TEST DRIVEN SOFTWARE DEVELOPMENT

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Software

by
Fatih ALGAN

June 2005
İZMİR

We approve the thesis of **Fatih ALGAN**

Date of Signature

……………………………. **18 October 2005**

Asst. Prof. Tuğkan Tuğlular
Supervisor
Department of Computer Engineering
İzmir Institute of Technology

……………………………. **18 October 2005**

Prof. Dr. Oğuz Dikeneli
Department of Computer Engineering
Ege University

……….. ………………….. **18 October 2005**

Asst. Prof. Bora Kumova
Department of Computer Engineering
İzmir Institute of Technology

……………………………… **18 October 2005**

Prof. Dr. Kayhan Erciyes
Head of Computer Engineering Department
İzmir Institute of Technology

………………………………

Assoc. Prof. Semahat ÖZDEMİR
Head of the Graduate School

ABSTRACT

This report aims at providing software developers with an understanding of the test driven software development practices and methods in the context of Java and J2EE “(WEB_19, 2005)” based technologies.

The study is presented in two main stages. The first stage describes the evolution of software verification and validation techniques and how they have been applied throughout the software development lifecycle, progressing upto **Agile Software Development** practices and **Test Driven Development**.

The second stage of the study presents a collection of patterns and methods for applying test driven development in real world scenarios. These techniques also aim to show how test driven development forces the classes and responsibilities to be decoupled and highly cohesive, leading to a beter object oriented design and high quality code.

ÖZET

Test Yönelimli Yazılım Geliştirme, yazılım geliştirmede çevik yaklaşımların ortaya çıkmasıyla beraber ortaya çıkmış bir programlama yöntemidir. Çevik yaklaşımların ardındaki temel fikir, yazılım geliştirme sürecini daha basit hale getirmek ve kısa ama devamlı iterasyonlarla, ve sistem kullanıcılarından yazılımın durumu hakkında devamlı geri besleme alabilecek şekilde organize etmektir.

Test Yönelimli Programlama ile karmaşık sistemler, küçük ve basit adımlardan oluşan artırımlarla geliştirilebilir. Yazılım sürekli olarak otomatize edilmiş testler ile sınanır, ve böylece yazılım üzerinde yapılan değişiklik ya da eklemelerden en az şekilde etkilenilir.

Test Yönelimli Programlama'nın başka bir anahtar noktası ise testlerin tamamen birbirinden izole olarak işletilebilmesi ilkesidir. Bu ilke sayesinde, testler yazılımın daha modüler ve birbirinden daha bağımsız parçalardan oluşacak şekilde geliştirilmesini yönlendirir.

Günümüzde yazılımlar nadiren kendi başlarına işlerler. Çoğu kurumsal uygulama veri tabanları, uygulama sunucuları, ağ altyapısı, işletim sistemleri ve bunların içine gömülmüş çok sayıda servisin sağladığı altyapı ile iletişim halinde ve bunlarla entegre çalışmak zorundadır. Ayrıca kurumsal uygulamalar birbirleriyle de entegre olmak ve iş süreçlerini beraber yürütmek zorundadır.

Değişik sistemlerin birbirlerinin sağladığı servislere böylesine ihtiyaç duyduğu bir ortamda bir yazılımın her bir küçük parçasını kendi başına ve izole olarak test edebilmek ancak bazı ileri test teknikleri ile mümkün olabilmekte, bazı platformlarda ise ancak çok yüksek maliyetlerle mümkün olabilmektedir.

Bu çalışmanın amacı Java ve J2EE platformlarında çalışan kurumsal bilgisayar yazılımlarının Test Yönelimli Yazılım Geliştirme Teknikleri ile geliştirilebilmesinin ne ölçüde mümkün olduğunu göstermek, ve bunun için çeşitli yöntemler sunmaktır.

TABLE OF CONTENTS

LIST OF FIGURES……………………………………………………………………...viii

LIST OF TABLES…………………………………………………………………...ix

CHAPTER 1 INTRODUCTION 10
1.1 Method and Approach.. 11

CHAPTER 2 INTRODUCTION TO SOFTWARE TESTING................................... 11
2.1 Software Verification and Validation 12
2.1.1 Static and Dynamic Verification:.. 12
2.2 Software Testing Process 12
2.2.1 White Box Testing 13
2.2.2 Black Box Testing... 13
2.3. Test Phases 14
2.4 Software Testing Process and Automated Tests 15
2.4.1 The Promise of Automated Testing 16

CHAPTER 3 AGILE SOFTWARE DEVELOPMENT 20
3.1 Agile Software Development Methodologies..................................... 22
3.2 Lean Development 23
3.3 Adaptive Software Development 24
3.4 Extreme Programming 25
3.4.1 Twelve Practices of Extreme Programming 26
3.5 Dynamic Systems Development Method... 27
3.6 Summary 28

CHAPTER 4 TEST DRIVEN DEVELOPMENT 29
4.1 Fundamental Concepts in Test Driven Development 29
4.2 Test Driven Development Methodology.. 30
4.3 Benefits of Test Driven Development.. 30
4.4 TDD and Software Requirements 32
4.5 TDD and Software Design 33
4.6 TDD and Software Maintenance.. 33
4.7 TDD and Software Documentation.. 33
4.8 Summary 34

CHAPTER 5 TEST DRIVEN DEVELOPMENT PRACTICES................................... 35
5.1 A Framework for Unit Testing... 35
5.1.1 Setting Up a Test Environment with JUnit 36
5.1.2 A Simple JUnit Test Case 36
5.1.3 The *TestCase* Class 36
5.1.4 Failure Messages.. 37
5.1.5 Signaling Failed Assertions.. 37
5.1.6 Differences Between Failures and Errors 38
5.2 Fundamental Test Driven Development Practices 38

5.2.1 Naming Test Case Classes 38
5.2.2 Testing Behavior Instead of Methods 39
5.2.3 Best Practices for Elementary Tests..40
5.2.3.1 Testing the Return Value of a Method...................................40
5.2.3.2 Testing Object Equality..41
5.2.3.3 Testing an Interface...42
5.2.3.4 Testing Throwing the Right Exception...................................44
5.2.4 Testing Design Patterns...45
5.2.4.1 Testing an Observer(Event Listener) 45
5.2.4.2 Testing an Observable(Event Source)....................................46
5.2.4.3 Testing a Singleton...48
5.2.4.4 Testing an Object Factory 48
5.2.4.5 Testing a Template Method 50
5.3 Testing in Dependent Environments...52
5.3.1 Testing and XML 53
5.3.1.1 Verifying the Order of Elements in an XML Document 55
5.3.1.2 Testing an XSL Stylesheet..56
5.3.2 Testing and Databases...58
5.3.2.1 A Persistency Framework 59
5.3.2.2 Testing Persistency Logic 62
5.3.3 Testing Enterprise Java Beans 64
5.3.3.1 Testing Stateless Session Beans..64
5.3.3.2 Testing Stateful Session Beans 65
5.3.3.3 Testing CMP Entity Beans..67
5.3.3.4 Testing Message Driven Beans...69
5.3.4 Testing J2EE Web Components...75
5.3.4.1 Testing the Session Data 75
5.3.4.2 Testing Processing a Request..79
5.3.4.3 Testing Servlet Initialization...84
5.3.4.3 Testing the ServletContext..85
5.3.4.4 Testing Rendering a JavaServer Page.....................................86
5.3.4.5 Testing Custom JSP Tags..90
5.3.5 Integration Testing 93
5.3.5.1 Testing Page Flow...93
5.3.5.2 Testing Navigation in a Struts Application.............................96
5.3.5.3 Testing Web Security..97
5.4 Summary 98

CHAPTER 6 IMPLEMENTATION OF TEST DRIVEN DEVELOPMENT 99
6.1 Requirements of the Ticket Seller Application...................................99
6.1.1 Functional Requirements of Ticket Seller: 99
6.1.1.1 User Story 1: Ticket Sale 100
6.1.1.2 User Story 2: Ticket Feedback..100
6.1.1.3 User Story 3: Feedback Approval...100
6.2 Preliminary Design of Ticket Seller..100
6.2.1 Persistence Layer of Ticket Seller..101
6.2.2 The Business Logic Layer of Ticket Seller..............................102
6.2.3 Presentation Layer of Ticket Seller..103
6.3 Implementation of the Persistence Layer..104
6.4 Implementing the Business Logic Layer...108

6.5 Implementing the Presentation Layer 117
6.6 Summary 123

CHAPTER 7 SOFTWARE METRICS AND TEST DRIVEN DEVELOPMENT 124
7.1 Classification of Software Metrics 124
7.1.1 Software Process Metrics and TDD 124
7.1.2 Software Product Metrics 125
7.1.2.1 Complexity Metrics 125
7.1.2.2Measurement over Time 127
7.2 Software Inspection 127
7.3 TDD Process Assessment 128
7.3.1 Retrospectives 128
7.3.2 TDD Specific Measurements 128
7.4 Evaluating Product Metrics of Ticket Seller Application 129
7.4.1 Setting up the Environment 130
7.4.2 Running the Tests 131
7.4.3 Analyzing the Metrics 131
7.5 Summary 135

CHAPTER 8 CONCLUSION 136

BIBLIOGRAPHY 138

LIST OF FIGURES

Figure **Page**

Figure 3.1 The Planning Spectrum……………………………………………...12

Figure 3.2 The ASD Lifecycle Phases…………………………………………16

Figure 5.1 The JUnit class Assert methods……………………………………..28

Figure 5.2 Proposed design to test a Message Driven Bean……………………….60

Figure 5.3 Refactoring design for tests…………………………………………68

Figure 6.1 The Architectural Layers Of Ticket Seller……………………………..91

Figure 6.2 The Persistence Layer Of Ticket Seller………………………………...92

Figure 6.3 The Business Logic Layer Of Ticket Seller……………………………93

Figure 6.4 The Presentation Layer Of Ticket Seller………………………………94

Figure 6.5 The Mock Persistence Layer Implementation…………………………101

Figure 6.6 High Level Architecture Of The Struts Framework…………………...108

Figure 6.7 Class Hierarchy For Presentation Layer Of The Ticket Seller………...109

Figure 6.8 Class Hierarchy For Presentation Layer Tests……………………….....111

Figure 7.1 First Test Coverage and Complexity Analysis Report………………....122

Figure 7.2 Test Coverage And Complexity Analysis Report of package edu.iyte.yemekhane.persistence…………………………………………………122

Figure 7.3 The uncovered parts of class PluginFactory……………………………123

Figure 7.4 The Uncovered Code parts that belongs to a simple data class………...124

Figure 7.5 Uncovered Code Belonging to class PersistenceManager……………...124

LIST OF TABLES

Table **Page**

Table 7.1 Cyclomatic Complexity Thresholds……………………………………120

Table 7.2 Complexity and Coverage analysis of each package…………………...125

CHAPTER 1

INTRODUCTION

The main concern of this study is on showing how test driven development can be applied in Java and J2EE "(WEB_19, 2005)" based projects.

The quality of a software is determined by how reliable the software is, how it satisfies the needs of the customer and how it can respond to continiously changing business requirements. Besides, most enterprise applications need to be interoperable with diverse technologies, integrate with legacy applications and scale up smoothly to be able to serve up increasing number of clients.

Automated software testing have made it possible to verify that a software satisfies the desired requirements before going into production. By automating the manual testing process, it becomes possible to simulate any number of clients that are working concurrently. Besides, it replaces the time-consuming and error prone manual testing process with a more manageable one. By running a suite of automated tests after a change to the software reveals which parts of the system are affected by the change and breaks.

Test driven developement is a new programming method in which testing, coding and refactoring activities all go in small iterative steps. Test driven development places the coding activity at the center task of software development.

The complex nature of software development and the need for responding to a dynamic, continiously changing business environment revealed the need for more dynamic and flexible software development methodologies than the traditional contract bound software development processes.

