where software elements, hardware elements or both are combined and tested until the complete system has been integrated (Farrell-Vinay 2008, 489).
Isolation framework	A framework used in unit tests to break dependencies (Osherove, 2009).
Mock	A test double that verifies indirect outputs of SUT against expectations (Meszaros, 2009).

Mole	Test double type used in Fakes framework to isolate third-party components without testable programming interface (Msdn 2012c).
Shim	Test double type used in Fakes framework to isolate third-party components without testable programming interface (Msdn 2012c). Shim Vs. Mole type in Moles framework.
Stub	A test double that injects indirect inputs into the system but ignores the outputs (Meszaros, 2009).
SUT	System, subsystem or component being tested, aka System Under Test (Farrell-Vinay 2008, 504).
System test	Test level to determine if the system meets its specified requirements (Farrell-Vinay 2008, 504).
Temporary test stub	A test double for DOC that is not yet available, often used with test-driven development (Meszaros, 2009).
Test case	A set of inputs, execution conditions and a pass/fail criterion (Pezze & Young 2008, 153).
Test double	An object used to replace the real DOC. Generic name for family (Meszaros, 2009).
Test spy	A test double similar to Mock, but captures the outputs for later verification (Meszaros, 2009).
Unit	Smallest piece of testable code (Pezze & Young 2008, 282).
White-box testing	Structural testing where knowledge of the SUT's internal logic is used (Farrell-Vinay 2008, 503).

1 INTRODUCTION

1.1 Background

Software Testing is a current topic. It should have been like that already for many years, but unfortunately it does not sound fancy and valuable. In fact, sometimes the testing can be even decreased for balancing the system development budget or getting the system to the market sooner. However, saving in testing can be dangerous and at least very risky. (Farrell-Vinay 2008, 1-5.)

Only by testing we can ensure that the system has the value it should have. This is done with thoroughly testing on each testing level starting from basic unit testing to final acceptance testing before shipping the product to the customer (Burnstein 2003, 132-137). There are many ways to put testing into practice, but the hierarchy is usually the same where each phase in testing is connected to the used testing strategy and to the steps in the current system development methodology.

The first phase in testing hierarchy is unit testing, where the smallest testable pieces are tested. In order to test these parts of the program individually, we have to break their dependencies to other parts. This can be done manually, but using isolation frameworks has several advantages from making the developing faster, and less error-prone to reducing the need to duplicate code writing. Isolated unit tests, in general, are much easier and time saving to write and use, and more importantly they can even encourage coders to write tests. (Osherove 2009, 99-102.)

1.2 Objectives

The main objective of this thesis is to look into today's most used isolation frameworks for .Net environment. The frameworks are compared and their basic use is shown with coding examples. Before diving into the world of isolation, testing is discussed in theory starting with basic testing principles and strategies.

The main goal is to find out which one of the isolation frameworks available is suitable for ABB's use in unit testing. The contract with the company also included actual unit testing for a new ABB software component using the isolation framework that this study would prove useful. During this thesis unit testing in practice will be learned and, in addition, the objective is to study and learn how the testing is connected into software development and it's methodologies.

1.3 Structure

The testing in theory is discussed in the second chapter starting with the reasons why we should test the software and how important it really is. We also take a look into negative attitudes towards software testing and how to overcome them. Then common software development methodologies are represented in connection with the testing.

Chapter 2.5 handles the strategies of testing from functional and structural methods to the combination of the both. The next chapter concentrates on the levels of testing and shows how they are connected to the software development process. Before the isolation frameworks are introduced a brief explanation about isolation itself is given.

Chapter three begins with the introduce of common isolation frameworks. Then four of the most interesting frameworks are taken under a closer look and used in coding examples. Chapter 4 shows the results of comparing the frameworks and the final chapter summarizes the whole thesis.

1.4 Company introduction

ABB is a multinational Swiss-Swedish industrial corporation. ABB's main industries are automation and power technologies. The corporation was founded when the Swiss BBC Brown Boveri & Cie (founded 1891) and Swedish ASEA (founded 1883) merged in 1988. The company's business is divided into five divisions which are Power Products, Power Systems, Low Voltage Products, Discrete Automation and Motion, and Process Automation. (ABB 2012a.)

ABB has about 145 000 employees in over 100 countries and it is one of the largest engineering companies in the world. In Finland over 7000 people are working for ABB. The reported global revenue was about 40 Billion dollars and the reported revenue in Finland was 2.6 Billion dollars in 2011. (ABB 2012b.)

In Finland the company has factories in Vaasa, Helsinki and Porvoo and it is the leading company in industrial maintenance area. ABB spent about 160 million Euros into product development in Finland in 2011 and it is also one of the biggest industrial employers in the country. (ABB 2012c.)

This thesis is done especially for the use of Medium voltage products unit in Vaasa and it's co-unit in India. The unit is responsible for the development, sales and marketing of protection and control equipment for Medium voltage electrical transmission that creates the base for smart grid.

2 SOFTWARE TESTING

2.1 Importance of testing

One way to approach software testing is to find an answer to a question: Why do we have to test? According to Pezzè & Young (2008, 15), the reason behind testing is either to estimate software quality or find defects, both aiming to improve the software. Craig & Jaskiel (2002, 536), described that in general software testing is seen as planned searching of errors by running the whole program or a part of it.

It is estimated that in a new program there are couple errors in every hundred lines of code. The number of bugs in long used applications can still be even one bug per thousand lines of code. Yet, it may require too much resourses to find and fix all errors. About 5% of the bugs can be even invisible because they do not cause malfunctions or they are fixed by another part of the program. In fact, with testing we can show that the program contains bugs, but we can't show it does not. (Haikala & Mikkonen 2011, 205-206.)

One thing is certain - software errors occur and they are expensive. The errors should be noticed and fixed at the earliest possible stage when they still are easier and cheaper to fix, as the costs of errors rise while the development process goes on. Typically, over half of the resources of a software project are spent on finding the errors, testing and fixing the program. (Haikala & Mikkonen 2011, 205-206.)

According to a report of The National Institute of Standards and Technology in 2002 (NIST 2002), software errors cost the U.S economy approximately 59.5 billion USD every year. The report estimated that over 50% of the errors are currently found after the later development process or in after sale use. By improving the testing and earlier identifications of the errors the save could be up to 22.2 billion USD annually.

For comparison, the whole software testing market in Finland is estimated to be worth 100-200 million Euros (130 - 270 million USD) (Lehto, 2013). The numbers may seem unrealistic, but if we assume that an average coder spends half of their coding time finding and fixing errors, the global cost of bugs may rise up to 316 billion dollars per year according to a recent research of a software firm Undo Ltd (University of Cambridge, 2013).

2.2 Reasons for omitting tests

Still, despite all the facts and numbers that show that testing is a vital part of software development and even as expensive as all the rest of the project, it is often given much less value. Peter Farrell-Vinay (2008, 2-5), lists several reasons why testing is often misunderstood, mislead, done improperly and eventually failed.

The major problem is that testing has a bad taste, it costs a lot and it can produce embarrassing results. The testing itself is a never ending process - some even may see this as a reason for not testing - but a software without testing is like a school system without tests. It might work (students might have learnt), who knows?

Testing can be seen as a time wasting activity that prevents the valuable (when it still has a value) product from making profit. The current economical situation can also tempt software companies to reduce the testing resources (Lehto, 2013). But the reality is that the value that testing can provide for the system can even turn the product's value from negative to positive. A system with bugs can be useless and worth nothing, no matter how much money has already been spent. Even patches will not save the system if it has already been claimed unusable by the public.

If the software development project does not have money, time or other resources for testing, then there is a lack of planning in the first place. A carefully planned project includes testing and resources for it. That way testing is also a lot cheaper and easier when it has been secured from the beginning.

A successful software project requires testing and even more important requirements specification to test against. Requirements specification is also crucial for developing, technical writers, marketing and for the management. In practice, the requirement specification cannot be perfect (otherwise we should have the system itself), but it has to meet the needs of the previous groups. Furthermore it has to be changeable, because in software development changes happen.

2.3 V-model

The relationships between the steps in software development and software testing are often demonstrated with an abstract V-model, the enhanced version of the classical waterfall model (Ghahrai 2008). The V-model connects each step in development process to associated phase of testing and vice versa, as seen in the figure 1 (Haikala & Mikkonen 2011, 206).

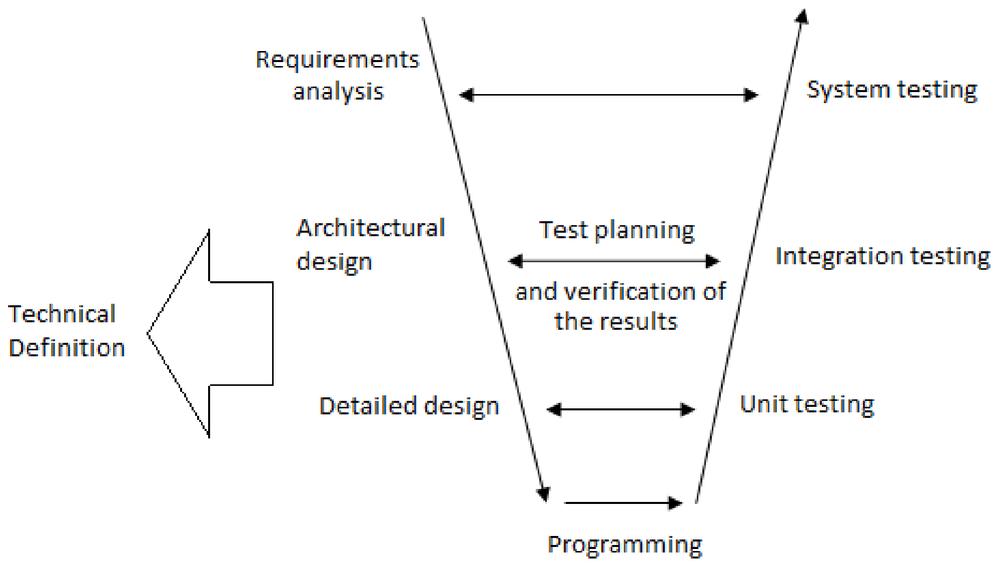


Figure 1. V-model in software testing (Haikala & Mikkonen 2011).

As Pezze & Young (2008, 15-18) describe the V-model is based on verification and validation. The verification stands for the coherency of design, specification and the code, whereas the validation checks if the system really meets its function. In practice, during every development step (left side) a corresponding testing plan is created and while testing (right side) the results are compared to the developing documents = Verification. Respectively, the final code is compared to the overall system specification = Validation.

In this theoretical model, only after one phase has been completed, checked and approved the next can be started. This ensures that the verification continues through all stages and each phase has been tested as planned. (Waterfall-model 2012.)

In that way errors can be noticed at early stages and the number of errors in final code can be reduced significantly. The V-model works best with small projects, and thanks to the model the chance to succeed is higher and the development process is much less time consuming than with the original waterfall model. But the V-model is not a trouble-free. (Ghahrai 2008.)

The V-model is already dead, says Ed Liversidge, the director of Harmonic Software Systems Ltd (Liversidge 2005). He accuses the model of misleading project managers to think that the forthcoming project is well understood and if the model is used the project is more likely to fail than succeed. Liversidge admits that the V-model has a number of good points like linking the phases and demanding the document writing but eventually it will not help.

The first reason for V-model to fail according to Mr. Liversidge is that it is simply too abstract and rigid to cover all situations and especially meet the changes that will happen in software development. The second reason is that doing unit tests separate from integration tests can be expensive and problematic in large projects due to unit test's possible need for a custom test harness. Liversidge warns software managers from leading into a false sense of security with the V-model because without flexible and problem solving engineers they would be using it too precisely and fail. Hereby the V-model can be used in software development as a directional guideline rather than a strict set of orders keeping in mind its advantages and disadvantages.

2.4 Scrum

Traditionally waterfall based methods are not the only used in software development. If one trend has attracted attention in software development lately, its Agile methodology. One of the most used agile methods is Scrum, which offers a model to lead a software development project. Unlike the waterfall based models like the V-model, Scrum has only three different core roles: Product owner, ScrumMaster and development team, whereas the V-model has at least five: descriptor, planner, coder, tester and project manager, all with certain tasks. (Poimala & Tolvanen, 2011.)

Scrum, in brief is rather a framework than complete methodology, focusing on dividing the project and maintaining the control of progressing. Agile development like Scrum divides the development progress into cycles. The most important cycle is a development cycle named Sprint, lasting from one week to two months, at which time the product should be basically complete. (Poimala & Tolvanen, 2011.) Figure 2 shows the basic framework of Scrum where the product is developed starting from product and sprint backlogs through sprints and daily scrums to potentially shippable product increment.

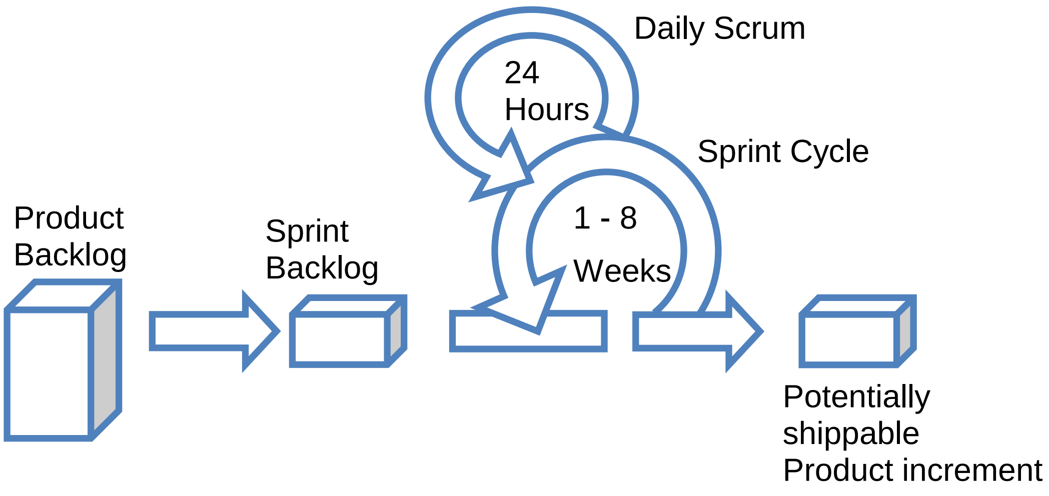


Figure 2. Scrum Framework.

During the sprint, the requirements cannot be changed and the team has full freedom to try achieve the goals of the sprint. The team itself can consist from workers with various job titles, each working on their best for the sprint, often crossing their preferred disciplines. While the product owner is responsible for the vision of the product, the ScrumMaster concentrates on helping the team to be their best and to keep up high performance. (Cohn, 2012.)

The main difference in testing with V-model and Scrum methodology is that in V-model testing is a phase, in Scrum it's not. During the sprint, the Scrum team works as a whole, test engineers included to achieve the current goal. This gives clearly different approach to testing than the traditional model, because after the sprint the feature should be ready, tested and no regressions should exist. (Tuomikoski, 2009.) In other words, the unit testing is even more important in agile methods like Scrum when the new cycle has to have the regression tests done in the previous cycle.

Testing in Scrum provides short feedback loop to the development when both developers and test engineers are working closely. It also enhances the communication and support during the development. Yet, short sprints set challenges to create enough code to be tested and in given time. (Tuomikoski, 2009.) Unfortunately, when the sprints can have even more intensive schedules than traditional development, the team may give up unit testing the code to reach the other sprint goals.

Agile methodologies have been criticized for not answering the question why the development project exists or how should be the development process led in the long run. Therefore, the project preparations should not have forgotten even with agile methods. (Poimala & Tolvanen, 2011.)

Both agile and waterfall based software development methods have their own ways to handle and describe the projects. Depending on current project other can be more suitable to use than another. Still, using whichever should not lead into a situation where testing is reduced because of limited time. If so, then the fault is somewhere else than in the methodologies.

2.5 Testing strategies

Despite the used software development methodology, in order to design accurate software test cases, a valid test approach is needed. Many techniques are used but often two major methods are mentioned: functional and structural (Burnstein 2003, 63-64). Both can be used with any unit or build but they offer different strengths, quality aspects and cons (Farrell-Vinay 2008, 18).

The V-model and the testing strategies are often linked to each other. As moving upwards with the model the testing usually changes from white-box to black-box testing. (Haikala & Mikkonen 2011, 209.) In Scrum the current test strategy is not as clear as with the V-model, but it is often depending on the current sprint goals. The sprint however might not include all the testing, so addition testing have to be planned outside the sprint. (Almeida, 2007.)

2.5.1 Functional (Black-Box) testing.

Functional testing, often referred as Black-Box testing, is an approach where tester can consider the software under test to be (in) a black box. The tester does not have knowledge of inner structure of the software, he only knows what it does. This enables the strategy to be used with any build from a single unit to complete system, because the approach is the same in every case and you cannot see what is inside the box (i.e. how it works). (Burnstein 2003, 63-64.)

The tests are written using the specification and tested against it to find errors where the system does not work as described. To be effective, the specification needs to be accurate and thorough (Farrell-Vinay 2008, 18). Also information from Input/Process/Output Diagram (IPO) or requirements specification can be used to describe the behavior or functionality of the system (Burnstein 2003, 63-64).

Common methods in Black-Box testing are Equivalence class partitioning and Boundary value analysis. In Equivalence class partitioning the input domain of the software is divided into equivalence classes. These classes are chosen by assumption that if one works with the software they all will. Or if one contains a defect, they all do. (Burnstein 2003, 65-72.) That also creates the weakness of the method, the solid-lookalike class can in fact consist of multiple classes (Haikala & Mikkonen 2011, 209).

If as well the edges of the equivalence classes are used, the method is called boundary value analysis. Adding the boundaries strengthens the basic equivalence class partitioning and increases potentially the possibility of finding errors. (Burnstein 2003, 72-73.) However, the expanded test cases are also more difficult to create (Haikala & Mikkonen 2011, 209).

In practice, the tester runs test with specified inputs and compares the output to expected values (see figure 3). This characterizes the black-box testing as functional, or specification-based method and it is also an effective method to check consistency of specifications. (Burnstein 2003, 64.)

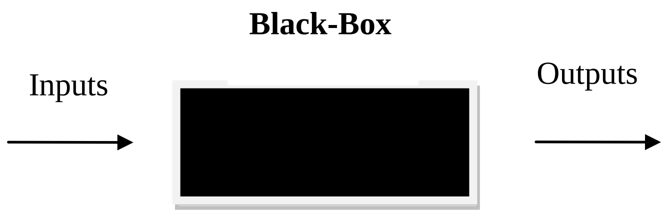


Figure 3. Black-Box testing (Burnstein 2003, 65).