Agile software development methodologies emerged as a response to these demands of the software development community. The common values that all these methodologies put forth were individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation and responding to change over following a plan. Most of these methodologies needed a set of tools in order to manage the complex nature of software development with these new values; **automated software testing and refactoring** "(Fowler, 1999)".

Consequently, test driven development has become the de-facto programming method of agile software development processes.

This study investigates on best practices and pattterns for applying test driven development in Java and J2EE "(WEB_19, 2005)" based projects. A second goal of the study is to show how test driven development disciplines the programmer to write clean and well designed code in which responsibilities are seperated and the infrastructure is pluggable.

1.1 Method and Approach

This study aims to be a communicative guideline that will provide software developers an understanding of how to apply test driven development in object oriented enterprise applications. The study can be seen as a catalog of a set of best practices and patterns for unit testing Java and J2EE “(WEB_19, 2005)” based applications. Each pattern presents a different problem and discusses one or more alternatives that can be used to solve the problem.

The analysis of each problem ends with seperating the responsibilities of the unit to be tested. These responsibilities may be both functional or infrastructure related. The main focus is on being able to test each reponsibility seperately in order to minimize dependencies. Then a way to test each seperated responsibility is presented, with code examples to clarify the situation when needed. Some tools and frameworks that would help in simplifying the developer’s task for applying the solution are presented and explained with sample code.

Although unit testing emphasizes on seperation of responsibilities which must be tested in isolation, this seperation might not be always feasible. The study aims to identify such cases and justify why low granularity testing in isolation would not be the best solution in hand. In such cases, different tools or techniques are presented once again with example code.

CHAPTER 2

INTRODUCTION TO SOFTWARE TESTING

2.1 Software Verification and Validation

Verification and Validation can be defined as checking processes which ensure that software conforms to its specification and meets the needs of the customer "(Sommerville, 1996)". The system should be verified and validated at each stage of the software process, therefore it starts with requirements reviews and continues through design, code reviews and product testing.

Verification involves checking that the program conforms to its specification, which is usually defined during requirements analysis "(Sommerville, 1996)". *Validation* involves checking that the implemented program meets the expectations of the customer "(Sommerville, 1996)". Requirements validation techniques, such as prototyping, helps in this respect. However, flaws can sometimes only be discovered at later stages of the software process.

2.1.1 Static and Dynamic Verification:

To achieve the goals of verification and validation two different techniques of system checking and analysis can be performed. Static techniques involve the analysis and review of system model such as requirements documents, design diagrams, the program source code, etc. Dynamic techniques or tests involve exercising an implementation.

Static verification includes program inspections, analysis and formal verification. Thus, it can only check the correspondence between a program and its specification, it cannot demonstrate that the software is operationally useful. Consequently, program testing is the only way to prove how the software operates. Testing involves exercising the program using data like the real data processed by the software. The existence of program defects or inadequacies is derived from unexpected system outputs.

2.2 Software Testing Process

Software consists of complex interacting components, so it is hard and inefficient to be tested as a single, monolithic unit. In object oriented systems, operations and data are integrated to form objects and object classes. Groups of classes which act in combination forms services. At a higher level an object oriented system can be viewed as a collection of components collaborating in order to generate a response to a particular input or set of input events.

The testing process should inspect the system under question form two main aspects which satisfies the *verification* and *validation* of the system. **Structural Testing** or **White Box Testing** corresponds to *verification* while **Functional Testing** or **Black Box Testing** serves validation purposes.

2.2.1 White Box Testing

Structural tests verify the structure of the software itself and require complete access to the object's source code. This is known as ***white box*** testing because the internal workings of the code is inspected "(Sommerville, 1996)".

White-box tests make sure that the software structure itself contributes to proper and efficient program execution. Complicated loop structures, common data areas, 100,000 lines of spaghetti code and nests of "*if clauses"* are evil. Well-designed control structures, sub-routines and reusable modular programs are good.

Many defects can be identified and eliminated by applying classic structural test, the code inspection or walk-through. Code inspection is like proofreading, it can find the mistakes the author missed, the "typo's" and logic errors that even the best programmers can produce. Debuggers are typical white-box tools.

The strength of white-box testing is also its weakness. The code needs to be examined by highly skilled technicians which means that tools and skills are highly specialized to the particular language and environment. Also, large or distributed system execution goes beyond one program, so a correct procedure might call another program that provides bad data. In large systems, it is the execution path as defined by the program calls, their input and output and the structure of common files that is important. This gets into a hybrid kind of testing that is often employed in intermediate or integration stages of testing.

2.2.2 Black Box Testing

Functional tests examine the observable behavior of software as evidenced by its outputs without reference to internal functions, so, by analogy it is sometimes called ***black box*** testing "(Sommerville, 1996)". If the program consistently provides the desired features with acceptable performance, then specific source code features are irrelevant. It's a pragmatic and down-to-earth assessment of software.

Black box tests better address the modern programming paradigm. As object-oriented programming, automatic code generation and code reuse becomes more prevalent, analysis of source code itself becomes less important and functional tests become more important.

Black box tests also better attack the quality target. Since only the people paying for an application can determine if it meets their needs, it is an advantage to create the quality criteria from this point of view from the beginning.

Black box tests have a basis in the scientific method. Like the process of science, functional tests must have a hypothesis (specifications), a defined method or procedure (test), reproducible components (test data), and a standard notation to record the results. Black box tests can be re-run after a change to make sure the change only produced intended results with no inadvertent effects.

2.3. Test Phases

There are several type of testing in a comprehensive software test process, many of which occur simultaneously. They can be named as *unit testing, integration testing, system testing, stress testing, regression tests, quality assurance testing and user acceptence tests* which are explained below;

- **Unit Test:** In some organizations, a peer review panel performs the design and/or code inspections. Unit or component tests usually involve some combination of structural and functional tests by programmers in their own systems "(WEB_1, 2005)". Component tests often require building some kind of supporting framework that allow components to execute.

- **Integration Test:** The individual components are combined with other components to make sure that necessary communications, links and data sharing occur properly. It is not truly system testing because the components are not implemented in the operating environment "(WEB_1, 2005)". The integration phase requires more planning and some reasonable sub-set of production-type data. Larger systems often require several integration steps. There are three basic integration test methods:

  All At Once: The all-at-once method provides a useful solution for simple integration problems, involving a small program possibly using a few previously tested modules.

  Bottom Up: Bottom-up testing involves individual testing of each module using a driver routine that calls the module and provides it with needed resources. Bottom-up testing often works well in less structured shops because there is less dependency on availability of other resources to accomplish the test. It is a more intuitive approach to testing that also usually finds errors in critical routines earlier than the top-down method. However, in a new system many modules must be integrated to produce system-level behavior, thus interface errors surface late in the process.

  Top Down: Top-down testing fits a prototyping environment that establishes an initial skeleton that fills individual modules are completed. The method lends itself to more structured organizations that plan out the entire test process. Although interface errors are found earlier, errors in critical low-level modules can be found later than required. What all this implies is that a combination of low-level bottom-up testing works best for critical modules, while high-level top-down modules provide an early working program that can give management and users more confidence in results early on in the process. There may be need for more than one set of integration environments to support this hybrid approach.

System Test: The system test phase begins once modules are integrated enough to perform tests in a whole system environment. System testing can occur in parallel with integration test, especially with the top-down method [2].

Performance / Stress Test: An important phase of the system test, often called load, volume or performance test. Stress tests try to determine the failure point of a system under extreme pressure. Stress tests are most useful when systems are being scaled up to larger environments or being implemented for the first time. Web sites, like any other large-scale system that requires multiple access and processing, contain vulnerable nodes that should be tested before deployment. Unfortunately, most stress testing can only simulate loads on various points of the system and cannot truly stress the entire network as the users would experience it. Fortunately, once stress and load factors have been successfully overcome, it is only necessary to stress test again if major changes take place. A drawback of performance testing is that can easily confirm that the system can handle heavy loads, but cannot so easily determine if the system is producing the correct information. In other words, processing incorrect transactions at high speed can cause much more damage and liability than simply stopping or slowing the processing of correct transactions.

Regression Test: Regression tests confirm that implementation of changes have not adversely affected other functions. Regression testing is a type of test as opposed to a phase in testing. Regression tests apply at all phases whenever a change is made.

Quality Assurance Test: Some organizations maintain a Quality Group that provides a different point of view, uses a different set of tests, and applies the tests in a different, more complete test environment. The group might look to see that organization standards have been followed in the specification, coding and documentation of the software. They might check to see that the original requirement is documented, verify that the software properly implements the required functions, and see that everything is ready for the users to take a crack at it.

User Acceptance Test and Installation Test: Traditionally, this is where the users 'get their first crack' at the software. Unfortunately, by this time, it's usually too late. If the users have not seen prototypes, been involved with the design, and understood the evolution of the system, they are inevitably going to be unhappy with the result. If every test can be performed as user acceptance tests, there will be a much better chance of a successful project.

2.4 Software Testing Process and Automated Tests

For a long time, software testing has been conducted manually; that is, a human tester runs the application using predefined processes. Since the beginning of the software industry, software engineers have made great efforts to automate the software testing process. Many successful companies have manufactured software test tools that are now on the market. Today, there are many commercial and open source software tools that can be used to find bugs so that they can be fixed prior to the product release "(WEB_2, 2005)". Some of these tools are intended for designing and executing test scenarios while some are for the purpose of defect tracking and test management.

Automated Testing is automating the *manual* testing process currently in use. The principle of automated testing is that there is a program (which could be a job stream) that runs the program being tested, feeding it the proper input, and checking the output against the output that was expected "(ςολοκη, 1990) . Once the test suite is written, no human intervention is needed, either to run the program or to look to see if it worked; the test suite does all that, and indicates whether the program's output was as expected. This requires that a formalized *manual testing process* currently exists in a company or organization. Minimally, such a process includes:

- Detailed test cases, including predictable *expected results*, which have been developed from business functional specifications and design documentation

- A standalone test environment, including a test database that is restorable to a known constant, such that the test cases are able to be repeated each time there are modifications made to the application.

The real use and purpose of automated test tools is to automate *regression testing*. This means that a database of *detailed* test cases that are *repeatable* must be developed and maintained, and this suite of tests is run every time there is a change to the application to ensure that the change does not produce unintended consequences.

2.4.1 The Promise of Automated Testing

Test automation can enable some testing tasks to be performed far more efficiently than could ever be achieved by testing manually. **Fewster and Graham** **"**(Fewster, Graham, 1999)" point out the advantages of automated testing over manual testing as listed below;

- Running existing (regression) tests on a new version of a program is perhaps the most obvious task, particularly in an environment where many programs are frequently modified. The effort involved in performing a set of regression tests should be minimal. Given that the tests already exist and have been automated to run on an earlier version of the program, it should be possible to select the tests and initiate their execution with just a few minutes of manual effort.

- A clear benefit of automation is the ability to run more tests in less time and therefore to make it possible to run them more often. This will lead to greater confidence in the system. Most people assume that they will run the same tests faster with automation. In fact they tend to run more tests, and those tests are run more often.

- Performing tests which would be difficult or impossible to do manually is possible. Attempting to perform a full-scale live test of an on line system with 200 users may be impossible, but the input from 200 users can be simulated using automated tests. By having end users define tests that can be replayed automatically, user scenario tests can be run at any time even by technical staff who do not understand the intricacies of the full business application. When testing manually, expected outcomes typically include the obvious things that

are visible to the tester. However, there are attributes that should be tested which are not easy to verify manually. For example a graphical user interface (GUI) object may trigger some event that does not produce any immediate output. A test execution tool may be able to check that the event has been, triggered, which would not be possible to check without using a tool.