2.5.2 Structural (White-Box) testing

As opposite to black-box testing, in white-box (or glass-box) testing, the tester has the knowledge of the software under test. In this strategy the tester aims to determine if all internal components in the software are functioning properly. The strategy has an exercising nature and it focuses to finding errors from the system by executing its structural elements. (Burnstein 2003, 64.)

Structural testing strategy can be used as an extension of functional testing. It is estimated that with good black-box testing you can exercise only up to 70% of the code, so other techniques are required (Farrell-Vinay 2008, 209). White-Box testing offers more accurate approach to test the code. But because of more thoroughness way to test, it will take time to exercise all statements or true/false branches that occur in a module or function, and so this strategy is more effective with smaller pieces of code. (Burnstein 2003, 64.)

Practically white-box testing is usually done using the logic of the system, but not following the specifications. The tester is often the programmer, because he already has the knowledge of the software and ability to create tests in given time. In fact, it would take more than reasonable time to test completely even a trivial program with this method, so some shortcuts and prioritization has to be used. (Myers 2004, 14.) One good strategy could be a combination of both strategies i.e. gray box testing.

2.5.3 Gray-Box testing

Gray (or Grey) box testing is a combination of functional (black) and structural (white) testing strategies. As you can imagine, in gray-box testing the software under test is (in) a gray box, you can see inside but not clearly. Therefore, the tester has limited knowledge of the system but the test cases are designed as in black box testing. (Softwaretestingfundamentals.)

With gray box testing we have the advantages of both functional and structural testing and the test coverage can be increased. However, with this combination comes also the disadvantages from both methods. For example, the code coverage may suffer due the limited access to source code or binaries and the identification of defects can be difficult. (Erenthika, 2012.)

Briefly the tests are done from the outside but with better information of the system. This gives a tester with little programming knowledge an opportunity to create the tests based on the code and then applied to the user interface elements of the SUT. (SBP tech blog, 2012.) However, as the gray box strategy can be used with any testing level, it is especially used in integration testing and to test Web services (Softwaretestingclass).

2.6 Testing levels

Haikala & Mikkonen (2011, 206-207), set three different testing levels for the testing according to the V-model: Unit testing (module testing, unit testing), integration testing and system testing. Burnstein (2003, 64), counts also an acceptance test in some type as a one and continues that each of these levels has their own goals and may contain one or more sublevels or phases. In a figure below (figure 4), Burnstein clearly points out the cohesion of testing levels and parts to be tested.

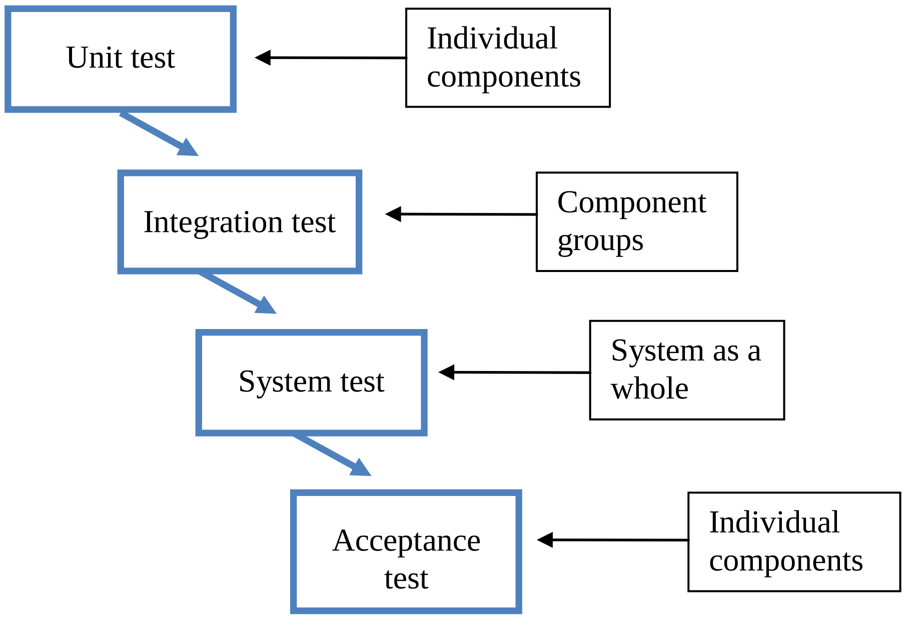


Figure 4. Levels of testing (Burnstein 203, 134).

2.6.1 Unit (module) testing

Unit testing tests a smallest piece of testable code, which can be a single class or a module with usually 100 - 1000 lines of code (Haikala & Mikkonen 2011, 207). However, the definition of "unit" can vary. Pezze & Young (2008, 282), defines unit as follow:

> "In object-oriented programs, small sets of strongly related functions or procedures are naturally identified with classes, which are generally the smallest work units that can be systematically tested."

As they continue that a single method should not be automatically considered as a unit, since they act by defining object state in a single class and the effect can be often seen only after effecting with other methods. However, Burnstein (2003, 137), writes that the method and the class or object are usually defined as a unit by researchers in object-oriented systems. Whereas in procedural languages unit is perceived as a function or procedure.

The main objective for unit testing is to make sure that each individual unit in the software under test is working as described in the specification Burnstein (2003, 138). Nevertheless, Myers (2004, 70), reminds that the goal is not to show that the unit equates the specification, only that the unit does contradict it. He also encourages to unit test by stating that focusing initially on smaller units larger elements can be managed. Debugging as well becomes easier when found error can be traced to particular unit, and that the testing can be done by testing several modules simultaneously.

Traditionally unit testing is done after the code for the unit is written. But especially with agile methods or eXtreme programming (an agile methodology highly responding to customer requirements (Extremeprogramming, 2009).), unit testing can be done using Test Driven-Development, known as TDD, where the tests are written before writing the code (Farrell-Vinay 2008, 232).

Practically in test driven-development, the test is written first and then the functional code is created to pass the test. TDD provides an active way to unit test the code and offers an opportunity to find a bug when it has been created, but it does not cover other testing. Also notable thing is that TDD is more of a developing method than a testing method. (Agiledata, 2010.)

So who should write the unit tests, the developer or someone else? Does developer who tests his own code create a quality gate or is it likely that he tries to prove the function of the program and not to find the errors in it? Farrell-Vinay (2008, 237), lists several reasons why the system test group should and should not unit test. The bottom line is that a special group of testers does not offer a significant benefit. The developers themselves can write the best tests for their code with lower costs and without having to learn the code. However, it is important to have another programmer to review the tests written by the developer.

2.6.2 Integration testing

Burnstein (2003, 152), sets two main objectives for Integration test. First, to find errors in the interfaces of units. Second, to assemble units to subsystems and subsystems to a full system. Also like in other testing levels, after completing this level the system should be ready for next level of testing.

She continues that the tester should not think that he is doing the same tests that have been already made in previous unit testing level. In integration testing, the modules are tested together, not individually, and therefore problems on communications and interfaces may occur.

Integration testing can be done in two ways: Bottom-up and Top-down. Bottom-up integration starts with testing the lowest-modules that do not call other modules i.e. bottom of the module hierarchy. These modules are integrated to upper level modules until the top is reached. Benefit in Bottom-up method is that the lowest modules are usually well tested, but the problem is that the complete system does not exist until all modules are integrated. (Burnstein 2003, 152-155.)

Top-down integration goes vice versa. The integration is started at the top level and starting the module below, all modules will be integrated and tested, until we reach the bottom. The rule is that when integrating lower level modules the upper caller module should have already been tested. This ensures the testing of upper level modules at early stages, but makes it difficult to make changes to the upper level modules if errors are found at low levels. (Burnstein 2003, 155-156.)

Often a combination (known as the sandwich or backbone strategy) of these two approaches is used. This can be due to for example of reuse existing modules or commercial off-the-shelf components, or need of develop prototypes for user feedback. In practice, we start from both ends, and following the hierarchy integrate towards the middle. (Pezze & Young 2008, 410.)

Below is a simple structure chart of integration (Figure 5). Despite the direction used, the modules are integrated together following the module hierarchy. In top-down integration we start at the top, in bottom-up from the lowest module and in sandwich from both ends. The integration and testing is done, until the entire system has been integrated.

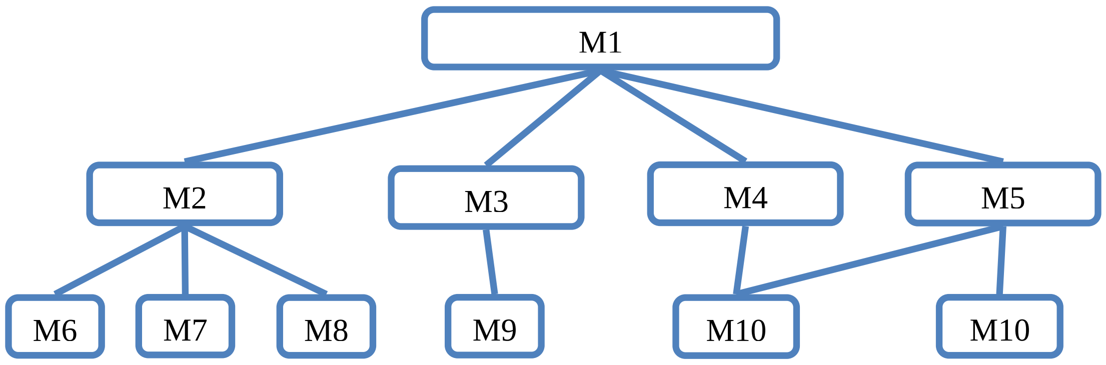


Figure 5. Module hierarchy (Burnstein 2003, 154).

2.6.3 System testing

After integration testing we should have an entire system to be tested. During system testing the system is compared to its requirements specification, functional specification, and manual and other customer level documentation. In this level of testing the testers should be independent of the development process, because the developers often tend to test the parts that are known to be working and therefore do not find so many bugs as a testers outside form the development. (Haikala & Mikkonen 2011, 208-209.)

Farrell-Vinay (2008, 243), points out that again the goal is not to show that the system meets its specification and working properly, the objective is to show it does not. System testing requires a large amount of resources, even half of the total testing resources, so it is obvious that this phase should be carefully planned. Also professional testers are often recommended. Haikala & Mikkonen (2011, 208), lists following types of system tests, that can tell about the nature of this testing level: Field testing, stress testing, reliability testing, installation testing and usability testing. They also include acceptance testing into system testing, but we will discuss it in detail later.

Actually there are tens of different test types to be used, but they all have the same goal: To test and exercise the software system as a whole and ensure the user's experience (Guru99). In practice, some of the test suites used in integration and unit testing can be used in system testing, but this should be due to using system cases early, not reusing unit and integration test cases (Pezze & Young 2008, 418).

2.6.4 Acceptance testing

Pezze & Young (2008, 422), define acceptance testing as follow: Based on statistical testing results or comparison to experience with previous products, the objective of acceptance testing is to determine if the product is ready for release. Therefore, this testing level is done from the user's point of view

Statistical testing requires test data from precise defined samples (what and how much). Yet, the results of lower level tests (i.e. systematic tests) are not valid, because their purpose is to focus finding errors, not produce statistically representative data.

Acceptance testing can be done also with users. These variations are called alpha & beta testing. Alpha testing often refers to a testing performed within the developing organization. Whereas beta testing is done at user's sites. The benefit and the downside from testing with users is to have users from each segment and weigh the results with right value.

In industry, the acceptance testing often has a real value, when the both parties agree that the service or product meet the requirements of the agreement. This usually works as a trigger for partial or full payment. Errors found at this level can be very harmful for the image and relationships of the companies and create large costs. (Aiia 2010.)

2.7 Isolation

As said earlier, unit testing tests a smallest piece of testable code. In order to test just and only the unit, isolation is often needed. By isolating the unit from other units, we ensure that we do not cross the unit boundaries and write integration tests instead.

Carrie Prebble (2008), puts it in this way: How could you find bugs in your unit under test if your test harness includes a library and connection via network? How would you know if the unit fails or the connection fails? The answer - you cannot. You can't find bugs from the unit unless you test only the unit.

This is all about of focusing one unit at a time, and removing all other dependencies that might cause an error. Prebble (2008), writes also that an isolated unit is controlled by the test. For example, if an unit creates a new helper GregorianCalendar, then December dates cannot be tested if it's not December. But with isolation we can create a test which creates the calendar and puts it to the unit.

The isolation in practice is achieved by using test doubles to replace the real depended-on components (DOC). The trick is that the test double does not have to act precisely as the real DOC, only to provide the same API so the system under test think it is the real one (Meszaros, 2009). This not only makes the testing easier but saves time and effort compared to non-isolated tests.

However, the main goal is to make impossible tests possible. Meszaros (2009), compares test double to a stunt actor, hired by film makers to act in risky scenes. Requirements for stunt double are depending on the scene. He / she may resemble only a bit the real one or may not be able to act at all, but what matter is that he is able to do all the dirty work.

The test double should be used to cover as little as possible. They should not replace the parts of the system under test that we are currently testing, because we want to test the real software, not fake. We should keep in mind that we can create different doubles for different tests, even with the same DOC. (Meszaros, 2009.)

There are several different types of test doubles, usually depending on the isolation framework used. The types used with common frameworks are represented next chapter in more detail. Meszaros (2009), characterizes most used doubles as in the figure below (figure 6), where from left to the right the object gets more intelligent and advanced. In brief: The dummy object can be as simple as null object, whereas a fake object can contain the same (but simplified) functionality as the real DOC.

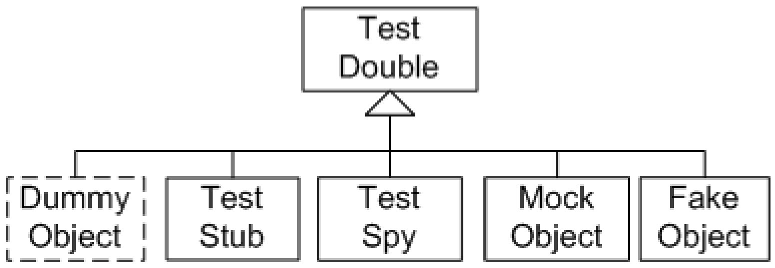


Figure 6. Test doubles (Meszaros 2009).

The type we should use depends on the functionality we want to mimic. Each type has their own ups and downs. Unconfigurable test doubles like dummy or fake objects are used, when we don't need pre-configured responses or expectations (Meszaros, 2009).

Hard-coded test doubles instead are used in single test cases where we tell the double what to return and expect (i.e. test stub, test spy & mock object). These doubles are usually handmade for very simple or very specific behavior and therefore need more effort. (Meszaros, 2009.)

Configurable test doubles are used when we want to use the test double in several tests, or reduce test code duplication. In a setup phase the test double's interface is configured to hold appropriate values during the runtime. When the methods on the test double are called by the system, the double returns the values of predefined variables. (Meszaros, 2009.)

3 ISOLATION FRAMEWORKS

In this chapter popular isolation frameworks for .Net and are introduced. Four frameworks: MS Moles, MS Fakes, Moq and FakeItEasy are taken into closer look and example tests are written and ran using them. All four are used with C# programming language on Visual Studio development environment. Moles works on 2010 version and Fakes on 2012. Moq and FakeItEasy examples are written with VS 2010, but they can be used with both versions.

3.1 Isolation frameworks used today

There are many frameworks that can handle isolation in .Net environment. Some have more powerful test objects than others, some are new or based on old codebase. Two polls (figure 7 & figure 8) below can give us a hint about the usage of mostly used isolation frameworks of today.

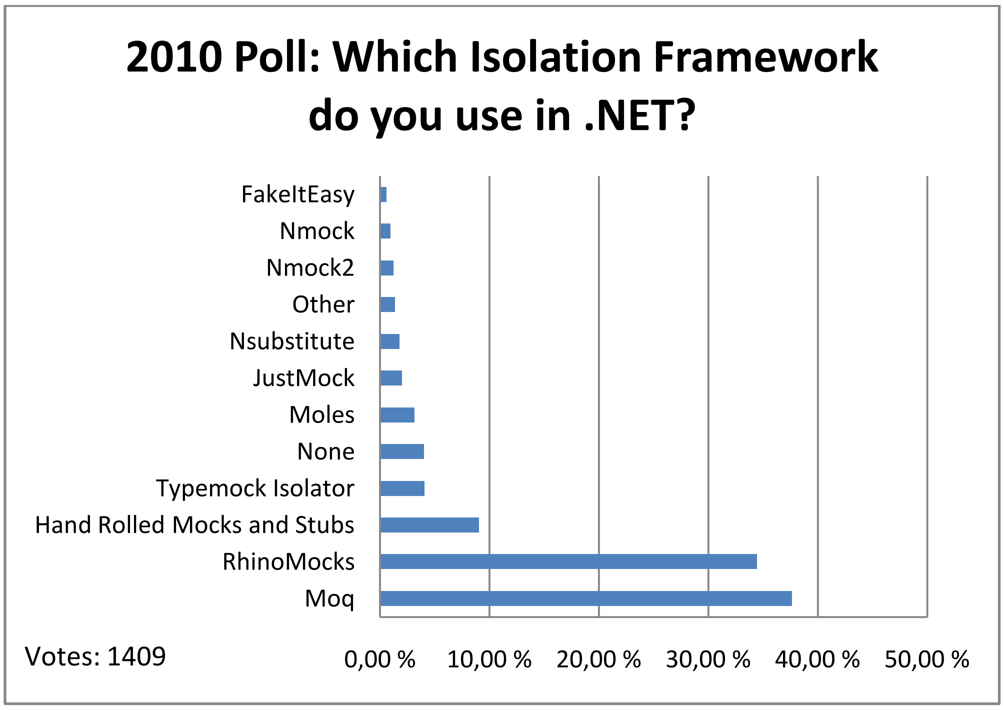


Figure 7. Isolation frameworks used in .Net 2010 (Osherove 2010).

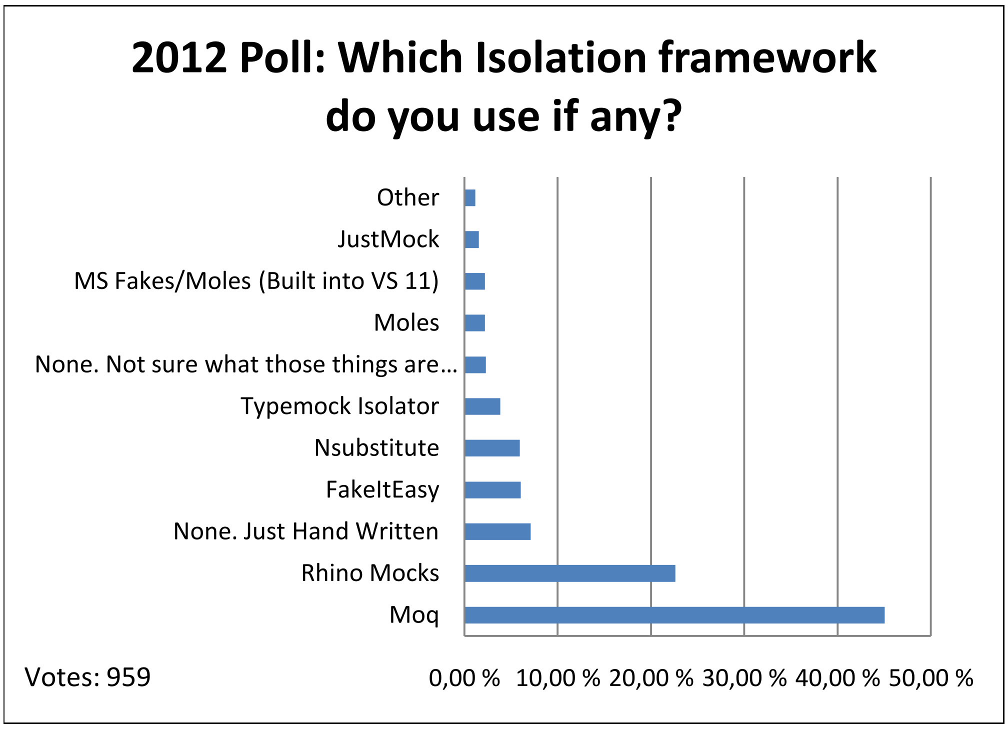


Figure 8. Isolation frameworks used in .Net 2012 (Osherove 2012).

3.2 Mostly used isolation frameworks

Moq is widely used open source isolation framework and has gained users especially from older **Rhino Mocks** due their similarities and same code base (CastleProject 2011).

Moles was a project of Microsoft Research but its development has ended. Moles has been replaced by its successor **Fakes** which is integrated in Visual Studio 2012 Ultimate version (MSDN 2013), and is also available for Premium version with update 2 (Harry, 2013). Yet, Moles is still in use because it is free and working with 2008 and 2010 versions (Microsoft research 2010a).

Typemock Isolator is a commercial isolation framework and has pricing starting from 799 $. Today there is a free edition of the program but it comes with limited functionality (Typemock, 2013). Typemock Isolator could be a powerful tool, but because it is not free and has quite small amount of users, we do not look into it in more detail.

FakeItEasy is a newcomer and it is basically a mix of Moq and Rhino Mocks. This open source framework has only one type of fake object (called fake). (FakeItEasy 2012.)

NSubstitute is open source framework that aims at the same target as FakeItEasy with easy to use and start with. Uses "substitute" to cover common test doubles. (NSbustitute.)

Handwitten Mocks. Some users still use handwritten types in order to break the dependencies and isolate their code. This method requires more time and good knowledge. Handwritten types are also hard to make compatible with other types and may be difficult to use by others than the creator. (Meszaros, 2009.)

3.3 Moles framework

Moles is an isolation framework for .Net developed with test generation tool called Pex by Microsoft Research. Moles provides two test objects (stubs and moles) to detour any .Net method. It can be used with Visual Studio 2008 & 2010 and also with other testing frameworks like NUnit. (Microsoft research, 2010b.) It is important to notice that the Moles Framework is no longer developed. Microsoft is going to replace Moles with Fakes and do not offer support for Moles anymore. (MSDN 2013.)

Fakes framework has some changes to Moles and is still under development. Main differences between Moles in Visual Studio 2010 and Fakes in Visual Studio 2012 Release Candidate version are listed in chapter 3.4.1.

The Moles framework has two different kinds of isolation techniques (i.e. test doubles) stub types and mole types. These components provide different ways to detour objects in various situations in unit testing. Basically the differences between stubs and moles can be determined as following.

- Stub types should be used to detour virtual methods and interfaces.
- Mole types should be used for a code that you cannot detour with stubs. For example sealed classes or static, non-virtual methods.

Microsoft recommends users to prefer stubs when possible. Stubs are lighter than mole types and have less performance issues with runtime rewriting (Microsoft research, 2010a).

3.3.1 Installing Moles framework in Microsoft Visual Studio 2010

In order to Install Moles framework in Microsoft Visual Studio 2010 do following.

1. Select Tools / Extension Manager... (figure 9)

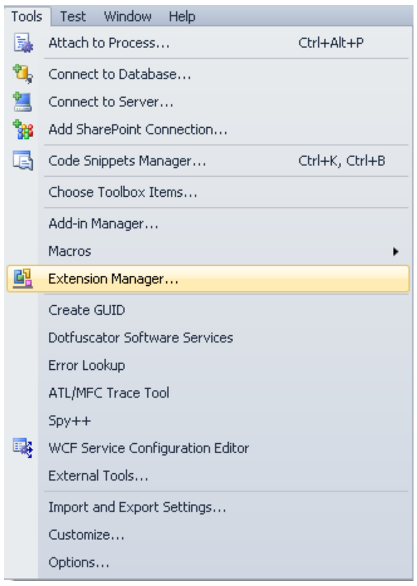
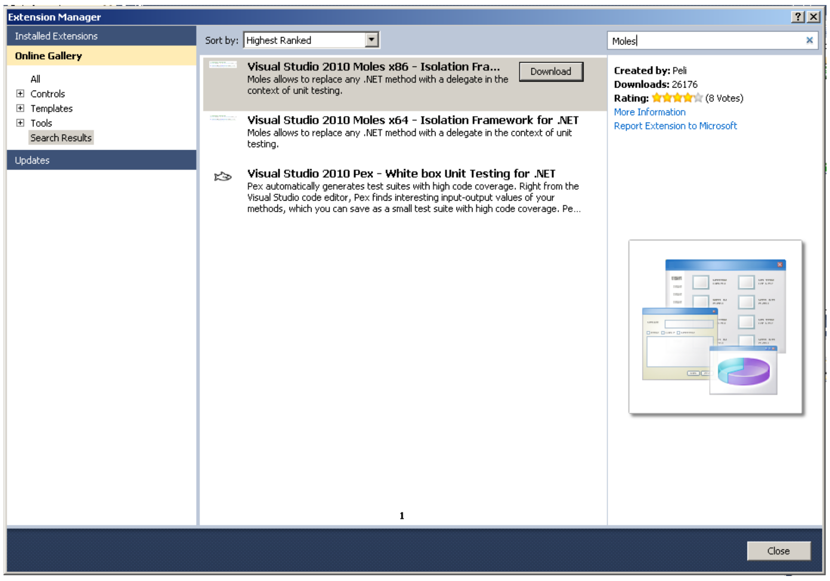


Figure 9. Tools bar and Extension Manager.

2. Select "Click here to go online and find extensions"
3. Type “Moles” in search bar on top right and hit enter



Visual Studio 2010 Moles x86 - Isolation Fra...
Moles allows to replace any .NET method with a delegate in the context of unit testing.

Visual Studio 2010 Moles x64 - Isolation Framework for .NET
Moles allows to replace any .NET method with a delegate in the context of unit testing.

Visual Studio 2010 Pex - White box Unit Testing for .NET
Pex automatically generates test suites with high code coverage. Right from the Visual Studio code editor, Pex finds interesting input-output values of your methods, which you can save as a small test suite with high code coverage. Pe...

Created by: Peli
Downloads: 26176
Rating: ★★★★☆ (8 Votes)

Figure 10. Extension Manager.

4. Select right version depending on your system (x86/x64) and click download (figure 10).
5. A file download will pop up. Download and run the installer.

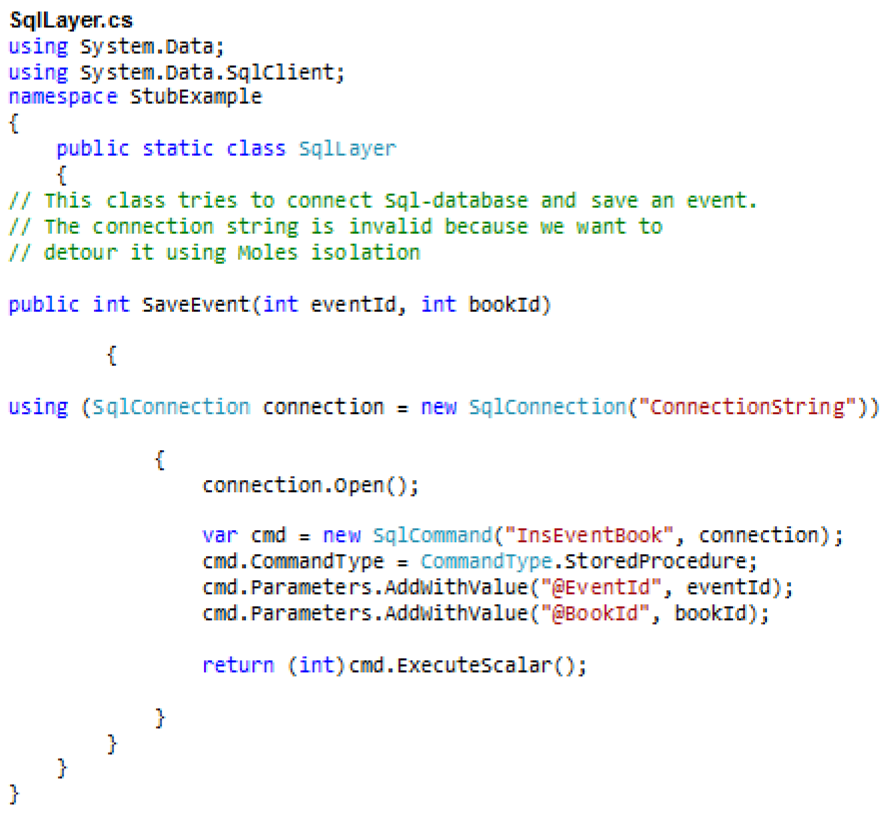
3.3.2 Mole code example

The example is a simple library project which has two class-files LibraryEvent.cs and SqlLayer.cs. In LibraryEvent.cs (figure 11) the program first defines books, user and events. At the bottom AddEventBook() tries to connect to SQL-database via SqlLayer.cs (figure 12) to save the event. The trick is that there is no Sql-database so in order to run unit tests we have to use an isolation framework to fake the SaveEvent().

LibraryEvent.cs

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;

namespace MolesExample
{
    public class LibraryEvent
    {
        // Constructor for event and user
        public int EventId
        { get; private set; }
        public int UserId
        { get; private set; }

        // Class for Books
        public class EventBook
        {
            public int BookId { get; private set; }
            public int EventBookId { get; private set; }

            public EventBook(int eventBookId, int bookId)
            {
                EventBookId = eventBookId;
                BookId = bookId;
            }
        }

     // List and Collection for books in event
     private List<EventBook> _eventBooks = new List<EventBook>();
     public ReadOnlyCollection<EventBook> EventBooks {get; private set;}

        // Simple definition for a library event
        public LibraryEvent(int eventId, int userId)
        {
            EventId = eventId;
            UserId = userId;
            EventBooks = new ReadOnlyCollection<EventBook>(_eventBooks);
        }

        // Saving a library event into sql-database via SqlLayer-class
        // and adding a Book to Eventlist
        public void AddEventBook(int BookId)
        {
            var eventBookId = SqlLayer.SaveEvent(EventId, BookId);
            _eventBooks.Add(new EventBook(eventBookId, BookId));
        }
    }
}
```

Figure 11. Mole example Library event class.



```
SqlLayer.cs
using System.Data;
using System.Data.SqlClient;
namespace StubExample
{
    public static class SqlLayer
    {
// This class tries to connect Sql-database and save an event.
// The connection string is invalid because we want to
// detour it using Moles isolation

public int SaveEvent(int eventId, int bookId)

        {

using (SqlConnection connection = new SqlConnection("ConnectionString"))

            {
                connection.Open();

                var cmd = new SqlCommand("InsEventBook", connection);
                cmd.CommandType = CommandType.StoredProcedure;
                cmd.Parameters.AddWithValue("@EventId", eventId);
                cmd.Parameters.AddWithValue("@BookId", bookId);

                return (int)cmd.ExecuteScalar();

            }
        }
    }
}
```

Figure 12. Mole example Sql layer class.

Before we can write the test, we have to add a reference to the test project and add a moles framework assembly to the reference. These operations will create and prepare the fake types of our actual code to be used in our isolated unit tests.

1. Open the example project in Visual Studio
2. Add new test project (File > Add > New Project… > Visual C# / Test)
3. Now we got an empty test project and our Solution Explorer should look like in figure 13.

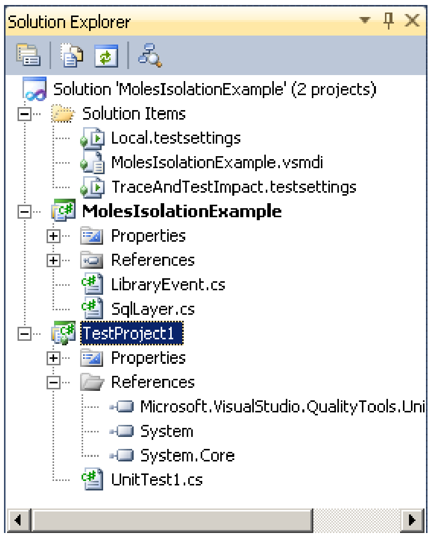


Figure 13. Solution explorer after creating a unit test project.

4. Test project comes with solution items, project properties, references and a class file where the actual test script will be written. Add a reference (figure 14) to our actual project (MolesIsolationExample) by right clicking on TestProject1 / References > Add Reference… > Projects (tab) > MolesIsolationExample > OK

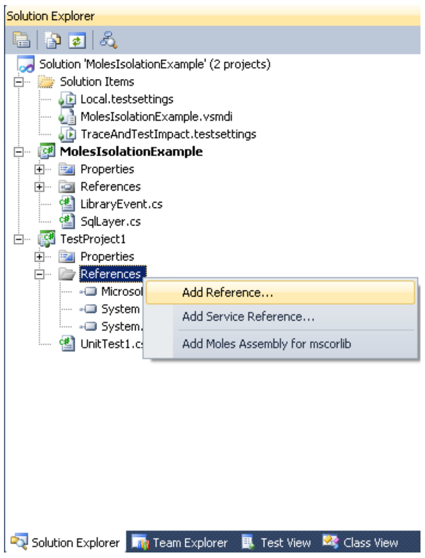


Figure 14. Adding Reference to the project.

5. Add Moles assembly to the reference we just added (“MolesIsolationExample”). Right click on the reference > Add Moles Assembly (figure 15).

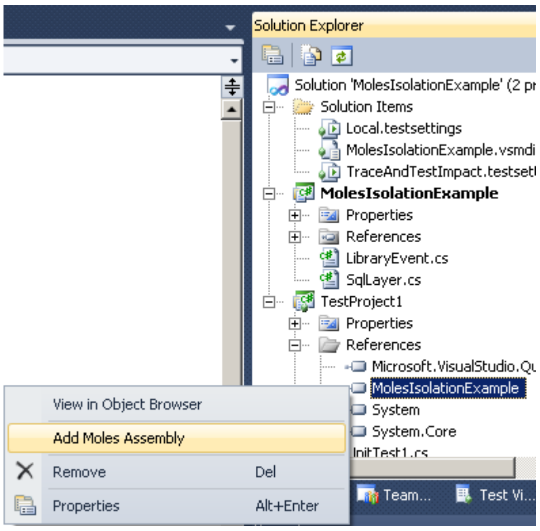


Figure 15. Adding Moles Assembly.

6. Adding Moles Assembly adds few new references and xml-file “ProjectName.moles” to our test project. After building the program our Solution Explorer should look like below in figure 16. (Note that it may need a refresh to show added files).

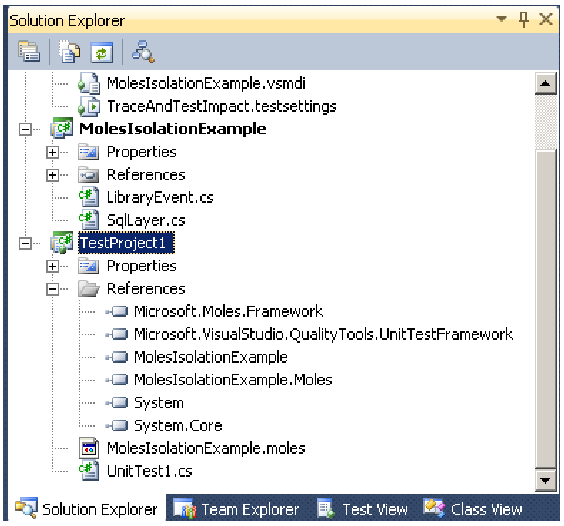
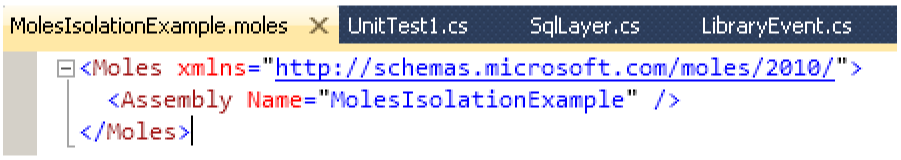


Figure 16. Solution Explorer after adding the assembly.