- Better use of resources. Automating menial and boring tasks, such as repeatedly entering the same test inputs, gives greater accuracy as well as improved staff morale, and frees skilled testers to put more effort into designing better test cases to be run. There will always be some testing which is best done manually; the testers can do a better job of manual testing if there are far fewer tests to be run manually. Machines that would otherwise lie idle overnight or at the weekend can be used to run automated tests.

- Consistency and repeatability of tests. Tests that are repeated automatically will be repeated exactly every time (at least the inputs will be; the outputs may differ due to timing, for example). This gives a level of consistency to the tests which is very difficult to achieve manually. The same tests can be executed on different hardware configurations, using different operating systems, or using different databases. This gives a consistency of cross-platform quality for multi-platform products which is virtually impossible to achieve with manual testing. The imposition of a good automated testing regime can also insure consistent standards both in testing and in development. For example, the tool can check that the same type of feature has been implemented in the same way in every application or program.

- Reuse of tests. The effort put into deciding what to test, designing the tests, and building the tests can be distributed over many executions of those tests. Tests which will be reused are worth spending time on to make sure they are reliable. This is also true of manual tests, but an automated test would be reused many more times than the same test repeated manually.

- Earlier time to market. Once a set of tests has been automated, it can be repeated far more quickly than it would be manually, so the testing elapsed time can be shortened (subject to other factors such as availability of developers to fix defects).

- Increased confidence. Knowing that an extensive set of automated tests has run successfully, there can be greater confidence that there won't be any unpleasant surprises when the system is released (providing that the tests being run are good tests!). In summary, more thorough testing can be achieved with less effort, giving increases in both quality and productivity.

While promising significant advantages, **Fewster and Graham** “(Fewster, Graham, 1999)” also argue that test automation cannot solve all the problems related to testing and cannot completely replace manual testing. They indicate the common problems that can be faced with while applying test automation as below;

Unrealistic expectations

Software industry is known for latching onto any new technical solution and thinking it will solve all current problems. Testing tools are no exception. There is a tendency to be optimistic about what can be achieved by a new tool. It is human nature to hope that this solution will at last solve all of the problems that are currently being undergone through. Vendors naturally emphasize the benefits and successes, and may play down the amount of effort needed to achieve lasting benefits. The effect of optimism and salesmanship together is to encourage unrealistic expectations. If management expectations are unrealistic, then no matter how well the tool is implemented from a technical point of view, it will not meet expectations.

Poor testing practice

If testing practice is poor, with poorly organized tests, little or inconsistent documentation, and tests that are not very good at finding defects, automating testing is not a good idea. It is far better to improve the effectiveness of testing first than to improve the efficiency of poor testing.

Expectation that automated tests will find a lot of new defects

A test is most likely to find a defect the first time it is run. If a test has already run and passed, running the same test again is much less likely to find a new defect, unless the test is exercising code that has been changed or could be affected by a change made in a different part of the software, or is being run in a different environment. Test execution tools are 'replay' tools, i.e. regression testing tools. Their use is in repeating tests that have already run. This is a very useful thing to do, but it is not likely to find a large number of new defects, particularly when run in the same hardware and software environment as before. Tests that do not find defects are not worthless, even though good test design should be directed at trying to find defects. Knowing that a set of tests has passed again gives confidence that the software is still working as well as it was before, and that changes elsewhere have not had unforeseen effects.

False sense of security

Just because a test suite runs without finding any defects, it does not mean that there are no defects in the software. The tests may be incomplete, or may contain defects themselves. If the expected outcomes are incorrect, automated tests will simply preserve those defective results indefinitely.

Maintenance of automated tests:

When software is changed it is often necessary to update some, or even all, of the tests so they can be re-run successfully. This is particularly true for automated tests. Test maintenance effort has been the death of many test automation initiatives. When it takes more effort to update the tests than it would take to re-run those tests manually, test automation will be abandoned.

Technical problems:

Commercial test execution tools are software products, sold by vendor companies. As third-party software products, they are not immune from defects or problems of support. It is perhaps a double disappointment to find that a testing tool has not been well tested, but unfortunately, it does happen. Interoperability of the tool with

other software can be a serious problem. The technological environment changes so rapidly that it is hard for the vendors to keep up. Many tools have looked ideal on paper, but have simply failed to work in some environments. The commercial test execution tools are large and complex products, and detailed technical knowledge is required in order to gain the best from the tool. Training supplied by the vendor or distributor is essential for all those who will use the tool directly, particularly the test automa-tor(s) (the people who automate the tests). In addition to technical problems with the tools themselves, technical problems may be experienced with the software under test. If software is not designed and built with testability in mind, it can be very difficult to test, either manually or automatically. Trying to use tools to test such software is an added complication which can only make test automation even more difficult.

Organizational issues

Automating testing is not a trivial exercise, and it needs to be well supported by management and implemented into the culture of the organization. Time must be allocated for choosing tools, for training, for experimenting and learning what works best, and for promoting tool use within the organization. An automation effort is unlikely to be successful unless there is one person who is the focal point for the use of the tool, the tool 'champion.' Typically, the champion is a person who is excited about automating testing, and will communicate his or her enthusiasm within the company. This person may be involved in selecting what tool to buy, and will be very active in promoting its use internally. Test automation is an infrastructure issue, not just a project issue. In larger organizations, test automation can rarely be justified on the basis of a single project, since the project will bear all of the startup costs and teething problems and may reap little of the benefits. If the scope of test automation is only for one project, people will then be assigned to new projects, and the impetus will be lost. Test automation often falls into decay at precisely the time it could provide the most value, i.e. when the software is updated. Standards are needed to insure consistent ways of using the tools throughout the organization. Otherwise every group may develop different approaches to test automation, making it difficult to transfer or share automated tests and testers between groups. Even a seemingly minor administrative issue such as having too few licenses for the people who want to use the tool can seriously impact the success and the cost of a test automation effort.

Perceptions of work effort may also change. If a test is run overnight, then when the testers arrive in the morning, they will need to spend some time looking through the results of the tests. This test analysis time is now clearly visible as a separate activity. When those tests were run manually, this test analysis time was embedded in the test execution activity, and was therefore not visible. Whenever a new tool (or indeed any new process) is implemented, there are inevitably adjustments that need to be made to adapt to new ways of working, which must be managed.

CHAPTER 3

AGILE SOFTWARE DEVELOPMENT

Agile Software Development methods emerged from the needs of the business community asking for lightweight along with faster and nimbler software development processes. This is especially the case with the rapidly groving and volatile Internet software industry as well as for the emerging mobile application development. While no agreement on what the concept of **agile** actually refers to exists, it has generated significant interest among practitioners and academia. The introduction of the **extreme programming** method (beter known as **XP,** "(Beck, 1999)"), has been widely acknowledged as the starting point for the various agile software development approaches. There are also a number of other methods invented since then that appear to belong to the same family of methodologies.

The **Agile Movement** in software industry was born with the **Agile Software Development Manifesto "**(WEB_3, 2005)" published by a group of software practitioners and consultants in 2001. The key values annouced by this manifesto were;

Individuals and interactions over processes and tools.
Working software over comperehensive documentation.
Customer collaboration over contract negotiation.
Responding to change over following a plan.

These central values can be explained as follows:

Individuals and Interactions Over Processes and Tools: The agile movement emphasizes the relationship and communality of software developers and the human role reflected in the contracts, as opposed to institutionalized processes and development tools. In the existing agile practices, this manifests itself in close team relationships, close working environment arrangements, and other procedures boosting team spirit.

Working Software Over Comprehensive Documentation: The vital goal of the software team is to continiously turn out tested working software. New releases are produced at frequent intervals, in some approaches even hourly or daily, but more usually bi-monthly or monthly. The developers are urged to keep the code simple, straightforward, and technically as advanced as possible, thus lessening the documentation burden to an appropriate level.

Customer Collaboration Over Contract Negotiation: The relationship and cooperation between the developers and the clients is given the preference over strict contracts, although the importance of well drafted contracts does grow at the same pace as the size of the software project. The negotiation process itself should be seen as a means of achieving and maintaining a viable relationship. From a business point of view, agile development is focused on delivering business value immediately as the project starts, thus reducing the risks of non-fulfillment regarding the contract.

Responding to Change Over Following a Plan: The development group, comprising both software developers and customer representatives, should be well-informed, competent and authorized to consider possible adjustment needs emerging during the development process life-cycle. This means that the participants are prepared to make changes and that also the existing contracts are formed with tools that support and allow these enhancements to be made.

According to **Highsmith and Cockburn (**Highsmith, Cockburn, 2001) what is new about agile methods is not the practices they use, but their recognition of people as the primary drivers of project success, coupled with an intense focus on effectiveness and maneuverability. This yields a new combination of values and principles that define an **agile** world view. **Boehm** "(Boehm, Turner, 2003)" illustrates the spectrum of different planning methods with **Figure 3.1**, in which hackers are placed at one end and the so-called inch-pebble ironbound contractual approach at the opposite end.

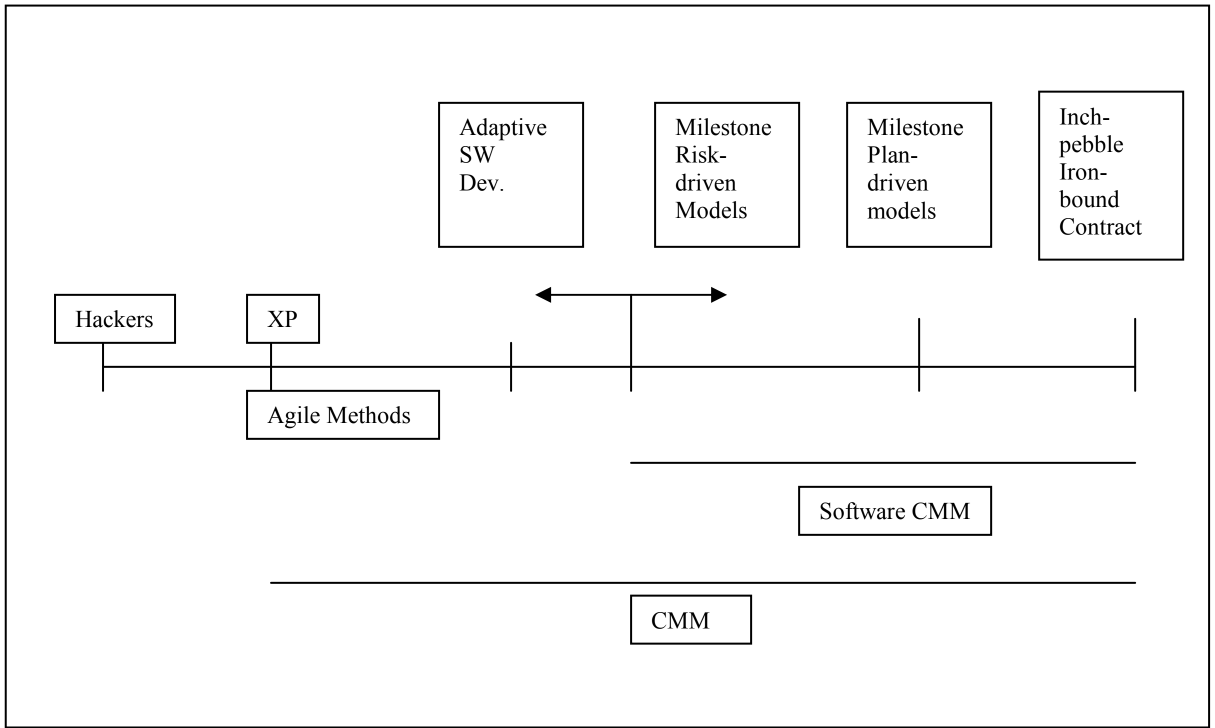

Figure 3.2 The Planning Spectrum. Unplanned and undisciplined hacking occupies the extreme left, while micromanaged milestone planning also known as inch-pebble planning, occupies the extreme right. "(Boehm, Turner, 2003)"

Boehm defines *planning* as "documented process procedures that involve tasks and milestone plans and product development strategies that involve requirements, designs and *architectural plans."* "(Boehm, Turner, 2003)". He differentiates agile methods from undisciplined hacking by emphisizing that agile disciplines employs a *fair* amount of planning and placing more value on planning than the resulting documentation. Besides, agile software development methods generally have criteria, as opposed to hacking, such as Extreme Programming's 12 practices "(Beck, 1999)".

Cockburn "(Cockburn, A)" defines the core of agile software development methods as the use of light but sufficient rules of project behavior and the use of human and communication oriented rules. The agile process is both light and sufficient. Lightness is a means of remaining maneuverable. Sufficiency is a matter of staying in

the game. He proposes the following *sweet spots,* the presence of which in software development work enhances the prospects for a successful project outcome:

Two to eight people in one room: Communication and community.
Onsite usage experts: Short and continious feedback cycles.
Short increments: One to three months, allows quick testing and repairing.
Fully automated regression tests: Unit and functional tests stabilize code and allow continious improvement.
Experienced developers: Experience speeds up the development time from 2 to 10 times compared to slower team members.

Agile development is not defined by a small set of practices and techniques. Agile development defines a strategic capability to create and respond to change, balance flexibility and structure, to draw creativity and innovation out of a development team, and leading organizations through rapidly changing environment and uncertainty.

Agile development does not neglect structure, but attempts to balance flexibility and structure trying to figure out the balance between chaos and rigidity. The greater the uncertainty, the faster the pace of change, and the lower the probability that success will be found in structure.

3.1 Agile Software Development Methodologies

There are a growing number of agile *methodologies*, and a number of agile practices such as **Scott Ambler's agile modeling** "(Ambler, 2002)"**.** The core set of these includes *Lean Development(LD)***,** *ASD, Scrum, eXtreme Programming (XP), Crystal methods, FDD, and DSDM*. Authors of all of these approaches (except LD) participated in writing the **Agile Software Development Manifesto** "(WEB_3, 2005)" and so its principles form a common bond among practitioners of these approaches. While individual practices are varied, **Highsmith "**(WEB_4, 2005)" states that they fall into six general categories :

- **Visioning:** A good visioning practice helps assure that agile projects remain focused on key business values (for example, ASD's product visioning session).

- **Project initiation:** A project's overall scope, objectives, constraints, clients, risks, etc. should be briefly documented (for example, ASD's one-page project data sheet).

- **Short, iterative, feature-driven, time-boxed development cycles:** Exploration should be done in definitive, customer-relevant chunks (for example, FDD's feature planning).

- **Constant feedback:** Exploratory processes require constant feedback to stay on track (for example, Scrum's short daily meetings and XP's pair programming).

- **Customer involvement:** Focusing on business value requires constant interaction between customers and developers (for example, DSDM's facilitated workshops and ASD's customer focus groups).

- **Technical excellence:** Creating and maintaining a technically excellent product makes a major contribution to creating business value today and in the future (for example, XP's refactoring). Some agile approaches focus more heavily on project management and collaboration practices (ASD, Scrum, and DSDM), while others such as XP focus on software development practices, although all the approaches touch the six key practice areas.

3.2 Lean Development

Lean software development has its roots from car manufacturing industry. In 1950, Toyota motor company began testing the product line at every station. They also reduced batch sizes so that cycle time was reduced, defects appeared quickly, and were corrected quickly. This prevented discovering defects near the end of the production line and having to re-work on many cars effected from the same defect. Lean manufacturing stressed on certain values such as minimizing the costs at the production line, empowering the people who added value and trying to stay away from rigid plans and procedures, and close partnership with customers. **Kumar** "(WEB_5, 2005)" explains these key values as follows when thay are applied to software development process,

- **Eliminate Waste:** Unneeded, extra features, large inventory of requirements, overly produced design and specification artifacts, bugs not caught or caught lately by tests, waiting for decisions(including customers) and document hand-offs are the major wastes in a software development process. Lean development stresses these wastes should be minimized.
- **Do It Right the First Time(Incorporate Feedback):** Lean development employs two key techniques that makes change easy. Test must be built at various stages of the development process. As development proceeds and changes are made, the unit and regression tests are run. If the tests don't pass, programming may be stopped until the problem is found and corrected. A comprehensive testing capability is the best way to accommodate change throughout the development process. The second technique that facilitates change is "refactoring", improving the design of existing software in a controlled and rapid manner. With refactoring, initial designs can focus on the basic issue at hand rather than speculate about other features that may be needed in the future. Later in the process, refactoring techniques can incorporate these additional features, as they are required, making it easy to accommodate the future if and when it becomes the present.
- **Empower Those Who Add Value:** Lean development gives priority to people and collaborating teams over paperwork and processes. It focuses on methods of forming and encouraging teams to address and resolve their own problems, recognizing that the people doing the work must determine the details. Software development involves the handoff of information at least once (from user to programmer) and often more than once (from user to designer to programmer).

Transferring all such information in writing leads to a great amount of tacit knowledge being lost by handing off information on paper. However, it can be far more effective to have small collaborating teams work across the boundaries of an information handoff, minimizing paperwork and maximizing communication.

- **Continiuos Improvement:** Iterative development and refactorings provide a continious improvement on the quality of the product as well as project and team organization for future projects. At each small iteration, problems regarding the process and product may be identified and responded accordingly.
- **Meeting Customer Requirements:** "Do it right the first time" principle the process should have the provision for the customer to make changes. The Software compliance and user (customer) acceptance testing must be done with reference to the customer requirements.
- **Pull From Demand:** In software development, the key to rapid delivery is to divide the problem into small batches (increments) pulled by a customer test. The single most effective mechanism for implementing lean production is adopting just-in-time, pull from demand flow. Similarly, the single most effective mechanism for implementing lean development is delivering increments of real business value in short time-boxes.
- **Maximizing Flow:** In lean software development, the idea is to maximize the flow of information and delivered value. In lean production, maximizing flow does not mean automation. Instead it means limiting what has to be transferred, and transferring that as few times as possible over the shortest distance with the widest communication bandwidth. Similarly in software development the idea is to eliminate as many documents and handoffs as possible. In the lean software development emphasis is to pair a skilled development team with a skilled customer team and to give them the responsibility and authority to develop the system in small, rapid increments, driven by the customer priority and feed back.
- **Ban Rigid Project Scope Management:** Holding the scope to exactly what was envisioned at the beginning of a project offers little value to the user whose world is changing. In fact, it causes anxiety and paralyzes decision-making, ensuring only that the final system will be outdated by the time it's delivered. Managing to a scope that's no longer valid wastes time and space, necessitating inefficient issue lists, extensive trade-off negotiations and multiple system fixes.

3.3 Adaptive Software Development

Adaptive Software Development focuses mainly on developing complex, large systems. The method strongly encourages incremental, iterative development with constant prototyping. An adaptive software development project is carried out in three phase cycles, *speculate, collaborate and learn* "(WEB_6, 2005)". Speculation is used instead of planning, since uncertainty and deviations are generally considered a weakness in planning. Collaborate stands for team-work for developing high-change systems. Learn stands for reacting to mistakes, and the fact that requirements may change during development.

Adaptive development cycles are illustrated in more detail in **Figure 3.2** **"**(WEB_6, 2005)". The Project initiation phase defines the project mission and cornerstones of the project. The Project initiation phase determines defines overall

schedule, as well as schedules and objectives fort he development cycles. The cycles typically lasts between four and eight weeks.

Adaptive Software Development is explicitly component oriented rather than task oriented. This means that the focus is more on the results and their quality rather than the tasks or the process used for producing the result. This practice is applied through the adaptive development cycles that contain the collaborate phase where several components may be developed concurrently. Planning the cycles is a part of the iterative process, as the definitions of the components are continiously refined to reflect any new information.

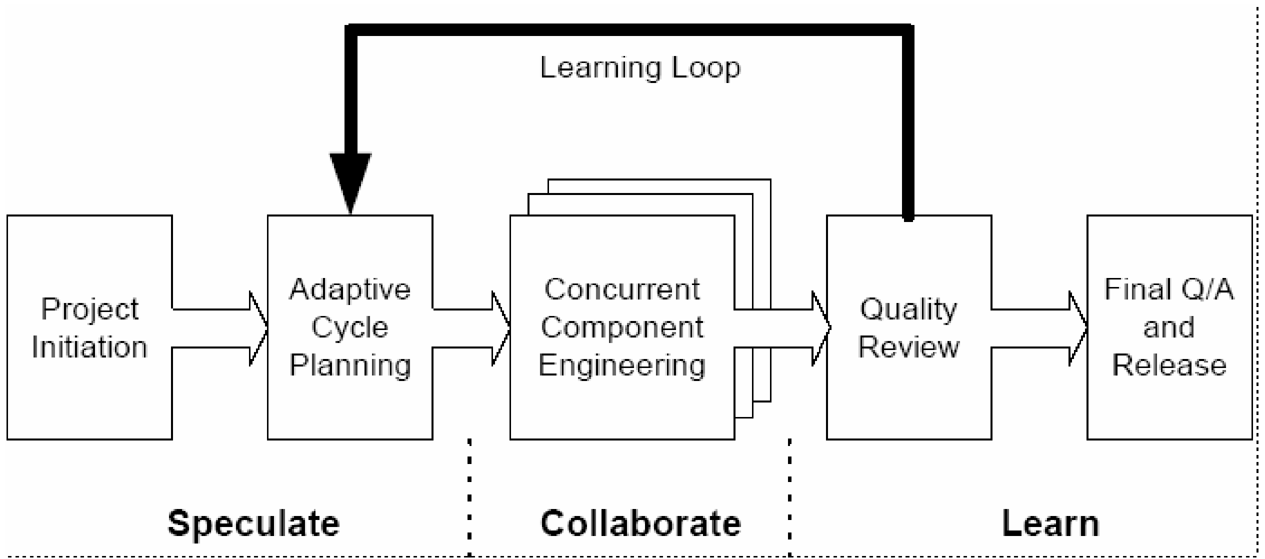

Figure 3.2 The ASD Lifecycle Phases
Source: "(WEB_6, 2005)"

The learning loop is gained from repeated quality reviews that focus on demonstrating the functionality of the software developed. The presence of customer as a group of experts is essential in performing the reviews.

The final stage in adaptive development is the final quality assurance and release stage. Adaptive development does not dictate how to carry out this pahse but stresses on the importance of capturing the lessons learned "(WEB_6, 2005)".

3.4 Extreme Programming

Extreme Programming(XP) was invented by Kent Beck and Ron Jeffries "(Beck, 1999)" and has gained the widest interest among the community among all agile methodologies. XP focuses on coding as the main task of software development process. Beck defines communication, simplicity, feedback and courage as the four key values of XP "(Beck, 1999)". *Communication* is facilitated through activities such as pair programming, task estimation and iteration planning. *Simplicity* stands for chooding the simplest design, technology, algorithms and techniques that will satisfy the customer's need for the current iteration of the project. *Feedback* is obtaned through code testing, working with customers, small iterations and frequent deliveries, pair programming and constant code reviews. *Courage* stands for the confidence to refactor

and rearchitect, or even throwing away code without fear. This courage is achieved through automated regression tests.

3.4.1 Twelve Practices of Extreme Programming

Beck, defines twelve essential practices that characterizes a process as XP, which are explained as follows "(Beck, 1999)";

Planning Game

The purpose of the planning game is to determine the scope of the current iteration. Planning game encompasses determining the scope of the project with the customer, priority of features, composition of releases and delivery dates. The developers assist the customer by estimating task duration, considering consequences and risks, organizing the team and performing technical risk management. The developers and the customers act as a team.