```
<Moles xmlns="http://schemas.microsoft.com/moles/2010/">
  <Assembly Name="MolesIsolationExample" />
</Moles>
```

Figure 17. Moles example .moles xml-file.

Moles framework uses the .moles file (figure 17) to generate the code for stub types and mole types. The xml-file points the assembly that you want to mole.

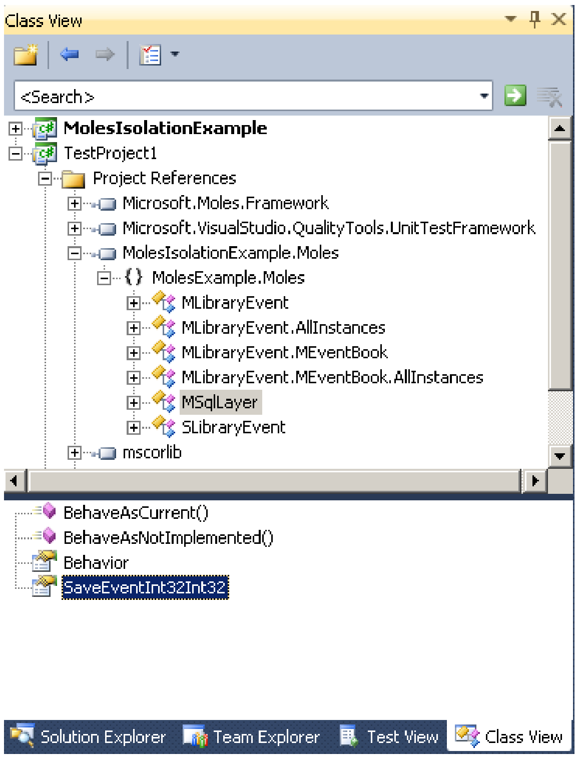
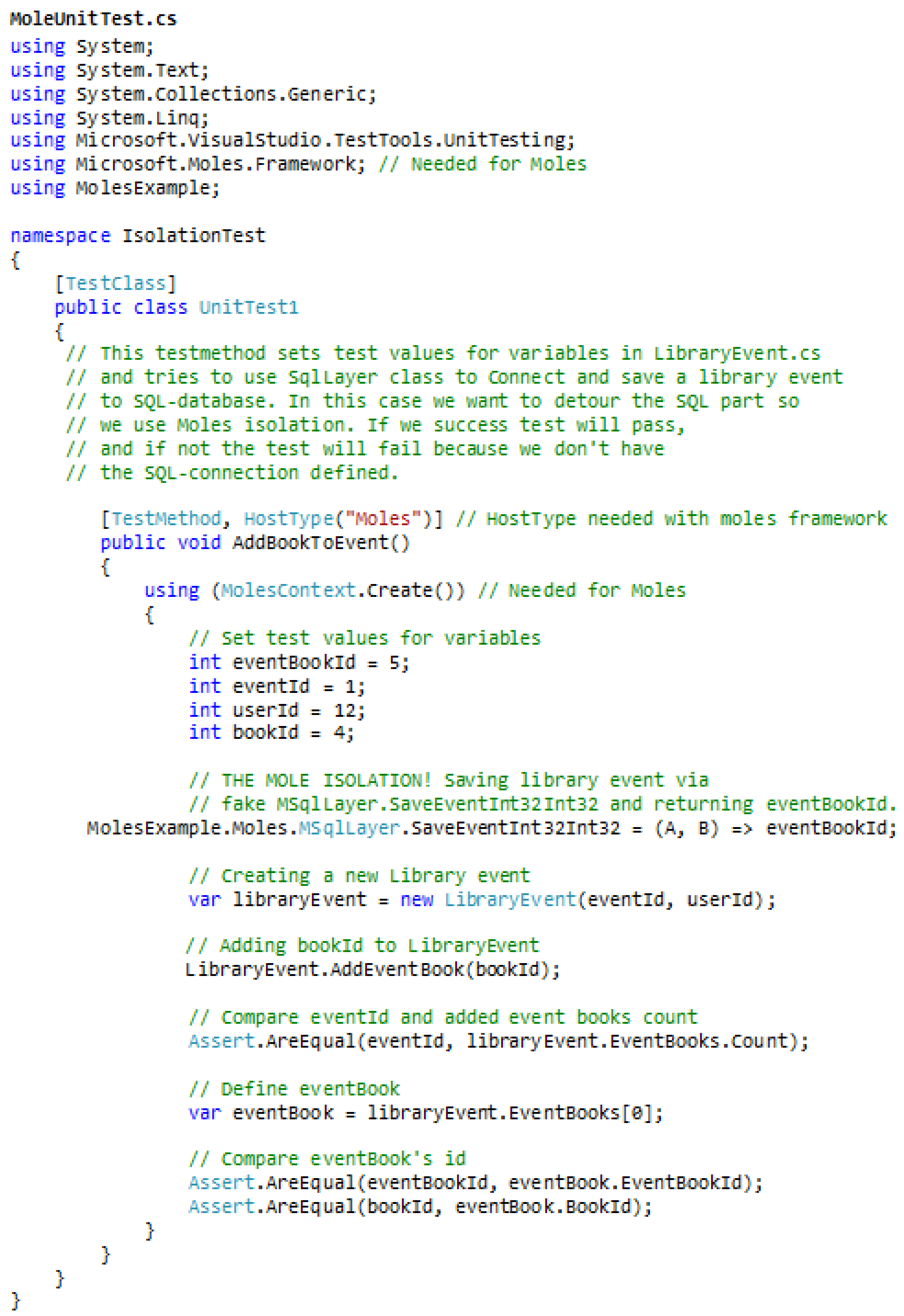


Figure 18. Fake types in Class View.

7. Notice items under TestProject1 > Project References > MolesIsolationExample.Moles > MSqlLayer (figure 18). We can see that after adding the reference and moles assembly, fake types are created under “ProjectName.moles”. These fake types will be used to detour the real ones in our isolated unit tests.

Moles framework uses prefixes to mark files. For example in figure 18 we can see that Moles has created Mole type files starting with “M” and Stub type files starting with “S”. In our example project we are going to use a fake “MSqlLayer.SaveEventInt32Int32()” to detour the real SqlLayer.SaveEvent().

Now we have prepared our code to be used in isolated unit tests. In order to write the test, open UnitTest1.cs under the TestProject. This is where the actual test code will be written. An Example test is below in figure 19.



```
MoleUnitTest.cs
using System;
using System.Text;
using System.Collections.Generic;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Microsoft.Moles.Framework; // Needed for Moles
using MolesExample;

namespace IsolationTest
{
    [TestClass]
    public class UnitTest1
    {
     // This testmethod sets test values for variables in LibraryEvent.cs
     // and tries to use SqlLayer class to Connect and save a library event
     // to SQL-database. In this case we want to detour the SQL part so
     // we use Moles isolation. If we success test will pass,
     // and if not the test will fail because we don't have
     // the SQL-connection defined.

        [TestMethod, HostType("Moles")] // HostType needed with moles framework
        public void AddBookToEvent()
        {
            using (MolesContext.Create()) // Needed for Moles
            {
                // Set test values for variables
                int eventBookId = 5;
                int eventId = 1;
                int userId = 12;
                int bookId = 4;

                // THE MOLE ISOLATION! Saving library event via
                // fake MSqlLayer.SaveEventInt32Int32 and returning eventBookId.
       MolesExample.Moles.MSqlLayer.SaveEventInt32Int32 = (A, B) => eventBookId;

                // Creating a new Library event
                var libraryEvent = new LibraryEvent(eventId, userId);

                // Adding bookId to LibraryEvent
                LibraryEvent.AddEventBook(bookId);

                // Compare eventId and added event books count
                Assert.AreEqual(eventId, libraryEvent.EventBooks.Count);

                // Define eventBook
                var eventBook = libraryEvent.EventBooks[0];

                // Compare eventBook's id
                Assert.AreEqual(eventBookId, eventBook.EventBookId);
                Assert.AreEqual(bookId, eventBook.BookId);
            }
        }
    }
}
```

Figure 19. Mole example unit test class.

To run the test, select Test > Run > All Tests in Solution (figure 20), or right click on the code and select Run Tests.

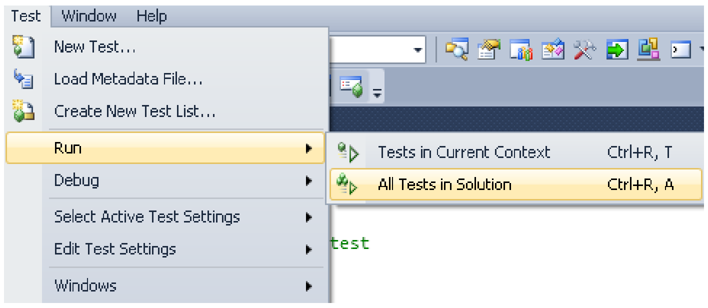
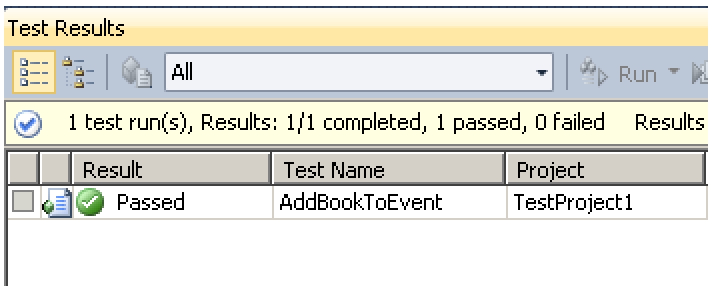


Figure 20. Run All Tests in Solution.

After successful testing you should get following (figure 21) test results.



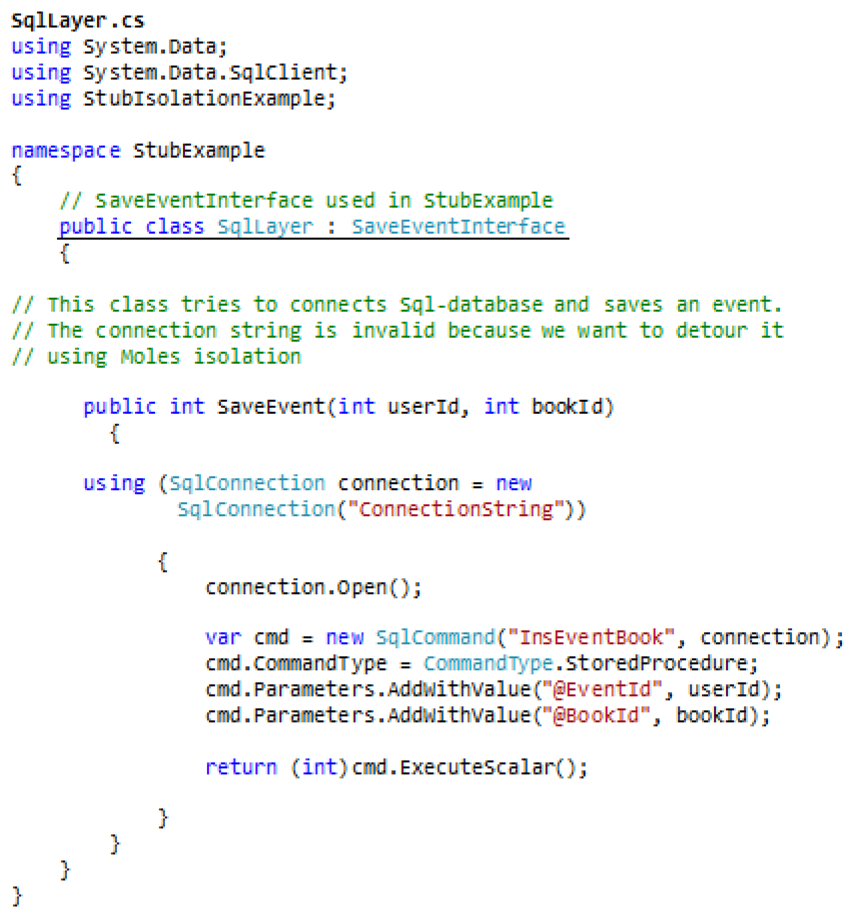
1 test run(s), Results: 1/1 completed, 1 passed, 0 failed Results

Result	Test Name	Project
Passed	AddBookToEvent	TestProject1

Figure 21. Mole example Test Results.

3.3.3 Stub coding example

Stub type is used to generate fake stub implementations of virtual methods and interfaces. It is recommended to prefer stubs to moles. The example project (figures 22, 23 and 24) is same library project we used in the previous mole example. The difference is that we use a simple interface called SaveEventInterface to call saveEvent() in SqlLayer.cs to show how to use stub type to fake an interface. Changes to mole example are underlined.



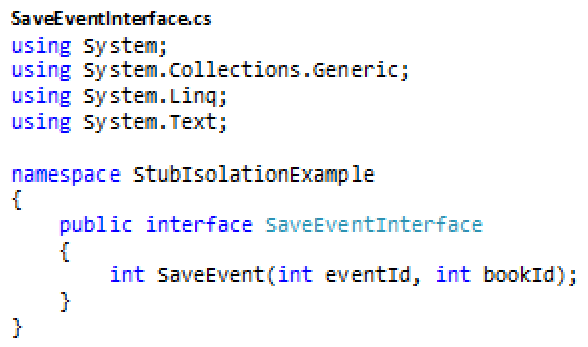
```
SqlLayer.cs
using System.Data;
using System.Data.SqlClient;
using StubIsolationExample;

namespace StubExample
{
    // SaveEventInterface used in StubExample
    public class SqlLayer : SaveEventInterface
    {

// This class tries to connects Sql-database and saves an event.
// The connection string is invalid because we want to detour it
// using Moles isolation

      public int SaveEvent(int userId, int bookId)
        {

      using (SqlConnection connection = new
              SqlConnection("ConnectionString"))

            {
                connection.Open();

                var cmd = new SqlCommand("InsEventBook", connection);
                cmd.CommandType = CommandType.StoredProcedure;
                cmd.Parameters.AddWithValue("@EventId", userId);
                cmd.Parameters.AddWithValue("@BookId", bookId);

                return (int)cmd.ExecuteScalar();

            }
        }
    }
}
```

Figure 22. Stub example Sql Layer class.

LibraryEvent.cs

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using StubIsolationExample;

namespace StubExample
{
    public class LibraryEvent
    {
        // Constructor for event and user
        public int EventId
        { get; private set; }
        public int UserId
        { get; private set; }

        // Class for Books
        public class EventBook
        {
            public int BookId { get; private set; }
            public int EventBookId { get; private set; }

            public EventBook(int eventBookId, int bookId)
            {
                EventBookId = eventBookId;
                BookId = bookId;
            }
        }

        // List and Collection for books in event
        private List<EventBook> _eventBooks = new List<EventBook>();
        public ReadOnlyCollection<EventBook> EventBooks { get; private set; }

        // Interface definition for StubExample
        private SaveEventInterface _saveEventInterface;

        // Simple definition for a library event
        public LibraryEvent(int eventId, int userId, SaveEventInterface
           saveEventInterface)
        {
            EventId = eventId;
            UserId = userId;
            EventBooks = new ReadOnlyCollection<EventBook>(_eventBooks);
            _saveEventInterface = saveEventInterface;
        }

// Saving a library event into sql-database via SqlLayer-class
// and adding a Book to Eventlist
        public void AddEventBook(int BookId)
        {
            var eventBookId = _saveEventInterface.SaveEvent(EventId, BookId);
            _eventBooks.Add(new EventBook(eventBookId, BookId));
        }
    }
}
```

Figure 23. Stub example Library event class.



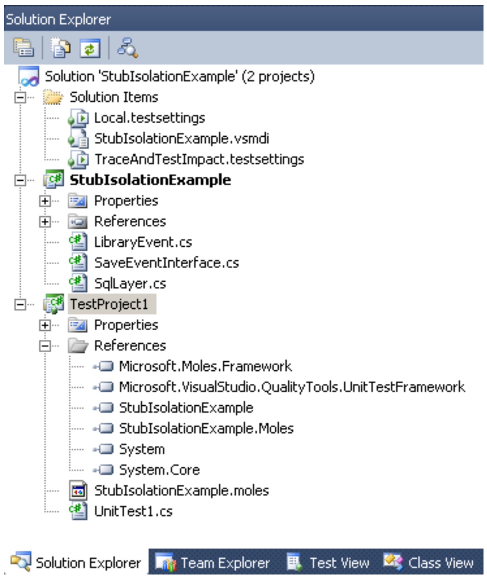
SaveEventInterface.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace StubIsolationExample
{
    public interface SaveEventInterface
    {
        int SaveEvent(int eventId, int bookId);
    }
}
```

Figure 24. Stub example Save event class.

Same way as in the Mole example we need to add the project reference to our test project and add the moles assembly to that reference. This creates the fake types of our real objects. This is done as follow:

1. Open the stub example project.
2. Add new test project (File > Add > New Project… > Visual C# / Test)
3. Add a reference to our actual project (StubIsolationExample). Right click on TestProject1 / References > Add Reference… > Projects (tab) > StubIsolationExample > OK
4. Add Moles assembly to the reference you just added ("StubIsolationExample"). Right click on the reference > Add Moles Assembly.
5. Adding Moles Assembly adds few new references and xml-file "ProjectName.moles" to your test project. After building the program your Solution Explorer should now look as in figure 25. (Note that it may need a refresh to show added files)

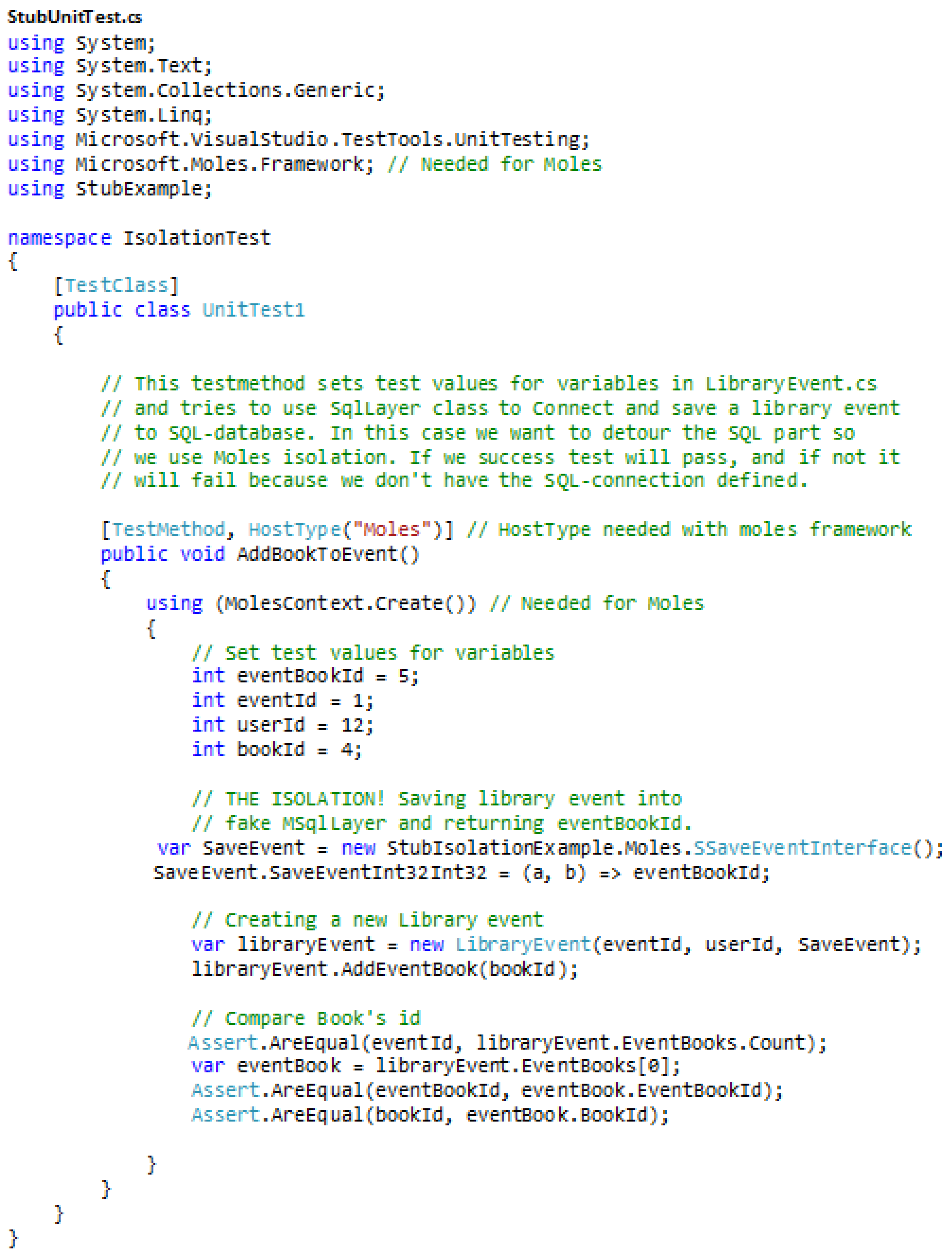


Solution Explorer

```
Solution 'StubIsolationExample' (2 projects)
  Solution Items
    Local.testsettings
    StubIsolationExample.vsmdi
    TraceAndTestImpact.testsettings
  StubIsolationExample
    Properties
    References
    LibraryEvent.cs
    SaveEventInterface.cs
    SqlLayer.cs
  TestProject1
    Properties
    References
      Microsoft.Moles.Framework
      Microsoft.VisualStudio.QualityTools.UnitTestFramework
      StubIsolationExample
      StubIsolationExample.Moles
      System
      System.Core
    StubIsolationExample.moles
    UnitTest1.cs
```

Figure 25. Solution Explorer after adding moles assembly and reference.

The test can now be written into UnitTest1.cs under the TestProject1. An example test where Moles is used to mimic the saveinterface is in followed in figure 26.



```
StubUnitTest.cs
using System;
using System.Text;
using System.Collections.Generic;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Microsoft.Moles.Framework; // Needed for Moles
using StubExample;

namespace IsolationTest
{
    [TestClass]
    public class UnitTest1
    {

        // This testmethod sets test values for variables in LibraryEvent.cs
        // and tries to use SqlLayer class to Connect and save a library event
        // to SQL-database. In this case we want to detour the SQL part so
        // we use Moles isolation. If we success test will pass, and if not it
        // will fail because we don't have the SQL-connection defined.

        [TestMethod, HostType("Moles")] // HostType needed with moles framework
        public void AddBookToEvent()
        {
            using (MolesContext.Create()) // Needed for Moles
            {
                // Set test values for variables
                int eventBookId = 5;
                int eventId = 1;
                int userId = 12;
                int bookId = 4;

                // THE ISOLATION! Saving library event into
                // fake MSqlLayer and returning eventBookId.
             var SaveEvent = new StubIsolationExample.Moles.SSaveEventInterface();
             SaveEvent.SaveEventInt32Int32 = (a, b) => eventBookId;

                // Creating a new Library event
                var libraryEvent = new LibraryEvent(eventId, userId, SaveEvent);
                libraryEvent.AddEventBook(bookId);

                // Compare Book's id
               Assert.AreEqual(eventId, libraryEvent.EventBooks.Count);
                var eventBook = libraryEvent.EventBooks[0];
                Assert.AreEqual(eventBookId, eventBook.EventBookId);
                Assert.AreEqual(bookId, eventBook.BookId);

            }
        }
    }
}
```

Figure 26. Stub example unit test class.

3.4 Fakes Framework

Fakes framework (also developed by Microsoft Research) is the next generation of Moles framework and has replaced it. These two frameworks are very alike, except for a few functions and naming policy. For example Fakes uses name Shim instead of Mole.

Yet, the Fakes framework is only available built in Visual Studio 2012 Ultimate version due to its dependency on IntelliTrace-component (MSDN 2013). This decision to not include the full unit test tools in other Visual Studio 2012 versions was noted by many developers and also caused astonishment (Visual Studio User Voice, 2013). Eventually Microsoft brought the Fakes also into Visual Studio 2012 Premium in VS212.2 update (Harry, 2013), but other versions remained without it.

3.4.1 Main differences between Fakes and Moles frameworks

The following table 1. contains comparison between Fakes and Moles frameworks in two different Visual Studio versions. Many differences are related to naming policies, but also other features have been changed. By knowing the differences, moving from Moles to Fakes can be made easier and the use of Fakes more effective.

Table 1. Main differences between Fakes and Moles frameworks.

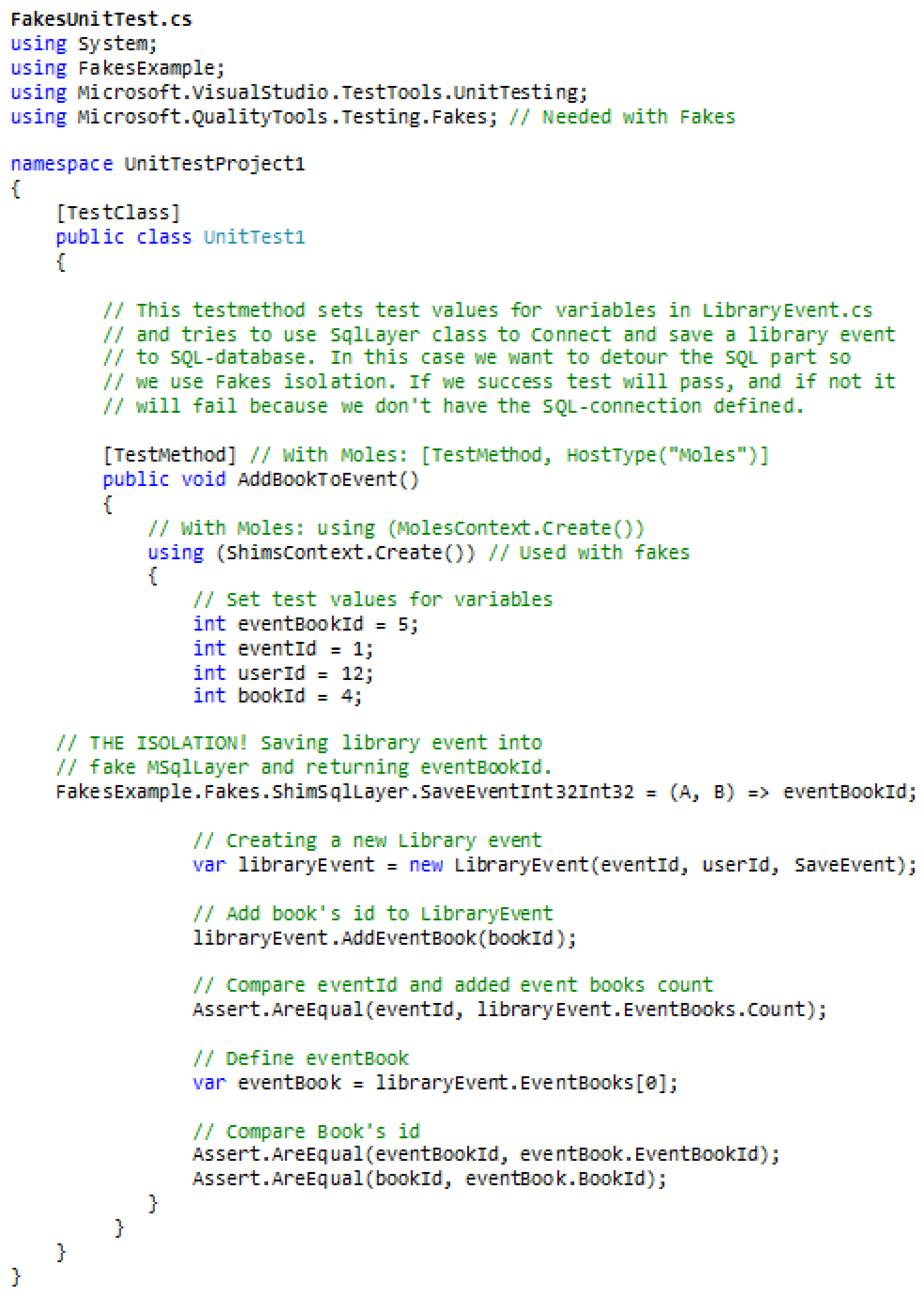
Target	Moles in Visual Studio 2010	Fakes in Visual Studio 2012 RC
HostType	HostType(“Moles”)	No host type needed. Using ShimContext instead of MolesContext. See the fakes example.
Using directive	`using Microsoft.Moles.Framework;`	`using Microsoft.QualityTools.Testing.Fakes;`
Isolation types	.Moles	.Fakes
.xml Assembly filename	.moles	.fakes
Mole / Shim files	Mole (files marked with prefix “M”)	Shim (files marked with prefix “Shim”)
Stub	Stub (prefix “S”)	Stub (prefix “Stub”)
Static constructor	Can be erased through assembly attribute. “[assembly: MolesEraseStaticConstructor(typeof(MyClass))]”	Allowed
Finalizers	Can be erased through assembly attribute: “[assembly: MolesEraseFinalizer(typeof(MyClass))]”	Not supported
CPU-profiler	Can be set between x.86, x.64 or AnyCPU by assembly attribute: “[assembly: MolesAssemblySettings(Bitness = MolesBitness.x86)]”	Handled by IntelliTrace component.

3.4.2 Fakes example

In this example we are going to do unit tests for the same StubIsolationExample we used with Moles Framework. Instead of the Moles now we are going to use Fakes framework with Visual Studio 2012 RC to isolate the dependencies and to run the test. Differences between Fakes and Moles examples are mainly cosmetic and related to the user interface. The differences can be seen in the screenshots in appendix 1. The example test for the Fakes framework is shown in figure 27.

Preparation

1. Open Visual Studio.
2. Open the stub example project.
3. Add a new Unit Test Project (File > Add > New Project… > Visual C# / Test)
4. Add a reference to our actual project (StubIsolationExample). Right click on TestProject1 / References > Add Reference… > Projects (tab) > Select StubIsolationExample > OK
5. Add Fakes assembly to the reference you just added (“StubIsolationExample”). Right click on the reference > Add Fakes Assembly.



```
FakesUnitTest.cs
using System;
using FakesExample;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Microsoft.QualityTools.Testing.Fakes; // Needed with Fakes

namespace UnitTestProject1
{
    [TestClass]
    public class UnitTest1
    {

        // This testmethod sets test values for variables in LibraryEvent.cs
        // and tries to use SqlLayer class to Connect and save a library event
        // to SQL-database. In this case we want to detour the SQL part so
        // we use Fakes isolation. If we success test will pass, and if not it
        // will fail because we don't have the SQL-connection defined.

        [TestMethod] // With Moles: [TestMethod, HostType("Moles")]
        public void AddBookToEvent()
        {
            // With Moles: using (MolesContext.Create())
            using (ShimsContext.Create()) // Used with fakes
            {
                // Set test values for variables
                int eventBookId = 5;
                int eventId = 1;
                int userId = 12;
                int bookId = 4;

    // THE ISOLATION! Saving library event into
    // fake MSqlLayer and returning eventBookId.
    FakesExample.Fakes.ShimSqlLayer.SaveEventInt32Int32 = (A, B) => eventBookId;

                // Creating a new Library event
                var libraryEvent = new LibraryEvent(eventId, userId, SaveEvent);

                // Add book's id to LibraryEvent
                libraryEvent.AddEventBook(bookId);

                // Compare eventId and added event books count
                Assert.AreEqual(eventId, libraryEvent.EventBooks.Count);

                // Define eventBook
                var eventBook = libraryEvent.EventBooks[0];

                // Compare Book's id
                Assert.AreEqual(eventBookId, eventBook.EventBookId);
                Assert.AreEqual(bookId, eventBook.BookId);
            }
         }
    }
}
```

Figure 27. Fakes example unit test class.

3.5 Moq Framework

Moq is a lightweight mocking library for .Net. It is widely used, open source and easy to set up (Moq). Moq derives from the same code base than the older but still used Rhino Mocks (CastleProject 2011). The Moq framework can be downloaded from http://code.google.com/p/moq/downloads/list and attached to the project as instructed in the next chapter or it can be downloaded and installed by using the NuGet Package manager with command: Install-Package Moq. Instructions for NuGet is found in chapter 3.6.1 FakeItEasy installation via Nuget. (Nuget.org, 2011.) Table 2 compares the features between Moq and Moles. Moq does the basic isolation, but cannot be used to isolate static methods or sealed classed.

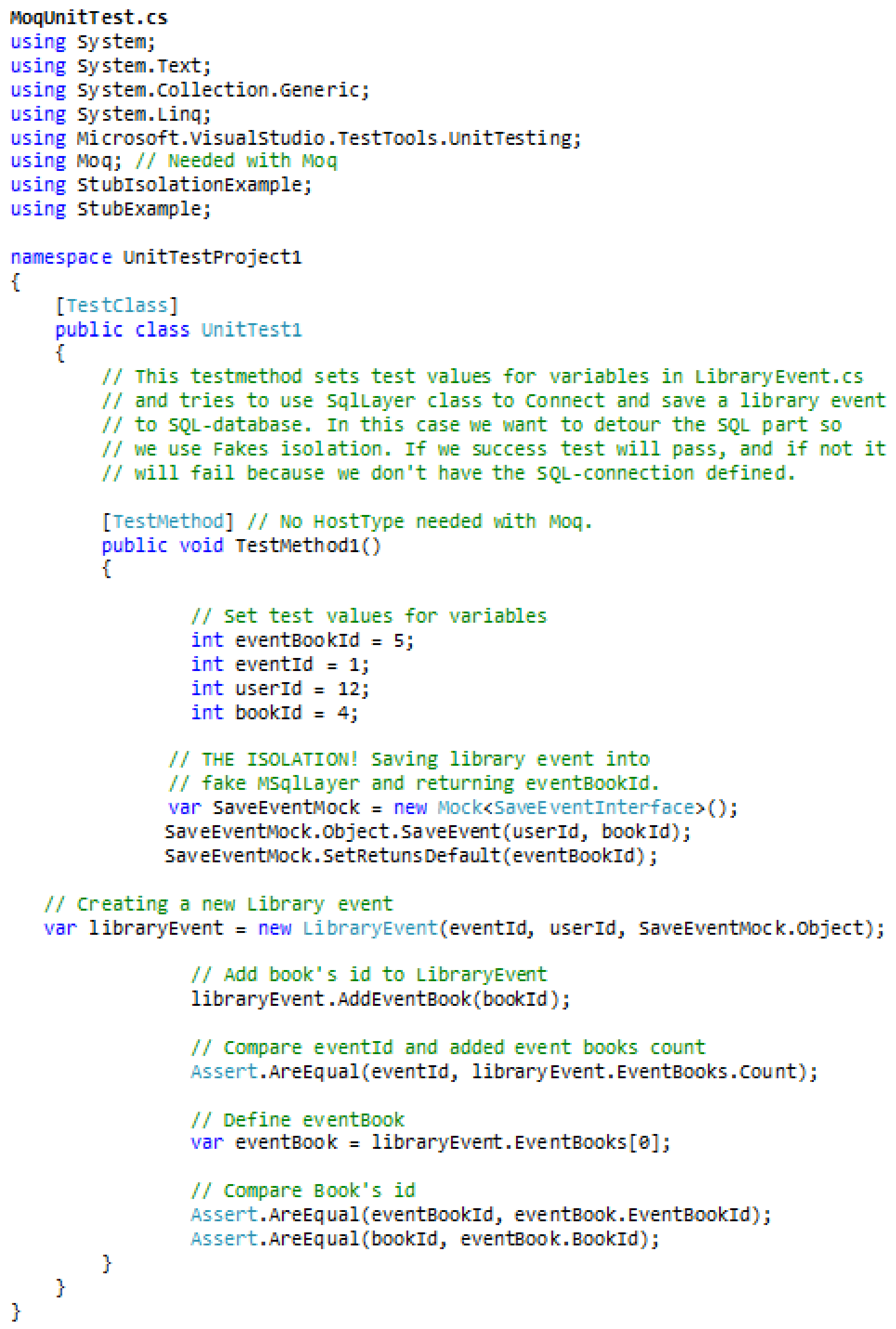
Table 2. Moq features compared to Moles features.

Isolation feature	Moles	Moq
Classes	Yes	Yes
Interfaces	Yes	Yes
Methods	Yes	Yes
Static methods	Yes	No
Sealed classes	Yes	No

The example we use with Moq is the same library project we used with StubIsolationExample with Moles framework. Preparation is done as following.

1. Save Moq framework to your computer
2. Open the StubIsolationExample project in Visual Studio.
3. Add new test project (File > Add > New Project… > Visual C# / Test)
4. Add a reference to our actual project (StubIsolationExample). Right click on TestProject1 / References > Add Reference… > Projects (tab) > StubIsolationExample > OK
5. Add Moq.dll reference to the test project. Right click on TestProject1 / References > Add Reference > Browse > Search and select Moq.dll and click OK.

An example unit test class for StubIsolationExample with Moq is below (figure 28). In this example a mock is created from the SaveEventInterface and then used like the real interface to save the event and return the eventBookId.



```
MoqUnitTest.cs
using System;
using System.Text;
using System.Collection.Generic;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq; // Needed with Moq
using StubIsolationExample;
using StubExample;

namespace UnitTestProject1
{
    [TestClass]
    public class UnitTest1
    {
        // This testmethod sets test values for variables in LibraryEvent.cs
        // and tries to use SqlLayer class to Connect and save a library event
        // to SQL-database. In this case we want to detour the SQL part so
        // we use Fakes isolation. If we success test will pass, and if not it
        // will fail because we don't have the SQL-connection defined.