Small Releases

The philosophy behind small releases practice is to provide the most business value with the least amount of coding effort. The features have to be atomic. A feature must implement enough functionality for it to have business value. The idea is to get the Project into production as soon as possible, so that it can provide value to customers and rapid feedback about customer requirements can be gained.

Simple Design

The idea behind simple design is keeping the code simple. The simples design possible does not try to solve future problems by guessing future needs. The simplest design passes the customer's acceptence test for that release of the software, has no duplicate logic code, and is not convoluted but expresses every developer's purpose.

Testing

Code can be rearchitected, refactored, or thrown out and completely redone. Later, the tests can show that the system stil works. Testing keeps the code fluid. Because tests typically check the public interface of classes and components, the implementation of the system can change drastically while the automated tests validate that the system stil fulfills the contract of the interfaces. A feature does not exist unless a test validates that it functions. Everything that can potentially break must have a test.

Continious Integration

Every few hours(at least once every day) the system should be fully built and tested, including all the latest changes. In order to facilitate this practice, build, distribution and deployment procedures should be automated.

Refactoring

Refactoring means improving the code and design without changing the functionality. Continious refactoring enables developers to add features while keeping the code simple and stil being able to run all the tests. The idea is not to duplicate code nor write ugly code. The act of refactoring centers on testing to validate that the code stil functions. Testing and refactoring go hand in hand.

Pair Programming

Pair programming means having two programmers working on the same task. The pairs change frequently(generally every day), so that collective code ownership, communication and collaboration increases within a team. Knowledge and experience within the team increases by employing pair programming. While one developer is focusing on the coding, the other developer can be thinking about the bigger picture, how the code will fit in with the other system.

Collective Ownership

The XP practice of collective ownership states that anyone can make a change to the system. No one owns a class. Everyone contributes, and if one of the developers need to make a change or refactor something to get it to work with a new feature, she can.

40-Hour-Week

The 40 hour week practice states that if developers cannot do their work in a 40 hour week, then something is wrong with the project. Burned out developers make a lot of mistakes and over-whelming schedules should not be norm.

On-Site Customer

The on site customer practice states that if at all possible, the customer should be available when the developers need to ask questions. And, if at all possible, the customer should be physically located with the developers.

Metaphor

A *metaphor* is a common language and set of terms used to envision the functionality of a project. These terms can equate to objects in the physical world, such as accounts or objects. The idea is for the customer and developers to have the same vision of the system and be able to speak about the system in a common dialect.

Coding Standart

Everyone in a team should follow the practice of using the same coding standart so that developers can understand one another's code.

3.5 Dynamic Systems Development Method

The DSDM was developed in the United Kingdom in the mid-1990s. It is an out-growth of, and extension to, rapid application development practices. The fundamental idea behind this method is that instead of fixing the amount of functionality in a product, and then adjusting time and resources to reach that functionality, it is preffered to fix time and resources, and then adjust the amount of functionality accordingly.

DSDM consists of five phases: feasibility study, business study, functional model iteration, design and build iteration and implementation. The first two phases are sequential, and done only once. The last three phases, during which the actual development work is done are iterative and incremental. DSDM approaches iterations as timeboxes. A timebox lasts for a predefined period of time, and the iteration has to end within the timebox. A typical timebox duration is from a few days to a few weeks. The five phases of DSDM are explained below with their essential output documents "(WEB_6, 2005)";

Business Study: This is the phase where the essential characteristics of the business and technology are analyzed. The proposed approach is organizing workshops where all relevant facets of the system are considered and developement priorities are established with the customer. The affected business processes and user classes are described in a Business Area Definition. Other two outputs generated in this pahse are System Architecture Definition and Outline Prototyping Plan. The architecture definition is the first system architecture sketch, and it is allowed to change during the course of the project. The prototyping plan should state the prototyping strategy fort he following stages and a plan for configuration management.

Functional Model Iteration: This is the first iterative and incremental phase. In every iteration, the contents and the approach for the iteration are defined, the iteration gone through and the results analyzed for further iterations. Both analysis and coding are done, prototypes are built, and the experiences gained are used in improving the analysis models. A functional model is produced as an output, containing the prototype code and the analysis models. Testing is also a continuing, essential part of this phase.

Design And Build Iteration: This is where the system is mainly built. The output is a tested system that fulfills at least the minimum agreed set of requirements. Design and build are iterative, and the design and functional prototypes are reviewed by the users, and further development is based on the comments of the users.

Implementation: This is the phase where the system is transferred from the development environment into the actual production environment. Training is given to users, and the system is handed over to them. The output of the implementation phase are a user manual and a Project review report. The latter summarizes the outcome of the project, and based on the results, the course of further development is set.

3.6 Summary

Agile software development methods officially started with the publication of the **Agile Manifesto.** The attempt of the manifesto was bringing a paradigm shift in the field of software engineering. Agile methods claim to put more emphasis on people, interaction, working software, customer collaboration and change rather than on processes, tools, contracts and plans. The word **agile** from software development perspective means incremental development with small releases and rapid cycles, close communication and collaboration of customers and developers, applying easy to understand processes instead of detailed and mechanical approaches and being able to make last moment changes. While sharing all the above perspectives, some agile methods are more focused than others. For example, they support different phases of the software product development to a varying degree. Differences also exist in the level of correctness in which the methods are tied to actual development practice. For example ASD is dominantly focused on principles that guide the software development while XP places emphasis on the development practices.

CHAPTER 4

TEST DRIVEN DEVELOPMENT

Test Driven Development (TDD) “(Beck, 2003)”, also known as test first programming or test first development as a new approach to software development where the programmer must first write a test that fails before writing any functional code. This practice became popular with the birth of **Extreme Programming(XP)** “(Beck, 1999)”, and is one of the key practices of XP.

Test driven development is a programming technique which is applied by following a series of steps repetetively. The first step is to quickly add a test, basically just enough code to fail. Next, the tests are run, often the complete test suite although for sake of speed only a subset of the test suite may be chosen, to ensure that the new test does in fact fail. Then the functional code is updated to make it pass the new tests. The next step is running the tests again. If they fail, the functional code needs to be updated to pass the tests. Once the tests pass the next step is to start over.

4.1 Fundamental Concepts in Test Driven Development

Test Driven Development relies on two main concepts, unit tests and refactoring. Unit tests allow the programmer to test the software in a more detailed way than she would with functional tests. With unit tests, small units of code at the application programming interface(API) level can be tested. Unit tests are also easier to run than functional tests since they do not require a full production environment to run. Unit tests help developers become more productive because it makes it easier to track down bugs and run regression tests to monitor the state of the software.

To create effective unit tests, the behavior of the unit of software that is being created must be understood first. This can usually be done by decomposing the software requirements into simple testable behaviors. Software requirements may be defined in a number of different ways, such as a formal requirement spec, use cases, or in the form of some simple user stories. For TDD, it doesn't matter how the requirements are specified, as long as they can be translated into tests.

Refactoring “(Fowler, 1999)” is the process of changing code for the sole benefit of improving the internal structure of the code without changing its function. Refactoring is basically cleaning up bad code. Refactoring is an important part of TDD, a programmer needs to develop an understanding of refactoring methods so that she can quickly recognize the patterns of bad code and refactor them to improve the code. The other important thing to be learned to use TDD is how to write effective tests that can be used to drive the development of the software. Most developers understand how to write tests to test code that has already been written, but writing tests before writing the code takes a different approach. To write tests that drive the development of code, the programmer must concentrate on how to test the functionality of the code rather than its implementation.

4.2 Test Driven Development Methodology

On the surface, TDD is a very simple methodology that relies on two main concepts: unit tests and refactoring. TDD is basically composed of the following steps:

- Writing a test that defines how a small part of the software should behave.
- Making the test run as easily and quickly as possible. Design of the code is not a concern, the sole aim is just getting it to work.
- Cleaning up the code. A step back is taken and any duplication or any other problems that were introduced to get the test to run is refactored and removed.

TDD is an iterative process, and these steps are repeated a number of times until satisfaction with the new code is achieved. TDD doesn't rely on a lot of up-front design to determine how the software is structured. The way TDD works is that requirements, or use cases, are decomposed into a set of behaviors that are needed to fulfill the requirement. For each behavior of the system, the first thing done is to write a *unit test* that will test this behavior. The unit test is written first so that a well-defined set of criteria is formed that can be used to tell when just enough code to implement the behavior has been written. One of the benefits of writing the test first is that it actually helps better define the behavior of the system and answer some design questions.

4.3 Benefits of Test Driven Development

Test Driven Development contributes to software development practice from many aspects such as requirements definition, writing clean and well designed code, change and configuration management. The promises of TDD can be summarized as follows;

Simple, Incremental Development: TDD takes a simple, incremental approach to the development of software. One of the main benefits to this approach is having a working software system almost immediately. The first iteration of this software system is very simple and doesn't have much functionality, but the functionality will improve as the development continues. This is a less risky approach than trying to build the entire system all at once, hoping it will work when all the pieces are put together.

Simpler Development Process: Developers who use TDD are more focused. The only thing that a TDD developer have to worry about is getting the next test to pass. The goal is focusing the attention on a small piece of the software, getting it to work, and moving on rather than trying to create the software by doing a lot of up-front design. Thousands of decisions have to be made to create a piece of software. To make all those decisions correctly before starting writing the code is a complex challenge to undergo many times. It is much easier to make those decisions as developing the code.

Constant Regression Testing: The domino effect is well known in software development. Sometimes a simple change to one module may have unforeseen consequences throughout the rest of the project. This is why regression testing is important. Regression testing is like self-defense against bugs. It's usually done only when a new release is sent to quality assurance (QA). By then it's sometimes hard to

trace which code change introduced a particular bug and makes it harder to fix. TDD runs the full set of unit tests every time a change is made to the code, in effect running a full regression test every time a minor change is made. This means any change to the code that has an undesired side effect will be detected almost immediately and be corrected, which should prevent any regression surprises when the software is handed over to QA. The other benefit of constant regression testing is having a fully working system at every iteration of development. This allows the development team to stop development at any time and quickly respond to any changes in requirements.

Improved Communication: Communicating the ideas needed to explain how a piece of software should work is not always easy with words or pictures. Words are often imprecise when it comes to explaining the complexities of the function of a software component. The unit tests can serve as a common language that can be used to communicate the exact behavior of a software component without ambiguities.

Improved Understanding of Required Software Behavior: The level of requirements on a project varies greatly. Sometimes requirements are very detailed and other times they are vague. Writing unit tests before writing the code helps developers focus on understanding the required behavior of the software. As writing a unit test, pass/fail criteria for the behavior of the software is being added. Each of these pass/fail criteria adds to the knowledge of how the software must behave. As more unit tests are added because of new features or new bugs, the set of unit tests come to represent a set of required behaviors of higher and higher fidelity.

Centralization of Knowledge: Humans all have a collective consciousness that stores ideas they all have in common. Unfortunately, programming is mostly a solitary pursuit. Modules are usually developed by a single individual, and a lot of the knowledge that went into designing the module is usually stuck in the head of the person who wrote the code. Even if it's well documented, clean code, it's sometimes hard to understand some of the design decisions that went into building the code. With TDD, the unit tests constitute a repository that provides some information about the design decisions that went into the design of the module. Together with the source code, this provides two different points of view for the module. The unit tests provide a list of requirements for the module. The source code provides the implementation of the requirements. Using these two sources of information makes it a lot easier for other developers to understand the module and make changes that won't introduce bugs.

Better Encapsulation and Modularity: Encapsulation and modularity help managing the chaos of software development. Developers cannot think about all the factors of a software project at one time. A good design will break up software into small, logical, manageable pieces with well defined interfaces. This encapsulation allows developers concentrate on one thing at a time as the application is built. The problem is that sometimes during the fog of development one may stray from the ideas of encapsulation and introduce some unintended coupling between classes. Unit tests can help detect unencapsulated a module. One of the principles of TDD says that the unit tests should be easy to run. This means that the requirements needed to run any of the unit tests should be minimized. Focusing on making testing easier will force a developer making more modular classes that have fewer dependencies.

Simpler Class Relationships: A well designed piece of software will have well defined levels that build upon each other and clearly defined interfaces between the levels. One of the results of having software that has well defined levels is that it's easier to test. The corollary to this is also true. If code is designed by writing tests, the focus will be very narrow, so the tests will tend not to create complex class relationships. The resulting code will be in the form of small building blocks that fit neatly together. If a unit test is hard to write, then this usually means there is a problem in the design of the code. Code that is hard to test is usually bad code. Since the creation of the unit tests help point out the bad code, this allows to correct the problem and produce better designed, more modular code.

Reduced Design Complexity: Developers try to be forward looking and build flexibility into software so that it can adapt to the ever-changing requirements and requests for new features. Developers are always adding methods into classes just in case they may be needed. This flexibility comes at the price of complexity. It's not that developers want to make the software more complex, it's just that they feel that it's easier to add the extra code up front than make changes later. Having a suite of unit tests allows to quickly tell if a change in code has unforeseen consequences. This will give the developer the confidence to make more radical changes to the software. In the TDD process, developers will constantly be refactoring code. Having the confidence to make major code changes any time during the development cycle will prevent developers from overbuilding the software and allow them to keep the design simple. The approach to developing software using TDD also helps reduce software complexity. With TDD the goal is only adding the code to satisfy the unit tests. This is usually called *developing by intention*. Using TDD, it's hard to add extra code that isn't needed. Since the unit tests are derived from the requirements of the system, the end result is just enough code to have the software work as required.

4.4 TDD and Software Requirements

It is a well known fact that one of the main reasons of a software project failure is misunderstood or badly managed requirements. Requirements documented in the form of design diagrams or free text always have the risk of being incomplete or unclear as opposed to program code, which is formal and by its nature, is unambigious. Consequently, simply designed and well decomposed tests reveal the behavior of a piece of code in an unambigious and clear way.

Agile software development methodologies assume that a full set of requirements for a system cannot be determined upfront. Instead, requirements are gathered and modified throughout development leading to a flexible development process. Requirements gathering process is accomplished through small releases and constant feedback from the customer. Requirements can be defined as features that the customer wishes for the system to have. A feature can also be expressed as an *action* that has a *result* on an *object.*

In Extreme Programming(XP) “(Beck, 1999)” requirements are collected as short user stories written on small cards. That means high level business functions are expressed as a collection of finer grained features. This is complementary with the rhythm of test first design, deciding on a feature, writing a small test that validates the feature is working and implementing it. XP states that a feature that has not been tested does not exist.

4.5 TDD and Software Design

One of the main focuses of agile software development practices is keeping the software as simple as possible without sacrificing quality. That is, any duplication of logic in software must be eliminated and the code should be kept clean. Agile methods proposes a key approach to accomplish these goals; implementing the simplest solution that works for a feature, just enough to pass the tests. Customer requirements constantly change. Trying to estimate possible future requirements and designing the software with this in mind may introduce many unneeded features into the software. This might make the software more complicated and bigger than needed, which in the end would result in additional maintenance burden. Having a full coverage of programmer tests makes it possible to refactor the code or introducing new requirements.

Test driven development also helps designing modular and well decomposed software. In order for a piece of code to be easily tested in isolation, its dependencies to other modules and functions must be minimized. This drives the developer to write more decoupled classes and
modules.

4.6 TDD and Software Maintenance

The design of a software system tends to decay during its lifetime, as new or changing requirements or bug fixes patched into the code unless it is continiously refactored and rearchitected. Without a suite of complete regression tests, refactoring a software is practically impossible because it will not be possible to know which parts of the software are affected from a change. Test driven development includes continious but small refactorings into the development activity itself. The code, test, refactor cycle is applied at every small step so that the design is always kept clean and the code is always kept working.

4.7 TDD and Software Documentation

Documenting software code is a daunting and mechanical task. Just like software design tends to decay in time, documentation tends to get outdated, including the comments in the code. Besides, it is not possible to validate the documentation by an automated process. In contrast, automated tests are always kept up to date because they are run all the time. Moreover, whatever form it is in, design diagrams or free text, documentation always has the risk of being ambigious. Agile methodologies does not claim that all documentation should be replaced with automated tests, however they tend to keep the documentation as small as possible.

4.8 Summary

Although its close relatioship to extreme programming, test driven development is not bounded by any development methodology. Test driven development is just a new paradigm of programming technique. It aims to keep the code clean and well designed by continious refactoring, always working by continious testing and the process itself lightweight by making coding itself as the main activity. TDD tries to keep the focus on what to implement by writing the test code before functional code. All these key points make test driven development a disciplined method of programming which gains more attention everyday from the software developers community.

CHAPTER 5

TEST DRIVEN DEVELOPMENT PRACTICES

Test driven development has brought into the world the concept of **programmer testing.** In recent years some programmers have discovered the benefits of writing their own tests, instead of relying on the thought of "*the testing department would care for it"*. Fixing defects is a time consuming process. It takes time for testers to discover the defect, to describe it in enough detail for the programmers to be able to recreate it. It takes time for the programmers to determine the causes of the defects, inspecting through the codes they have not seen possibly for months. Much of this wasted time could be avoided if the programmers simply tested their own code. The real value of programmer testing shines especially during change management and software maintanence. Regression tests applied by the programmer would reveal the breaking parts of a software after a change has been made.

The testing that programmers do is usually called ***unit testing.*** However, this term is somewhat vague and overused, since there's no formal definition of what a **unit** is. Is it a method, a class, or a code path? So, the term **object testing** is also used among the community to refer to tests on individual objects.

5.1 A Framework for Unit Testing

In a paper called *Simple Smalltalk Testing: With Patterns*, Kent Beck "(WEB_7, 2005)" described how to write object tests using smalltalk. This paper presented the evolution of a simple testing framework that became known as *SUnit*. "(WEB_13, 2005)" Kent teamed up with Erich Gamma to port the framework to Java and called the result *JUnit*. "(WEB_14, 2005)" Since 1999, JUnit has evolved into an industry standard testing and design tool for Java, gaining wide acceptance not only on open source projects, but also in commercial software companies. Kent Beck's testing framework has been ported to over 30 different programming languages and environments.

JUnit was created as a framework for writing automated, self-verifying tests in Java, which in JUnit are called *test cases*. JUnit provides a natural grouping mechanism for related tests, which it calls a *test suite*. JUnit also provides *test runners* that can be used to execute a test suite. The test runner reports on the tests that fail, and if none fail, it simply says "OK." When writing JUnit tests, all the programming knowledge is put into the tests themselves so that they become entirely programmer independent. This means that anyone can run tests without knowing what they do, and if they are curious, they only need to read the test code.

5.1.1 Setting Up a Test Environment with JUnit

JUnit package can be acquired from www.junit.org. It is also bundled with almost all of the popular Java Integrated Development Environments including Eclipse "(WEB_15, 2005)", NetBeans "(WEB_16, 2005)" and JDeveloper "(WEB_17, 2005)", as a set of plug-ins so that no additional setup work is needed by the programmer to write JUnit tests.

5.1.2 A Simple JUnit Test Case

The example below is a small but complete JUnit test case class.

```
public class MoneyTest extends TestCase { //Create a subclass of TestCase

        public void testAdd() { //Each test is a method
                Money addend = new Money(30, 0);  //30 dollars, 0 cents
                Money augend = new Money(20, 0);
                Money sum = addend.add(augend);
                assertEquals(5000, sum.inCents()); //The parameters should be equal
        }

}
```

This example demonstrates several aspects of JUnit, including:

- To create a test, a method is written that expresses the test. This method is named **testAdd(),** using a JUnit naming convention that allows JUnit to find and execute the tests automatically. This convention states that the name of a method implementing a test must start with "test".
- The test needs a home. The method is placed in a class that extends the JUnit framework class *TestCase.*
- To Express how the object under test is expected to behave, assertions are made. An assertion is simply a statement of the expectation. JUnit provides a number of methods for making assertions. Here *assertEquals()* is used, which tells JUnit, "if these two values are not the same, the test should fail."

5.1.3 The *TestCase* Class

The *TestCase* class which is located in the package *junit.framework* is the center of the JUnit framework. There is a confusing point about the term *test case* and its relation to the *TestCase* class, which is a name collision. The term *test case* generally refers to a single test, verifying a specific path through the code. However, multiple *test cases* are collected into a single class which itself is a subclass of ***junit.framework.TestCase*** with each *test case* implemented as a method on ***TestCase.***
The class ***TestCase*** extends a utility class named ***Assert*** in the Junit framework. The ***Assert*** class provides methods to make assertions about the state of the objects under test. The basic assertion methods in JUnit are described in **Table 5.1.** JUnit also provides additional assertion methods for the logical opposites of the ones listed in the

table: ***assertFalse(),*** ***assertNotSame(),*** **and** ***assertNotNull()***. Two of the overloaded versions of ***assertEquals()*** are slightly different. The versions that compare double and float values require a third parameter: a *tolerance level*. This tolerance level specifies how close floating-point values need to be before being considered equal. Because floating-point arithmetic is imprecise at best, it can be specified "these two values can be within 0.0001, which is close enough" by coding ***assertEquals(expectedDouble, actualDouble, 0.0001d).***

Method	What it does
`assertTrue(boolean condition)`	Fails if `condition` is false; passes otherwise.
`assertEquals(Object expected, Object actual)`	Fails if `expected` and `actual` are not equal, according to the `equals()` method; passes otherwise.
`assertEquals(int expected, int actual)`	Fails if `expected` and `actual` are not equal according to the == operator; passes otherwise. There is an overloaded version of this method for each primitive type: `int`, `float`, `double`, `char`, `byte`, `long`, `short`, and `boolean`. (See Note about `assertEquals()`.)
`assertSame(Object expected, Object actual)`	Fails if `expected` and `actual` refer to different objects in memory; passes if they refer to the same object in memory. Objects that are not the same might still be equal according to the `equals()` method.
`assertNull(Object object)`	Passes if object is `null`; fails otherwise.

Figure 5.2 The JUnit class Assert provides several methods for making assertions.

5.1.4 Failure Messages

When an assertion fails, it can be of use including a short message that indicates the nature of the failure, even a reason for it. Each of the assertion methods accepts as an optional first parameter a String containing a message to display when the assertion fails. Besides a user defined one, ***assertEquals()*** has its own customized failure message, so that if an equality assertion fails this message is seen:

junit.framework.AssertionFailedError: expected:<4999> but was:<5000>

5.1.5 Signaling Failed Assertions

The key to understanding how JUnit decides when a test passes or fails lies in knowing how these assertion methods signal that an assertion has failed. In order for JUnit tests to be self-verifying, assertions about the state of objects under test must be made and *JUnit* must raise a red flag when production code does not behave according to the assertions. In Java, as in C++ and Smalltalk, the way to raise a red flag is with an exception. When a JUnit assertion fails, the assertion method throws an exception to indicate that the assertion has failed. To be more precise, when an assertion fails, the

assertion method throws an error: an ***AssertionFailedError***. The following is the source for assertTrue():

```
static public void assertTrue(boolean condition) {
        if (!condition)
                throw new AssertionFailedError();
}
```

When it is asserted that a condition is true but it is *not*, then the method throws an ***AssertionFailedError*** to indicate the failed assertion. The JUnit framework then catches that error, marks the test as failed, remembers that it failed, and moves on to the next test. At the end of the test run, JUnit lists all the tests that failed; the rest are considered as having passed.