        [TestMethod] // No HostType needed with Moq.
        public void TestMethod1()
        {

                // Set test values for variables
                int eventBookId = 5;
                int eventId = 1;
                int userId = 12;
                int bookId = 4;

              // THE ISOLATION! Saving library event into
              // fake MSqlLayer and returning eventBookId.
              var SaveEventMock = new Mock<SaveEventInterface>();
             SaveEventMock.Object.SaveEvent(userId, bookId);
             SaveEventMock.SetRetunsDefault(eventBookId);

   // Creating a new Library event
   var libraryEvent = new LibraryEvent(eventId, userId, SaveEventMock.Object);

                // Add book's id to LibraryEvent
                libraryEvent.AddEventBook(bookId);

                // Compare eventId and added event books count
                Assert.AreEqual(eventId, libraryEvent.EventBooks.Count);

                // Define eventBook
                var eventBook = libraryEvent.EventBooks[0];

                // Compare Book's id
                Assert.AreEqual(eventBookId, eventBook.EventBookId);
                Assert.AreEqual(bookId, eventBook.BookId);
        }
    }
}
```

Figure 28. MOQ example unit test class.

3.6 FakeItEasy

FakeItEasy is a free software isolation framework made under MIT licence (FakeItEasy, 2013). It is, at least by the creator, a mix of Rhino Mocks and Moq frameworks. Compared to Moles and Fakes, FakeItEasy has the same features as Moq listed in table 2 in chapter 3.5.

As mentioned before, FakeItEasy has only one kind of fake object type called "fake" and it makes no difference between mocking and stubbing (Hägne 2012). Yet, FakeItEasy has also dummy test double type that can be used to create dummy instances where values are not important to the test (Hägne 2011).

> "I used Rhino Mocks before and I quite liked it, especially after the AAA-syntax was introduced I did like the fluent API of Moq better though. What I didn't like with Moq was the "mock object" where you have to use mock.Object everywhere, I like the Rhino-approach with "natural" mocks better. Every instance looks and feels like a normal instance of the faked type.
>
> I wanted the best of both worlds and also I wanted to see what I could do with the syntax when I had absolutely free hands. Personally I (obviously) think I created something that is a good mix with the best from both world, but that's quite easy when you're standing on the shoulders of giants". (Hägne 2012.)

3.6.1 FakeItEasy installation via NuGet

FakeItEasy versions dated later than August 2011, requires installing NuGet Package Manager to Visual Studio (older versions can be downloaded and attached to the project like the Moq from http://code.google.com/p/fakeiteasy/downloads/list). The newer versions of FakeItEasy are installed and added to the project via the Nuget. (Nuget.org, 2013.) This operation is fairly easy and done as follow:

1. Download and install Nuget Package Manager from: http://visualstudiogallery.msdn.microsoft.com/27077b70-9dad-4c64-adcf-c7cf6bc9970c
2. Open the StubIsolationExample project in Visual Studio.
3. Add new test project (File > Add > New Project… > Visual C# / Test)
4. Open Nuget Package Manager Console from Tools > Library Package Manager > Package Manager Console (Figure 29).

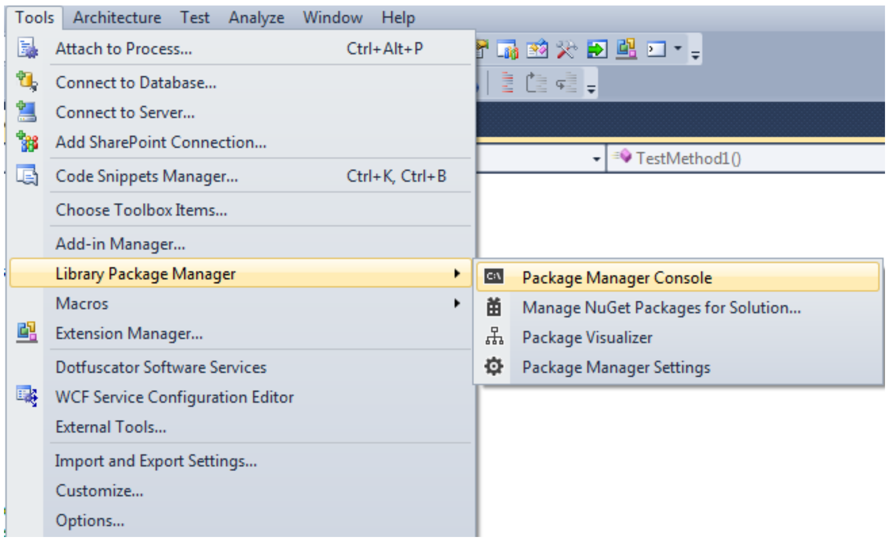
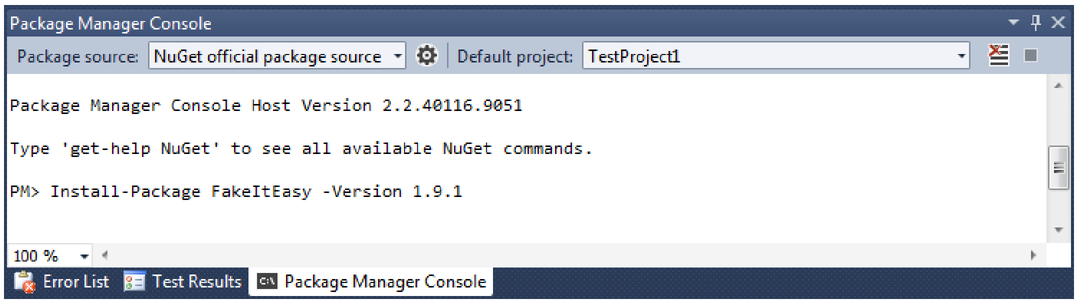

Tools Architecture Test Analyze Window Help
Attach to Process... Ctrl+Alt+P
Connect to Database...
Connect to Server...
Add SharePoint Connection...
Code Snippets Manager... Ctrl+K, Ctrl+B
Choose Toolbox Items...
Add-in Manager...
Library Package Manager
Macros
Extension Manager...
Dotfuscator Software Services
WCF Service Configuration Editor
External Tools...
Import and Export Settings...
Customize...
Options...
Package Manager Console
Manage NuGet Packages for Solution...
Package Visualizer
Package Manager Settings
TestMethod1()

Figure 29. Nuget Package Manager Console.

5. Check the latest version of FakeItEasy from https://www.nuget.org/packages/FakeItEasy/. For example the version 1.9.1 was newest stable version in March 6, 2013.
6. Select the Default project where you want to install FakeItEasy from the Package Manager Console and type following command: Install-Package FakeItEasy -Version 1.9.1, using the version you want to install and hit enter (Figure 30).


Package Manager Console
Package source: NuGet official package source
Default project: TestProject1
100 %
Error List
Test Results
Package Manager Console

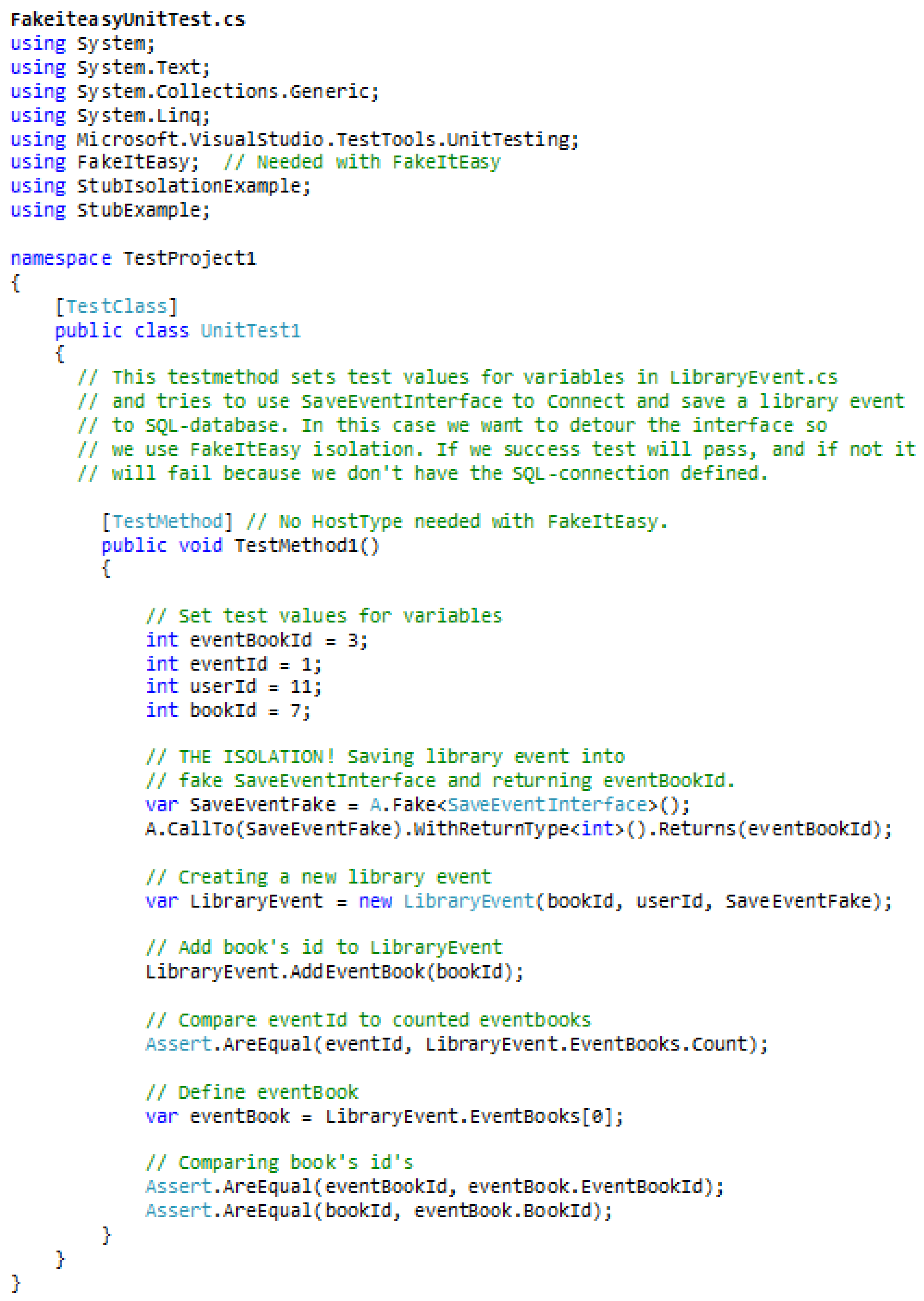
```
Package Manager Console Host Version 2.2.40116.9051

Type 'get-help NuGet' to see all available NuGet commands.

PM> Install-Package FakeItEasy -Version 1.9.1
```

Figure 30. Installing FakeItEasy package.

7. If the FakeItEasy was successfully installed, you should now have the FakeItEasy.dll reference in the project references.
8. Add a reference to our test project (StubIsolationExample). Right click on TestProject1 / References > Add Reference… > Projects (tab) > StubIsolationExample > OK.
9. Add using definition "using FakeItEasy;" on the top of your test class.

3.6.2 FakeItEasy example

An example unit test class for StubIsolationExample with FakeItEasy is below (figure 31). Using FakeItEasy a fake is created from SaveEventInterface and used like the real interface and the return value is set for EventBookId.



```
FakeiteasyUnitTest.cs
using System;
using System.Text;
using System.Collections.Generic;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using FakeItEasy;  // Needed with FakeItEasy
using StubIsolationExample;
using StubExample;

namespace TestProject1
{
    [TestClass]
    public class UnitTest1
    {
      // This testmethod sets test values for variables in LibraryEvent.cs
      // and tries to use SaveEventInterface to Connect and save a library event
      // to SQL-database. In this case we want to detour the interface so
      // we use FakeItEasy isolation. If we success test will pass, and if not it
      // will fail because we don't have the SQL-connection defined.

        [TestMethod] // No HostType needed with FakeItEasy.
        public void TestMethod1()
        {

            // Set test values for variables
            int eventBookId = 3;
            int eventId = 1;
            int userId = 11;
            int bookId = 7;

            // THE ISOLATION! Saving library event into
            // fake SaveEventInterface and returning eventBookId.
            var SaveEventFake = A.Fake<SaveEventInterface>();
            A.CallTo(SaveEventFake).WithReturnType<int>().Returns(eventBookId);

            // Creating a new library event
            var LibraryEvent = new LibraryEvent(bookId, userId, SaveEventFake);

            // Add book's id to LibraryEvent
            LibraryEvent.AddEventBook(bookId);

            // Compare eventId to counted eventbooks
            Assert.AreEqual(eventId, LibraryEvent.EventBooks.Count);

            // Define eventBook
            var eventBook = LibraryEvent.EventBooks[0];

            // Comparing book's id's
            Assert.AreEqual(eventBookId, eventBook.EventBookId);
            Assert.AreEqual(bookId, eventBook.BookId);
        }
    }
}
```

Figure 31. FakeItEasy unit test class.

4 RESULTS

All Isolation frameworks with examples in this thesis are useful and workable and the installation or attachment to the actual project is relatively easy and basic use is simple. At the beginning of studying isolation in unit testing, Moles framework was mainly used, and it seemed to be effective and after learning the basic calls, also quite straightforward. Moles also had good documentation, support and implementation (due to the support of big commercial firm Microsoft), remarkably better than open source frameworks.

After discovering the change from Moles to Fakes, studying Fakes started which was available preinstalled in Visual Studio 2012 RC. Yet later it became known that Fakes is also a dead end, because it will be available only in Visual Studio 2012 Ultimate version, which will not be in the use of an ordinary coder due to its expensive license. This made it clear that the only remaining and acceptable choice would be an open source framework. The chance to update the Fakes into the Premium version did not have significance to the results.

Moq is a handy tool and very easy to set up. However, during the unit testing for the ABB's new component a situation was often faced where the Moq's mock did not act as easily as expected and in the way the unit to be tested should have required. After several problem cases like this FakeItEasy started to gain attention.

FakeItEasy is as easy to install as Moq, and requires even less knowledge about isolation frameworks, stubs, mocks, etc. This is because there is only one test double type built in. It is true that with Moq you can easily set up mocks you want to setup to do whatever you need, especially with objects with read only values, but with FakeItEasy you simply work faster by writing fakes for dependencies you do not want to setup so precisely.

By this thesis the recommendation is to use both open source isolation frameworks Moq and FakeItEasy in unit testing, because together they complete each other and make it easier to write and run unit tests. Starting with FakeItEasy would be a good idea and to checking if it fulfills the needs of isolation and if it does not, at least it gives the user the basics of isolation in a simple form. Not later than then the developer is ready to use Moq or any other isolation framework, but at least Moq is a good way to continue.

A notable thing with isolation frameworks is that they develop, get better and advance rapidly and by the time you have learned to use one, there will be another new feature or even complete framework available. Completely different matter is when we will have an automated unit testing tool, that could be used in production. Still, the fact is that until then we do not have the choice to learn to do the testing ourselves.

5 SUMMARY

The basics of isolated unit tests in .Net environment were introduced in this thesis. In the beginning the importance of testing and the reasons why we do not test were discussed and the basic testing strategies and testing levels were connected to the common software development methods. After explaining how isolation is achieved in theory, we moved into isolation frameworks, where with coding examples the isolation was put into practice.

As a result we got information about testing as a part of software development and knowledge of isolated unit tests using these tools. This information is helpful for everyone interested in and related to software development and especially testing. This thesis taught a lot of theory and practice from the software development to actual unit testing code written by professional software developers.

The subject was challenging, because there is insufficient or incoherent information about the isolation available. There is wide range of literature and studies made about testing in general, but isolation seems to be quite anew field. The terminology of isolation techniques, for example, can vary and as one talks about a stub, he can actually mean a mock or something else. The situation will change while the techniques develop and become more common, but at the moment this thesis is a good way to get to know how isolation can be done in unit testing.

BIBLIOGRAPHY

ABB 2012a. [Online document]. [Ref. 15. November 2012]. ABB Ltd. Available at: http://new.abb.com/about/abb-in-brief/history

ABB 2012b. [Online document]. [Ref. 15. November 2012]. ABB Ltd. Available at: http://www.abb.fi/cawp/fiabb251/b23b1eb7a45bc7b3c2256b200045bd29.aspx

ABB 2012c. [Online document]. [Ref. 15. November 2012]. ABB Ltd. Available at: http://www.abb.fi/cawp/fiabb251/0b5e2755355c156dc12579bb003910a4.aspx

Agiledata, 2010.Introduction to Test Driven Development (TDD). [Web page]. [Ref 8. March 2013]. Available at: http://www.agiledata.org/essays/tdd.html

Aiia 2010. Acceptance testing. [Online publication]. Australian information industry association. [Ref 29. January 2013]. Available at: http://www.aiia.biz/legal/consulting/acceptance-testing

Almeida, G. 2007. Scrum, Test and Testers - Where are the relation?. [Online publication]. [Ref 15. March 2013]. Available at: http://www.gerson.se/Docs/ScrumTest_and_Testers.pdf

Burnstein, I. 2003. Practical software testing. New York: Springer-Verlag.

CastleProject. [Online document]. [Ref. 15. November 2012]. Available at:http://docs.castleproject.org/Default.aspx?Page=DynamicProxy&NS=Tools&AspxAutoDetectCookieSupport=1

Cohn, M. 2012. What is Scrum Methodology? .[Online publication]. [Ref 1. March 2013]. Available at:http://www.mountaingoatsoftware.com/topics/scrum

Erenthika, D. 2012. Gray box testing. [Online document]. [Ref 27. February 2013]. Available at: http://www.slideshare.net/dasuner/gray-box-testing

Extremeprogramming, 2009. [Online document]. [Ref 8. March 2013]. Available at: http://www.extremeprogramming.org/

FakeItEasy, 2012. [Online document]. [Ref. 15. November 2012]. Available at: https://github.com/FakeItEasy/FakeItEasy/wiki/Why-was-FakeItEasy-created%3F

FakeItEasy, 2013. [Online document]. Licence [Ref 20. March 2013]. Available at: https://github.com/FakeItEasy/FakeItEasy/blob/master/License.txt

Farrell-Vinay, P. 2008. Manage Software testing. New York: Auerbach Publications.

Ghahrai, A. 2008.V Model. [Online document]. TestingExcellence.com. [Ref. 17. December 2012]. Available at: http://www.testingexcellence.com/v-model/

Haikala, I. & Mikkonen, T. 2011. Ohjelmistotuotannon käytännöt. Helsinki: Talentum.

Harry, B. 2013. Announcing Visual Studio 2012 Update 2 (VS2012.2). [Online document]. [Ref. 15. March 2013]. Available at: http://blogs.msdn.com/b/bharry/archive/2013/01/30/announcing-visual-studio-2012-update-2-vs2012-2.aspx

Hägne, P. 2012. [Online article]. [Ref. 5 December 2012]. Available at: https://github.com/FakeItEasy/FakeItEasy/wiki/Why-was-FakeItEasy-created%3F

Hägne, P. 2011. [Online document]. [Ref. 7. March 2013]. Available at: http://stackoverflow.com/questions/7801212/what-is-a-dummy-used-for-in-fakeiteasy/7808294#7808294

Lehto, T. 2013. "Ketterä testaus keventää: Sulautettu tietotekniikka korvaa vauhdilla tuotannon analogisia järjestelmiä. Tämä vaatii luotettavaa ohjelmistotestausta". 3T: Tuotanto, Talous, Työelämä 5 (2013), 12-13.

Liversidge, E. 2005. The Death of the V-model. [Online document]. [Ref. 17. December 2012]. Available at: http://www.harmonicss.co.uk/index.php/hss-downloads/doc_download/12-death-of-the-v-model

Meszaros, G. 2009. Test Double. [Web site]. [Ref 1. February 2013]. Available at: http://xunitpatterns.com/Test%20Double.html

Microsoft 2013. Capabilities comparison of Visual Studio versions. [Online document]. Microsoft Co. [Ref. 21. January 2013]. Available at: http://www.microsoft.com/visualstudio/eng/products/compare

Microsoft research 2010a. Unit Testing with Microsoft Moles. [Online publication]. Microsoft Co. [Ref. 15. October 2012]. Available at: http://research.microsoft.com/en-us/projects/pex/molestutorial.docx

Microsoft research 2010b. Getting started with Microsoft Pex and Moles. [Online publication]. Microsoft Co. [Ref. 6. February 2013]. Available at: http://research.microsoft.com/en-us/projects/pex/getstarted.pdf

Moq [Online document]. [Ref. 15. November 2012]. Available at: http://code.google.com/p/moq/

MSDN. Overview of the .NET Framework. [Online document]. Microsoft Co.[Ref 18. February .2013]. Available at: http://msdn.microsoft.com/en-us/library/zw4w595w.aspx

MSDN, 2012a. Visual C#. [Online document]. Microsoft Co. [Ref 12. March 2013]. Available at: http://msdn.microsoft.com/en-us/library/kx37x362.aspx

MSDN, 2012b. Isolating Code under Test with Microsoft Fakes. [Online document]. Microsoft Co. [Ref. 26. March 2013]. Available at: http://msdn.microsoft.com/en-us/library/hh549175.aspx#shims

Myers, G. J. 2004. The Art of Software Testing.Second Edition. Hoboken: Wiley.

NIST 2002. [Online article]. [Ref. 5. December 2012]. Available at: http://www.abeacha.com/NIST_press_release_bugs_cost.htm

NSubstitute. [Web site]. [Ref 20. March 2013]. Available at: http://nsubstitute.github.com/

Nuget.org, 2011. Moq. [Web page]. [Ref 5. March 2013]. Available at: http://nuget.org/packages/moq

Nuget.org, 2013. Fake It Easy!. [Web page]. [Ref 5. March 2013]. Available at: https://www.nuget.org/packages/FakeItEasy

Osherove, R. 2009. The Art of Unit Testing.Greenwich: Manning Publications Co.

Osherove, R. 2010. 2010 Poll: Which isolation framework do you use in .NET?. [Online publication]. [Ref 5. February 2013]. Available at: http://osherove.com/blog/2010/9/10/2010-poll-which-isolation-framework-do-you-use-in-net.html

Osherove, R. 2012. 2012 Poll: Which isolation framework do you use in .NET?. [Online publication]. [Ref 5. February 2013]. Available at: http://osherove.com/blog/2012/5/4/annual-poll-which-isolation-framework-do-you-use-if-any.html

Pezzè, M. & Young, M. 2008. Software testing and analysis: Process, Principles, and techniques.Hoboken: Wiley.

Poimala, S. & Tolvanen, P. 2011. Ketteryys haltuun. [Online publication]. [Ref 18. March 2013]. Available at:http://www.meteoriitti.com/fi-FI/tiedotteet/ajankohtaista/ketteryys-haltuun-ketteran-kehityksen-yleiset-periaatteet

Prebble, C. 2008. What It Means to Mock: Isolating Units for Testing. [Online Document]. [Ref 29. January 2013]. Available at: http://www.javaranch.com/journal/2008/04/what-it-means-to-mock.html

SBP tech blog, 2012. Gray-box: the bridge between black-box and white-box testing [Online document]. [Ref. 16. January 2013]. Available at: http://www.sbp-romania.com/Blog/2012/05/14/gray-box-the-bridge-between-black-box-and-white-box-testing.aspx

Softwaretestingclass. Gray box testing. [Online document]. [Ref. 16. January 2013]. Available at: http://www.softwaretestingclass.com/gray-box-testing/

Softwaretestingfundamentals, 2010. Gray box testing. [Online document]. [Ref. 16. January 2013]. Available at: http://softwaretestingfundamentals.com/gray-box-testing/

System-testing. [Online document]. System-testing. [Ref 28. January 2013]. Available at: http://www.guru99.com/system-testing.html

Tuomikoski, J. 2009. Testing in Scrum. [Online document]. [Ref 1. March 2013]. Available at: http://www.tol.oulu.fi/users/ilkka.tervonen/Ote_vierailu_09.pdf

Typemock 2012. Isolator v7 pricing.[Web page].[Ref. 11 December 2012]. Available at: http://www.typemock.com/pricing

University of Cambridge, 2013. "Research by Cambridge MBAs for tech firm Undo finds software bugs cost the industry $316 billion a year". [Online document]. [Ref 27. February 2013]. Available at: https://www.jbs.cam.ac.uk/media/2013/research-by-cambridge-mbas-for-tech-firm-undo-finds-software-bugs-cost-the-industry-316-billion-a-year/

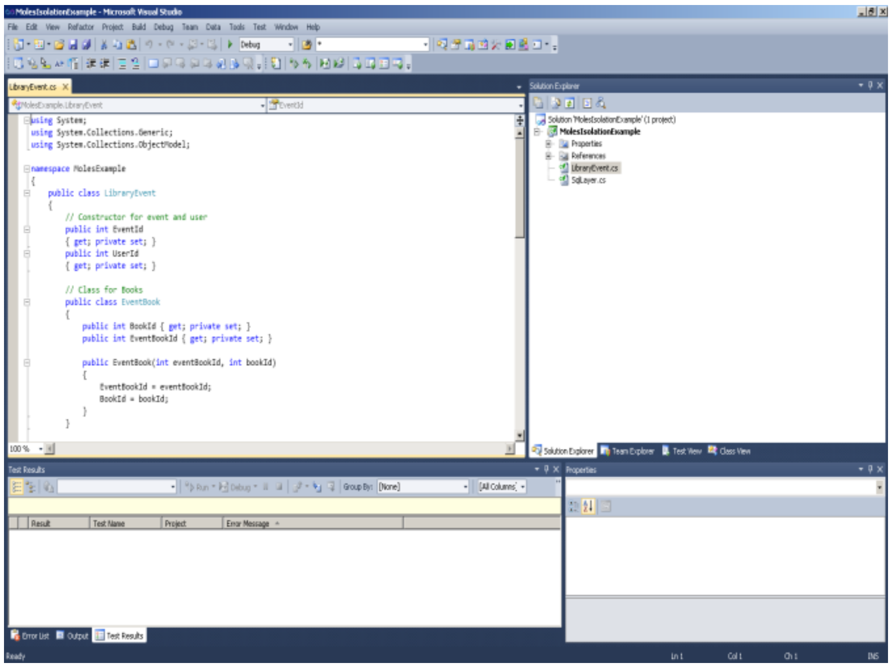
Visual Studio User Voice, 2013. [Online document]. [Ref. 15. March 2013]. Available at: http://visualstudio.uservoice.com/forums/121579-visual-studio/suggestions/2919309-provide-microsoft-fakes-with-all-visual-studio-edi

Waterfall-model 2012 [Online document]. [Ref. 11. December 2012]. Available at: http://www.waterfall-model.com/v-model-waterfall-model/

APPENDICES

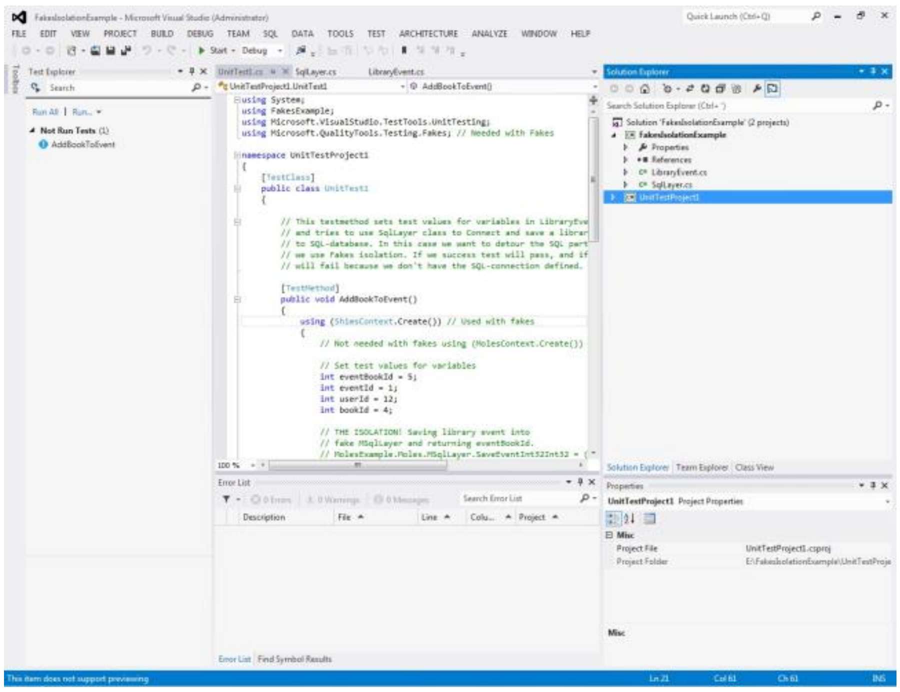
APPENDIX 1. Differences with Fakes and Moles examples

Main view



```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;

namespace MolesExample
{
    public class LibraryEvent
    {
        // Constructor for event and user
        public int EventId
        { get; private set; }
        public int UserId
        { get; private set; }

        // Class for Books
        public class EventBook
        {
            public int BookId { get; private set; }
            public int EventBookId { get; private set; }

            public EventBook(int eventBookId, int bookId)
            {
                EventBookId = eventBookId;
                BookId = bookId;
            }
        }
```

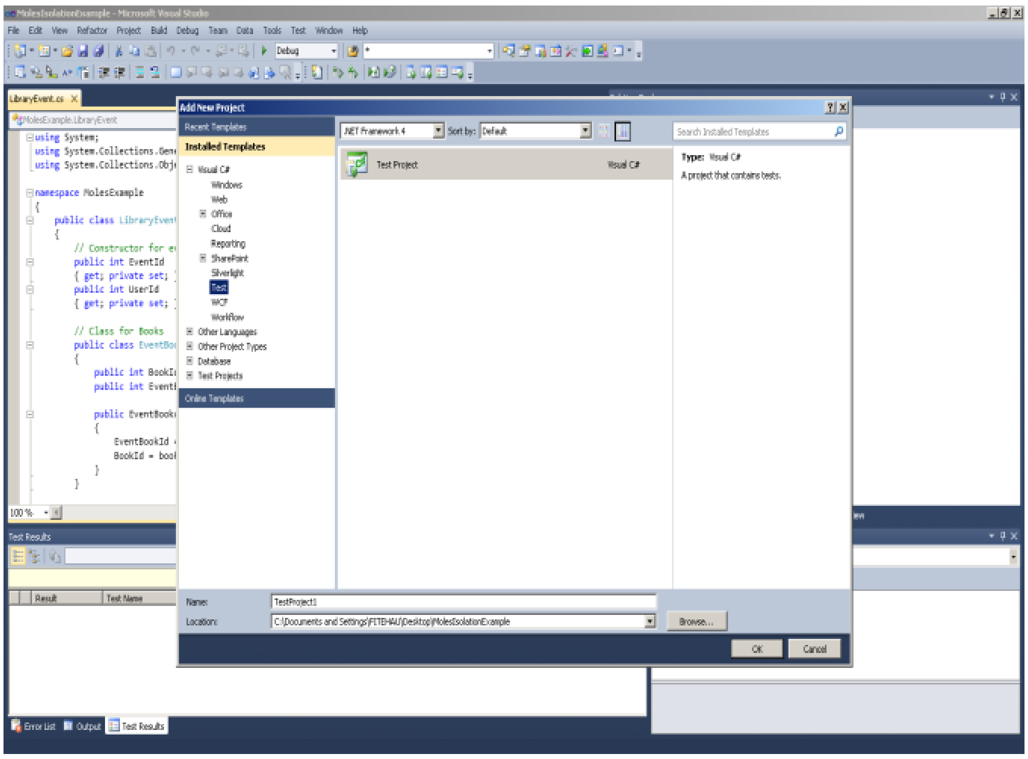
Figure 32. Main view in Visual Studio 2010.



```
using System;
using FakesExample;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Microsoft.QualityTools.Testing.Fakes; // Needed with Fakes

namespace UnitTestProject1
{
    [TestClass]
    public class UnitTest1
    {

        // This testmethod sets test values for variables in LibraryEve
        // and tries to use SqlLayer class to Connect and save a librar
        // to SQL-database. In this case we want to detour the SQL part
        // we use Fakes isolation. If we success test will pass, and if
        // will fail because we don't have the SQL-connection defined.

        [TestMethod]
        public void AddBookToEvent()
        {
            using (ShimsContext.Create()) // Used with fakes
            {
                // Not needed with fakes using (MolesContext.Create())

                // Set test values for variables
                int eventBookId = 5;
                int eventId = 1;
                int userId = 12;
                int bookId = 4;

                // THE ISOLATION! Saving library event into
                // fake MSqlLayer and returning eventBookId.
                // MolesExample.Moles.MSqlLayer.SaveEventInt32Int32 = (
```

Figure 33. Main view in Visual Studio 2012 RC.

Adding unit test project



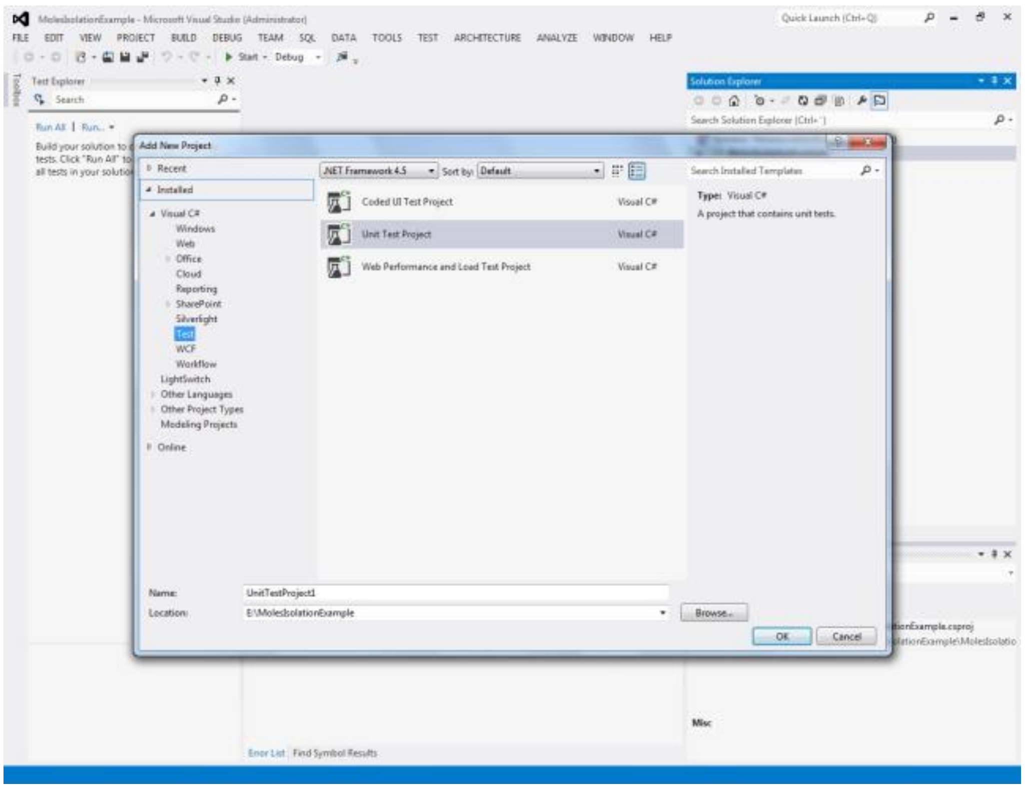
```
using System;
using System.Collections.Gene
using System.Collections.Obje

namespace MolesExample
{
    public class LibraryEvent
    {
        // Constructor for e
        public int EventId
        { get; private set; }
        public int UserId
        { get; private set; }

        // Class for Books
        public class EventBo
        {
            public int BookI
            public int EventI

            public EventBook
            {
                EventBookId =
                BookId = boo
            }
        }
```

Add New Project: .NET Framework 4; Sort by: Default; Test Project – Visual C#; Type: Visual C#; A project that contains tests.

Name: TestProject1

Location: C:\Documents and Settings\F1TEHAU\Desktop\MolesIsolationExample

Figure 34. Adding unit test project in Visual Studio 2010.



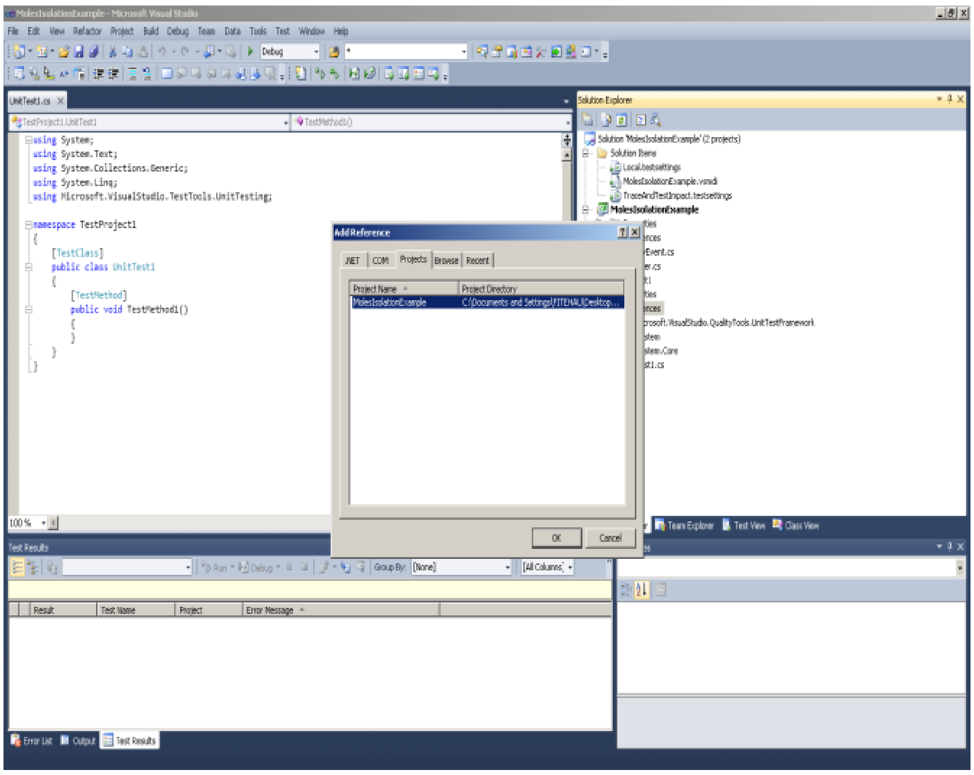
Add New Project: .NET Framework 4.5; Sort by: Default; Coded UI Test Project – Visual C#; Unit Test Project – Visual C#; Web Performance and Load Test Project – Visual C#; Type: Visual C#; A project that contains unit tests.

Name: UnitTestProject1

Location: E:\MolesIsolationExample

Figure 35. Adding unit test project in Visual Studio 2012 RC.

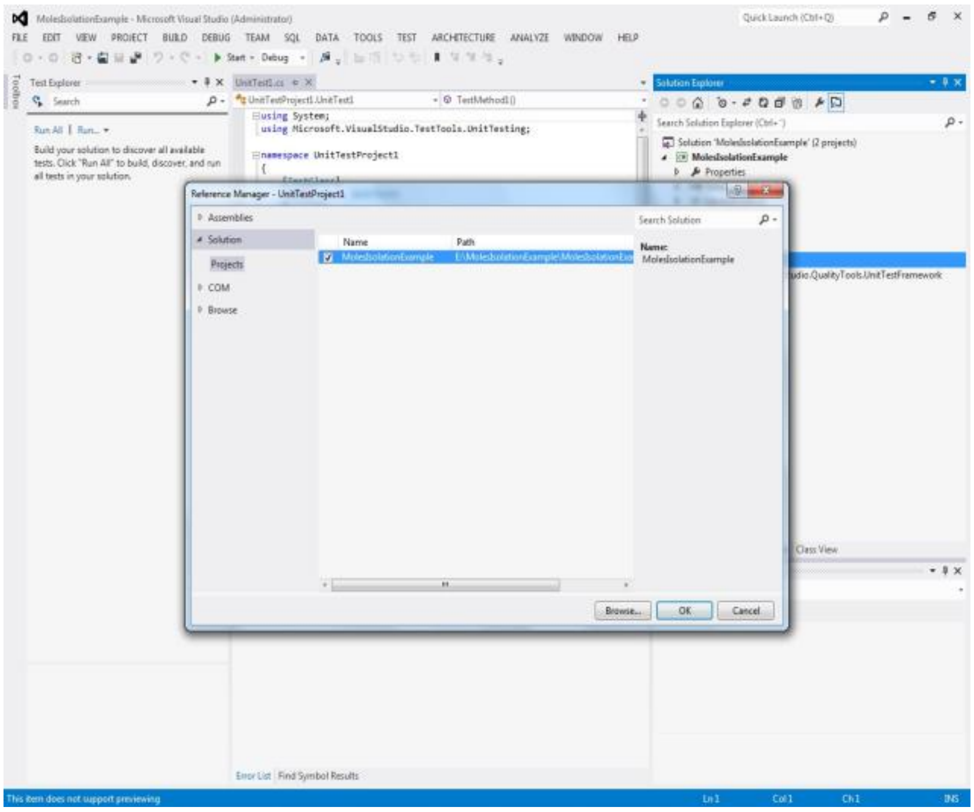
Adding reference



```
using System;
using System.Text;
using System.Collections.Generic;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace TestProject1
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

Project Name	Project Directory
MolesIsolationExample	C:\Documents and Settings\TITEHALI\Desktop...