5.1.6 Differences Between Failures and Errors

Normally, Java code that the programmer writes does not throw errors, but rather only exceptions. General practice leaves the throwing of errors to the Java Virtual Machine itself, because an error indicates a low-level, unrecoverable problem, such as not being able to load a class. This is the kind of failure which the programmer is not expected to recover. For that reason, it might seem strange for JUnit to throw an error to indicate an assertion failure. JUnit throws an error rather than an exception because in Java, errors are unchecked; therefore, not every test method needs to declare that it might throw an error. It might be argued that a ***RuntimeException*** would have accomplished the same, but if JUnit threw the kinds of exceptions the production code might throw, then JUnit tests might interfere with production code. Such interference would diminish JUnit's value.

When a test contains a failed assertion, JUnit counts that as a failed test; but when a test throws an exception (and does not catch it), JUnit counts that as an error. The difference is subtle, but useful: a failed assertion usually indicates that the production code is wrong, but an error indicates that there is a problem either with the test itself or in the surrounding environment.

5.2 Fundamental Test Driven Development Practices

This section discusses some fundamental best practices about testing simple objects, which should be considered at every test case design. Well designed, testable production code makes automated unit testing possible. Well designed test cases makes it possible to write flexible, reusable, easy to maintain tests. Well maintained tests leads to high code coverage and can better describe how the system is expected to behave.

5.2.1 Naming Test Case Classes

When naming test case classes, the simplest guideline is to name the test case class after the class under test. In other words, tests for an *Account* `object` go into the class *AccountTest*, tests for a *FileSystem* `object` go into the class *FileSystemTest*. The principal benefit of this naming convention is that it is easy to find a test, as long as the

class being tested is known. While starting out with this naming convention is recommended, it is important to understand that this is *only* a naming convention, and not a requirement of JUnit. Test case classes that grow unusually large or become difficult to navigate should be split into smaller classes. Identifying the tests that share a common fixture and factoring them into a separate test case class would be the way to go in such a situation. If there are six special cases for withdrawing money from an account, then moving them into a test case class called *AccountWithdrawalTest* or *WithdrawFromAccountTest* would lead to more clean and maintainable seperation of test cases.

5.2.2 Testing Behavior Instead of Methods

Tests should focus on the behavior they are testing without worrying about which class is under test. This is why test names tend to be verbs rather than nouns: it is behavior (verbs) being tested and not classes (nouns). Still, the difference between behavior and methods might not be clear: behavior is implemented as methods, so testing behavior must be about test methods. But that is not entirely true. Behavior is implemented as methods, but the way chosen to implement a certain behavior depends on a variety of factors, some of which boil down to personal preference. A number of decisions are made by the programmer when implementing behavior as methods: their names, their parameter lists, which methods are public and which are private, the classes on which the methods are placed, and so on. These are *some* of the ways in which methods might differ, even though the underlying behavior might be the same. The implementation can vary in ways that the tests do not need to determine. Sometimes a single method implements all the required behavior, and in that case, testing that method directly is all needed. More complex behaviors require the collaboration of several methods or even objects to implement. If tests depend too much on the particulars of the implementation, then the programmer creates work for herself as she tries to refactor(improve the design). Furthermore, some methods merely participate in a particular feature, rather than implement it. Testing those methods in isolation might be more trouble than it is worth. Doing so drives up the complexity of the test suite (by using more tests) and makes refactoring more difficult and all for perhaps not much gain over testing the behavior at a slightly higher level. By focusing on testing behavior rather than each individual method, a beter balance between test coverage and the freedom needed to support refactoring can be achieved.

To illustrate the point, testing a stack can be considered. A stack provides a few basic operations: push (add to the top of the stack), pop (remove from the top of the stack), and peek (look at the top of the stack). When deciding how to test a stack implementation, the following tests comes to mind first:

- Popping an empty stack fails somehow.
- Peeking at an empty stack shows nothing.
- Push an object onto the stack, then peek at it, expecting the same object that has been pushed.
- Push an object onto the stack, then pop it, expecting the same object that has been pushed.
- Push two different objects onto the stack, then pop twice, expecting the objects in reverse order from the order in which they had been pushed.

These tests focus on *fixture,* the state of the stack when the operation is performed, rather than on the operation itself. Besides the methods combine to provide the desired behavior. If *push()* does not work, then there is no good way to verify *pop()* and *peek()*. Moreover, when using a stack, all three operations are used, so it does not make sense to test them in isolation, but rather to test that the stack generally behaves correctly, depending on its contents. This points the fact that an object is a cohesive collection of related operations on the same set of data. The object's *overall* behavior, a composite of the behaviors of its methods in a given state, is what is important. So focusing test effort on the object as a whole, rather than its parts is the essential idea.

5.2.3 Best Practices for Elementary Tests

5.2.3.1 Testing the Return Value of a Method

Here is how to test a method that returns a value: invoke the method and compare its return value against the value expected:

```
public void testNewListIsEmpty() {
        List list = new ArrayList();
        assertEquals(0, list.size());
}
```

The method *assertEquals()* is intended to compare either primitive values or objects. A primitive value is a value of type *int, float, boolean, double, long, char,* or *byte*. The framework compares primitives as values, so that *assertEquals(3, value)* passes as long as value is a numeric primitive variable with the value 3. If the method under test returns an *Object*, rather than a primitive, then more work needs to be done to compare its return value against an expected value.

There are two techniques for comparing the expected value to the actual return value from the method invoked. The first technique is to compare each readable property of the method's return value with the values expected. The second is to create an object representing the value expected and then compare it to the actual value with a single line of code.

```
public void testSynchronizedListHasSameContents() {
        List list = new ArrayList();
        list.add("Albert");
        list.add("Henry");
        list.add("Catherine");
        List synchronizedList = Collections.synchronizedList(list);
        assertEquals("Albert", synchronizedList.get(0));
        assertEquals("Henry", synchronizedList.get(1));
        assertEquals("Catherine", synchronizedList.get(2));
}
```

The code snippet above tests *Collections.synchronizedList()*. This method is supposed to add thread safety features to a *List* without affecting the contents of the list. This is an example of the first technique, where each element of the *ArrayList* is compared against the values created in the *SynchronizedList*. The most notable shortcoming of this technique is that it involves a great deal of typing.

The test needs to verify that the two list's corresponding elements should be equal. The contract of *List.equals()* method in Java says that lists are equal if they have the same elements at the same indices. So, the above test could be rewritten as;

```
public void testSynchronizedListHasSameContents() {
        List list = Arrays.asList(
                new String[] { "Albert", "Henry", "Catherine" });
        assertEquals(list, Collections.synchronizedList(list));
}
```

Assigning the responsibility of determining equality into the objects that are best equipped to do it, the lists themselves simplifies the test code and makes it more maintainable.

5.2.3.2 Testing Object Equality

In Java language value objects are compared by the *equals()* method of class *Object*. Storing value objects in collections*(List, Set, Map)* requires implementing *equals()* and *hashcode()* methods appropriately. Mathematically the method *equals()* must satisfy three properties; the reflexive, symmetric, and transitive properties.

- The reflexive property says that an object is equal to itself.
- The symmetric property says that if A is equal to B, then B is equal to A.
- The transitive property says that if A is equal to B and B is equal to C, then A is equal to C.

Beyond these mathematical properties, the `equals()` method must be consistent; no matter how many times you call it on an object, *equals()* answers the same way as long as neither object being compared changes. Finally, no object equals ***null.***

```
public class MoneyEqualsTest extends TestCase {
        private Money a;
        private Money b;
        private Money c;

        protected void setUp() {
                a = new Money(100, 0);
                b = new Money(100, 0);
                c = new Money(200, 0);
        }

        public void testReflexive() {
                assertEquals(a, a);
                assertEquals(b, b);
                assertEquals(c, c);
        }

        public void testSymmetric() {
                assertEquals(a, b);
                assertEquals(b, a);
                assertFalse(a.equals(c));
                assertFalse(c.equals(a));
        }
```

```
        public void testConsistent() {
                for (int i = 0; i < 1000; i++) {
                        assertEquals(a, b);
                        assertFalse(a.equals(c));
                }
        }

        public void testNotEqualToNull() {
                assertFalse(a.equals(null));
                assertFalse(c.equals(null));
        }

}
```

The TestCase class above tests all needed functionality for the equivalence of Money objects. However, trying to write that much test code for a simple equality test is a tremendous work. An open source package, GSBase "(WEB_18, 2005)" provides a number of utilities for writing JUnit tests, including a class called the *EqualsTester*. This class runs a complete suit of tests on the value objects to determine whether the implementation of *equals()* satisfies all necessary properties. The *TestCase* class listed below tests the *Money* class using *EqualsBuilder*.

```
public class MoneyTest extends TestCase {
        public void testEquals() {
                Money a = new Money(100, 0);
                Money b = new Money(100, 0);
                Money c = new Money(50, 0);
                Object d = null;
                new EqualsTester(a, b, c, d);
        }
}
```

The EqualsTester takes four parameters as listed below;

- `a` is a control object against which the other three objects are to be compared.
- `b` is a different object in memory than *a* but equal to *a* according to its value.
- `c` is expected not to be equal to *a*. If the class under test is ***final***, cannot be subclassed, then *d* ought to be ***null***; otherwise, *d* represents an object that looks equal to *a* but is not. That means, for example, that *d* has the same properties as *a*, but because *d* has additional properties through subclassing, it is not equal to *a*.

5.2.3.3 Testing an Interface

When publishing an interface, the aim is to enforce a common behavior among all possible implementations of that interface. This is not only the case for the existing implementations but also for the possible future ones. Consequently, both the existing and future implementations of the interface must be tested against the desired behavior of the interface without duplicating the underlying test code.

The solution to this problem is introducing an *abstract test case* whose methods test the intended common behavior and that uses factory methods to defer creating the object(implementation) under test to the test case of the implementation.

Below is an illustration of this technique by testing the interface *java.util.Iterator;*

```
public abstract class IteratorTest extends TestCase {

        private Iterator noMoreElementsIterator;
        protected abstract Iterator makeNoMoreElementsIterator();

        protected void setUp() {
                noMoreElementsIterator = makeNoMoreElementsIterator();
        }

        public void testHasNextNoMoreElements() {
                assertFalse(noMoreElementsIterator.hasNext());
        }



        public void testNextNoMoreElements() {
                try {
                        noMoreElementsIterator.next();
                        fail("No exception with no elements remaining!");
                } catch (NoSuchElementException expected) {}
        }

        public void testRemoveNoMoreElements() {
                try {
                        noMoreElementsIterator.remove();
                        fail("No exception with no elements remaining!");
                } catch (IllegalStateException expected) {}
        }
}
```

The class *ListIteratorTest* below extends the *IteratorTest* and inherits all the test cases from its superclass.

```
public class ListIteratorTest extends IteratorTest {

        protected Iterator makeNoMoreElementsIterator() {
                List empty = new ArrayList();
                return empty.iterator();
        }
}
```

Rather than extend *junit.framework.TestCase* directly, *ListIteratorTest* extends the abstract test case *IteratorTest*. The *ListIteratorTest* implementation of this creation method returns an iterator over an empty *List*. Similarly, if a Set-based iterator were being tested, a *SetIteratorTest* would have to be created (extending *IteratorTest*) whose *makeNoMoreElementsIterator()* would return an iterator over an empty *Set*.