Figure 36. Adding reference in Visual Studio 2010.



```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace UnitTestProject1
{
```

Name	Path
☑ MolesIsolationExample	E:\MolesIsolationExample\MolesIsolationEx

Name:
MolesIsolationExample

Figure 37. Adding reference in Visual Studio 2012 RC.

Run the tests

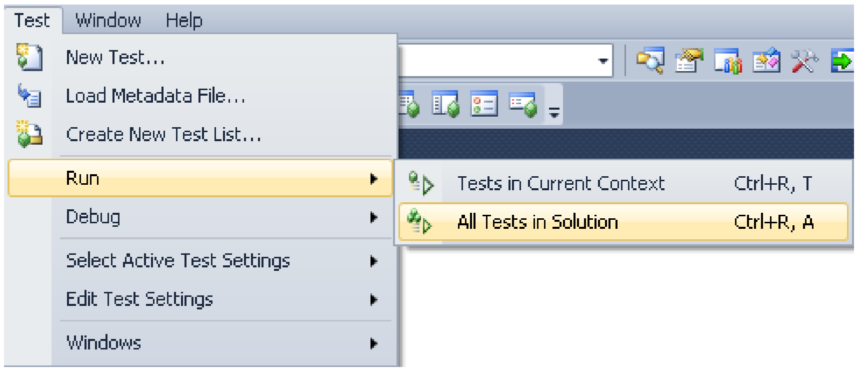


Figure 38. Run test in Visual Studio 2010.

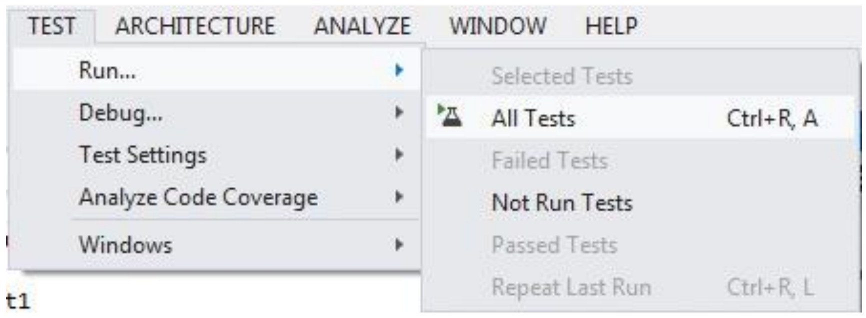
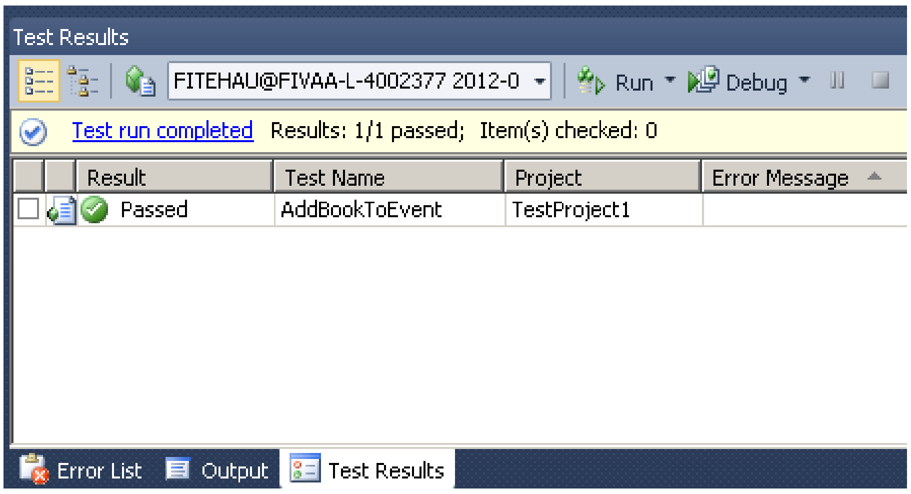


Figure 39. Run test in Visual Studio 2012 RC.

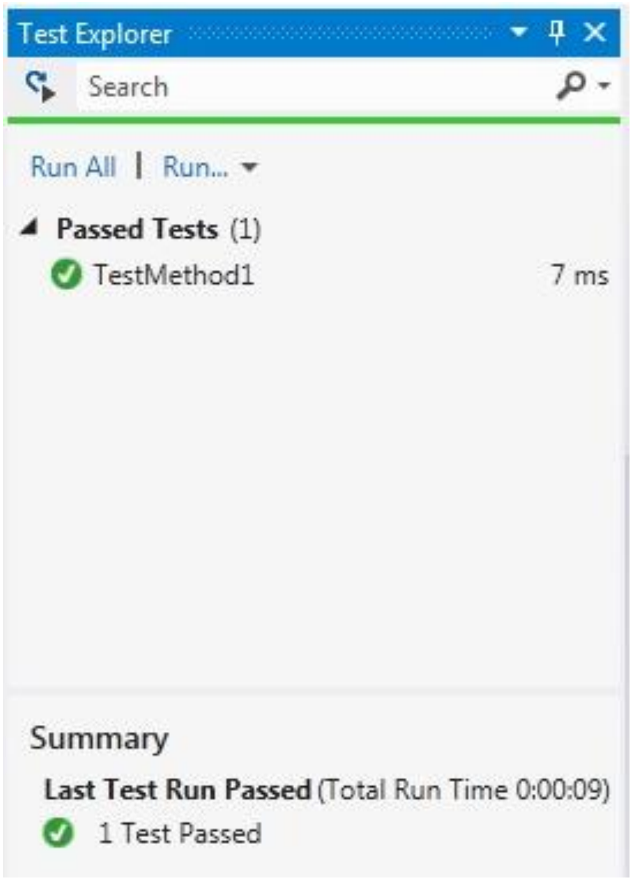
Test results



Test run completed Results: 1/1 passed; Item(s) checked: 0

Result	Test Name	Project	Error Message
Passed	AddBookToEvent	TestProject1	

Figure 40. Test results in Visual Studio 2010.



Passed Tests (1)
TestMethod1 7 ms

Summary
Last Test Run Passed (Total Run Time 0:00:09)
1 Test Passed

Figure 41. Test results in Visual Studio 2012 RC.

Adding Assembly

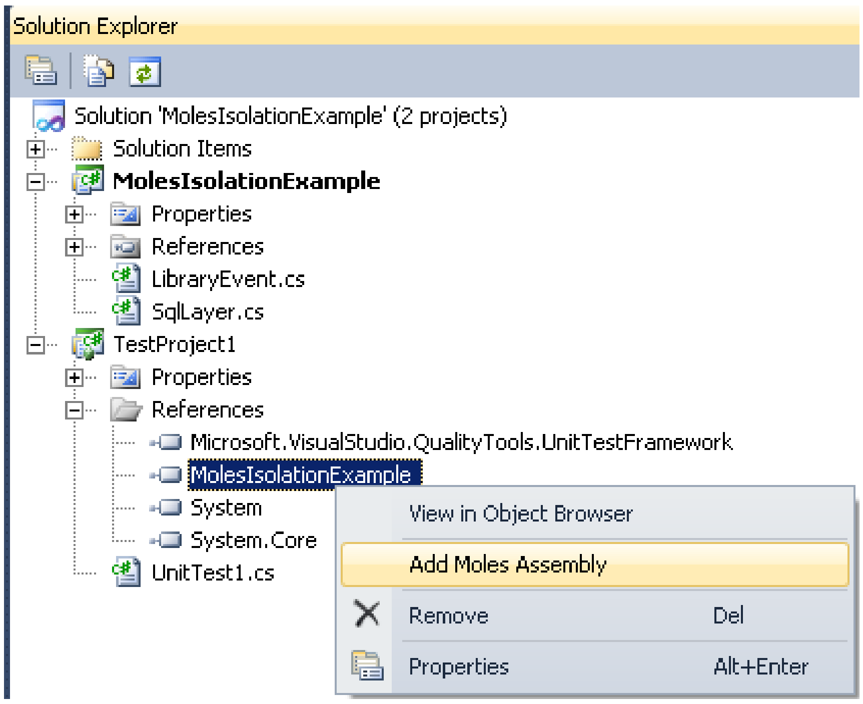


Figure 42. Adding assembly in Visual Studio 2010.

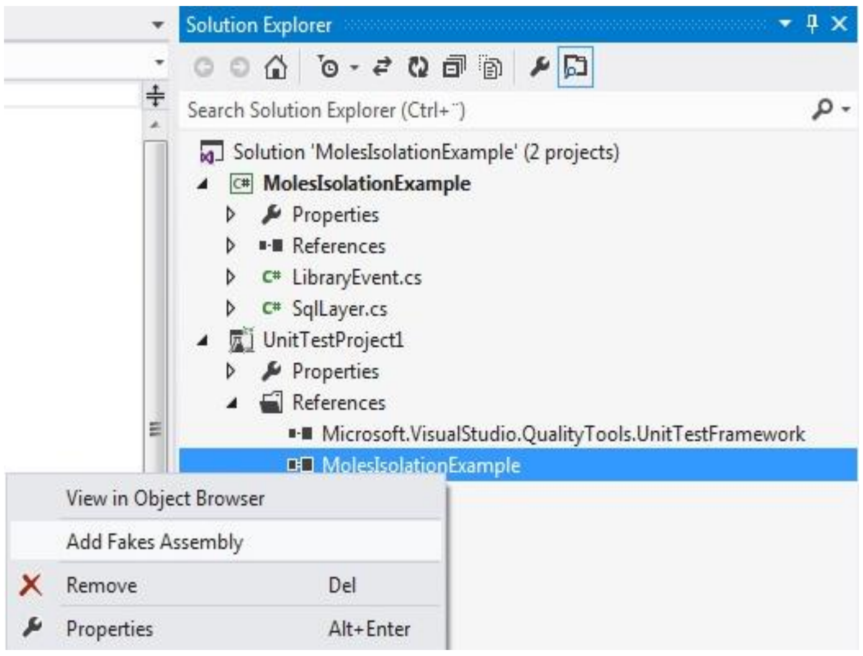
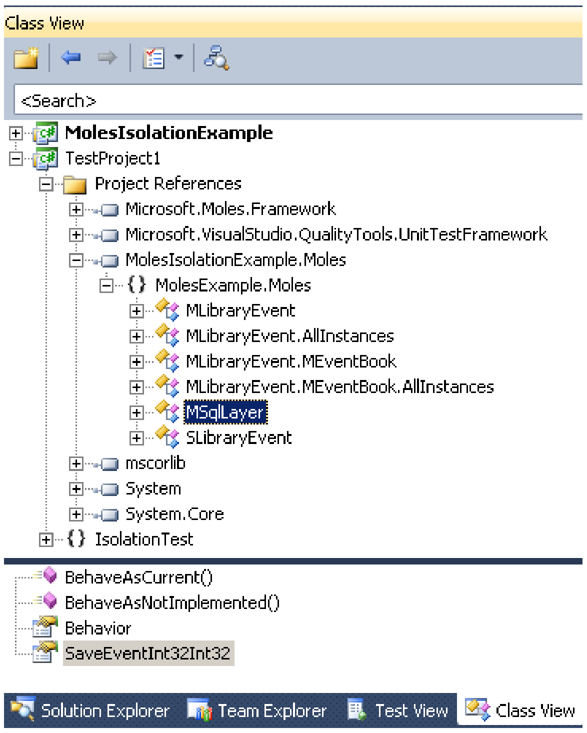


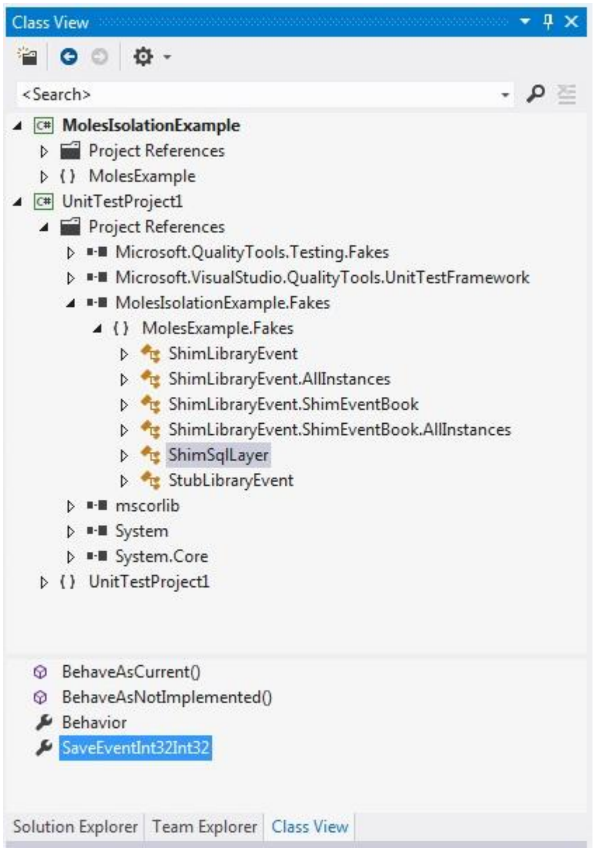
Figure 43. Adding assembly in Visual Studio 2012 RC.

Class View



```
Class View
<Search>
MolesIsolationExample
TestProject1
  Project References
    Microsoft.Moles.Framework
    Microsoft.VisualStudio.QualityTools.UnitTestFramework
    MolesIsolationExample.Moles
      MolesExample.Moles
        MLibraryEvent
        MLibraryEvent.AllInstances
        MLibraryEvent.MEventBook
        MLibraryEvent.MEventBook.AllInstances
        MSqlLayer
        SLibraryEvent
    mscorlib
    System
    System.Core
  IsolationTest

BehaveAsCurrent()
BehaveAsNotImplemented()
Behavior
SaveEventInt32Int32

Solution Explorer   Team Explorer   Test View   Class View
```

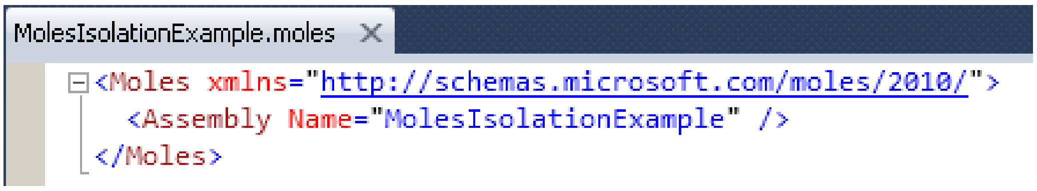
Figure 44. Class view in Visual Studio 2010.



```
Class View
<Search>
MolesIsolationExample
  Project References
  MolesExample
UnitTestProject1
  Project References
    Microsoft.QualityTools.Testing.Fakes
    Microsoft.VisualStudio.QualityTools.UnitTestFramework
    MolesIsolationExample.Fakes
      MolesExample.Fakes
        ShimLibraryEvent
        ShimLibraryEvent.AllInstances
        ShimLibraryEvent.ShimEventBook
        ShimLibraryEvent.ShimEventBook.AllInstances
        ShimSqlLayer
        StubLibraryEvent
    mscorlib
    System
    System.Core
  UnitTestProject1

BehaveAsCurrent()
BehaveAsNotImplemented()
Behavior
SaveEventInt32Int32

Solution Explorer   Team Explorer   Class View
```

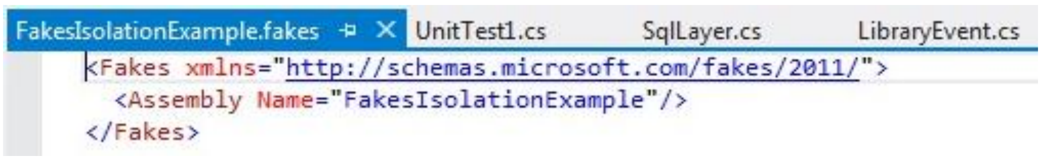
Figure 45. Class view in Visual Studio 2012 RC.

Assembly .xml file



```
<Moles xmlns="http://schemas.microsoft.com/moles/2010/">
  <Assembly Name="MolesIsolationExample" />
</Moles>
```

Figure 46. Assembly .xml-file in Visual Studio 2010.



```
<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/">
  <Assembly Name="FakesIsolationExample"/>
</Fakes>
```

Figure 47. Assembly .xml-file in Visual Studio 2012 RC.