5.2.3.4 Testing Throwing the Right Exception

Sometimes it must be verified that a method under test throws an expected exception under the appropriate circumstances. In Junit, a test fails (with either a failure or an error) if an assertion fails or it throws an exception, otherwise the test passes. In other words, if a test executes from beginning to end without exiting from the middle then it passes. The following code demonstrates the implementation pattern for writing tests for throwing an exception;

```
public void testConstructorDiesWithNull() throws Exception {
        try {
                Fraction oneOverZero = new Fraction(1, 0);
                fail("Created fraction 1/0! That's undefined!");
        } catch (IllegalArgumentException expected) {
                assertEquals("denominator", expected.getMessage());
        }
}
```

If the method under test throws an unexpected exception-something other than the exception for which the method is being tested- then Junit reports an error, rather than a failure, because the test method throws that exception up into the Junit framework. While the method above is easy to understand, it can be rewritten as below with a more object-oriented approach;

```
public void testForException() {
        assertThrows(MyException.class, new ExceptionalClosure() {
                public Object execute(Object input) throws Exception {
                        return doSomethingThatShouldThrowMyException();
                }
        });
}

public static void assertThrows(Class expectedExceptionClass,
        ExceptionalClosure closure) {
        String expectedExceptionClassName = expectedExceptionClass.getName();
        try {
                closure.execute(null);
                fail("Block did not throw an exception of type "
                        + expectedExceptionClassName);
        } catch (Exception e) {
                assertTrue("Caught exception of type <"+ e.getClass().getName()
                + ">, expected one of type <"+ expectedExceptionClassName
                + ">", expectedExceptionClass.isInstance(e));
        }
}
```

5.2.4 Testing Design Patterns

Design patterns are recurring solutions to software design problems that is found commonly in real-world application development. Design patterns are about design and interaction of objects, as well as providing a communication platform concerning elegant, reusable solutions to commonly encountered programming challenges. The Gang of Four (GOF) "(Gamma, Helm, Johnson, Vlissides, 1995)" patterns are generally considered the foundation for all other patterns. Some design patterns may present challenges for testing. For example the *Singleton* "(Gamma, Helm, Johnson, Vlissides, 1995)" presents a test isolation problem whereas it may not be obvious how to test an *Observer* "(Gamma, Helm, Johnson, Vlissides, 1995)" without its *Observable* "(Gamma, Helm, Johnson, Vlissides, 1995)". This section discusses some ideas on testing some of the widely used design patterns.

5.2.4.1 Testing an Observer(Event Listener)

The power of the Observer (Event Listener) is that it is not coupled to the Observable (Event Source). The nature of their collaboration makes it possible to test the Observer in perfect isolation. When considering how to test an Observer, an idea might be to create a fake Observable, an object that plays the role of the real Observable, but does something simpler. However, the simplest way to test an Observer is simply to invoke its event handler methods directly and make assertions about what they do. The Observer does not care about the origin of the events it is listening for, by merely invoking the event handler method directly with a variety of events.

JUnit itself has an example of Event Listener. One of its core interfaces is **junit.framework.TestListener.** A test listener registers with the test runner to find out when tests begin, when they end, and when they fail. The text-based test runner registers a *TestListener* to print out a dot (.) for each test, an "F" for each failure and an "E" for each error. Because event handler methods typically do not return values, some side effect is needed to be observed to verify their correctness. In this case, the side effect is whatever the test listener prints out...*somewhere*. In production, this test listener should print to the console, but that would be an invisible side effect for the tests.

```
public class TextBasedTestListenerTest extends TestCase {
        private TextBasedTestListener testListener;
        private StringWriter stringWriter;

        protected void setUp() throws Exception {
                stringWriter = new StringWriter();
                testListener =
                new TextBasedTestListener(new PrintWriter(stringWriter));
        }

        public void testStartTestEvent() throws Exception {
                testListener.startTest(this);
                assertEquals(".", stringWriter.toString());
        }
```

```
		public void testAddFailureEvent() throws Exception {
			testListener.addFailure(this, new AssertionFailedError());
			assertEquals("F", stringWriter.toString());
		}

		public void testAddErrorEvent() throws Exception {
			testListener.addError(this, new RuntimeException());
			assertEquals("E", stringWriter.toString());
		}

		public void testEndTestEvent() throws Exception {
			testListener.endTest(this);
			assertEquals("", stringWriter.toString());
		}

		public void testCompletePassingTestScenario() throws Exception {
			testListener.startTest(this);
			testListener.endTest(this);
			assertEquals(".", stringWriter.toString());
		}

		public void testCompleteTestFailureScenario() throws Exception {
			testListener.startTest(this);
			testListener.addFailure(this, new AssertionFailedError());
			testListener.endTest(this);
			assertEquals(".F", stringWriter.toString());
		}

		public void testCompleteTestErrorScenario() throws Exception {
			testListener.startTest(this);
			testListener.addError(this, new RuntimeException());
			testListener.endTest(this);
			assertEquals(".E", stringWriter.toString());
		}

	}
```

These tests are straightforward: at the start of a test, a dot is seen; when a test fails, an “F” is to be seen and so on.

5.2.4.2 Testing an Observable(Event Source)

Without an event listener there is no way to verify that the event source works correctly. However, the event listener does not have to be the production event listener. The simplest kind of event listener does nothing important when it receives an event, except possibly remember the events it received. To test an observable, the conditions under which an event is expected to be generated should be recreated. Then it should be verified that the observable has notified its listeners. A solution might be that the *test case* class implements the event listener interface, collects the events it receives in a List, and then verifies the received events against a List of expected events. In the former section (5.2.4.1) a text based event listener, an implementation of *TestListener*, which printed information in response to various text execution events was presented.

The following example verifies that JUnit generates those events correctly. Generating those events requires executing a test. The *TestResult* object that is passed into the *TestCase* when executing the test, acts as the observable.

```
public class TestCaseEventsTest extends TestCase implements TestListener {

        private List events;

        public TestCaseEventsTest(String name) {
                super(name);
        }

        protected void setUp() throws Exception {
                events = new ArrayList();
        }

        public void addError(Test test, Throwable throwable) {
                events.add(Arrays.asList(new Object[] { "addError", test, throwable }));
        }

        public void addFailure(Test test, AssertionFailedError failure) {
                events.add(Arrays.asList(new Object[] { "addFailure", test,
                failure.getMessage()}));
        }

        public void endTest(Test test) {
                events.add(Arrays.asList(new Object[] { "endTest", test }));
        }

        public void startTest(Test test) {
                events.add(Arrays.asList(new Object[] { "startTest", test}));
        }

        public void dummyPassingTest() { }

        public void testPassingTestCase() throws Exception {
                final TestCase testCase =
                new TestCaseEventsTest("dummyPassingTest");
                TestResult testResult = new TestResult();
                testResult.addListener(this);
                testCase.run(testResult);
                List expectedEvents = new ArrayList();
                expectedEvents.add(Arrays.asList(new Object[] { "startTest", testCase }));
                expectedEvents.add(Arrays.asList(new Object[] { "endTest", testCase }));
                assertEquals(expectedEvents, events);
        }

}
```

In the example above the test case class implements *TestListener* and collects the various method invocations as events. Because *TestResult* does not generate event objects, information about each event handler method invocation is collected in a List. With this in place, a test is needed which itself will execute another test. The simplest kind of test to execute is a *dummy passing test*. In the example, a dummy passing test is created, a spy *TestListener* is attached to a *TestResult*, and then the test is executed using the *TestResult*. The dummy passing test is named in such a way that the default test suite will not pick it up. This way, only the *TestCaseEventTest. testPassingTestCase()* method can execute the dummy test.

5.2.4.3 Testing a Singleton

The problem with testing a **singleton** "(Gamma, Helm, Johnson, Vlissides, 1995)" is that a singleton object retains state from test to test, making it difficult to isolate the tests that use it. If more than one singleton collaborate with one another, the situation gets worse, since they force the programmer to write longer, more complex text fixtures. Some alternative solutions to achieve test isolation with singletons are listed below;

- Changing the singleton class so that it is no more a singleton. The application entry point instantiates one object, maintain a reference to it, and hand that reference to any components that might need it. However, this solution affects every client of the singleton.
- Making the singleton class constructors *public* so that the tests can instantiate the class but all production clients can continue using the class as a singleton. This solution has the risk of allowing production classes to use the newly visible constructors.
- Adding a *reset()* method to the singleton so that the unique instance could be reset to its initial state between each test run.
- If the singleton has a *protected* constructor, then subclassing the singleton, placing *public* constructors in the subclass and testing the subclass. If test classes are kept in a seperate source tree from production classes, there will be no way for the production code to bypass the singleton and instantiate an instance of the subclass.
- Executing each test in its own JVM or at least a class loader. This is one way to ensure that each test executes a freshly initialized singleton. While seeming elegant, this is a heavyweight solution that brings too much programming burden.

5.2.4.4 Testing an Object Factory

A Factory Method "(Gamma, Helm, Johnson, Vlissides, 1995)" is a kind of object creation method that applies some logic to decide which class to instantiate. Most commonly, the factory method chooses among many subclasses of a class or many implementations of an interface. The code using a factory method often knows nothing about the specific implementation it receives. Testing an object factory brings up many challenges to solve;

- Reading data from either the file system or a database
- Using that data to decide which subclass or implementation to create
- Passing a variable list of parameters to the constructor
- Answering the result

The basic approach is to test the two concrete methods of a factory method seperately;

1. Choosing the *creation method* to invoke. This is the rule that decides which implementation to instantiate.
2. Invoking the *creation method* correctly. The factory method needs to send the correct parameters to the creation method it has decided to use.

The following code snippet tests the first of these two behaviors;

```
public void testChooseMakeSquare() {
        SpyShapeMaker spyShapeMaker = new SpyShapeMaker();
        ShapeFactory factory = new ShapeFactory(spyShapeMaker);
        double[] dummyParameters = new double[0];
        factory.makeShape("square", dummyParameters);
        assertEquals(1, spyShapeMaker.getShapesMadeCount());
        assertEquals(1, spyShapeMaker.getSquaresMadeCount());
}
```

Here, a *SpyShapeMaker* is used. A spy collects information about how other objects invoke its methods. So, a *SpyShapeMaker* is given to the *ShapeFactory*, the spy collects information about the intelligence of the factory and records it. In production environment a **real** *ShapeMaker* is passed to the factory. A simpler design could be for the *ShapeFactory* to own the *ShapeMaker* and manage its life cycle. However, there are some good reasons for making the *ShapeMaker* visible to the outside. First, the tests talk to the *ShapeMaker*, testing it directly. Therefore, more than just the factory uses it. Second, in order to achieve test isolation it is necessary to be able to substitute alternate implementations of an object's collaborators at runtime. So, in the tests, a *SpyShapeMaker* is passed to the factory in order to test it, and in production, a real *ShapeMaker* is passed.

In order to test the second behavior, a production quality class that can make the desired kind of shape is needed. The tests for *ShapeMaker* are very simple;

```
public void testMakeSquare() {
        ShapeMaker shapeMaker = new ShapeMaker();
        Square expected = new Square(5.0d);
        assertEquals(expected, shapeMaker.makeSquare(5.0d));
}
```

Now there's a test for choosing the creation method (the *ShapeFactory* correctly asks the *ShapeMaker* for a square) and a test for the creation method itself (the *ShapeMaker* makes the expected square when asked to make a square). In order to test making new kinds of shapes, adding a "choose the right creation method" test to the *ShapeFactory* is enough. In order to test making different instances of existing kinds of shapes, adding a "creation method" test for the *ShapeMaker* is sufficient.

Finally, the code snippet below, shows a sample implementation of a SpyShapeMaker, able to count how many squares and shapes it has been asked to create;

```
public class SpyShapeMaker extends ShapeMaker {
        private int squaresMade = 0;

        public int getShapesMadeCount() {
                return squaresMade;
        }

        public int getSquaresMadeCount() {
```

```
		return squaresMade;
	}

	public Square makeSquare(double sideLength) {
		squaresMade++;
		return super.makeSquare(sideLength);
	}
}
```

5.2.4.5 Testing a Template Method

Template methods are generally created from classes in an existing hierarchy. *Template Method Pattern* "(Gamma, Helm, Johnson, Vlissides, 1995)" is used when each subclass implements a method to perform the same primitive operations in the same order, even though each subclass might implement some of those primitive operations differently. First, each primitive operation is *extracted* "(Fowler, 1999)" into an appropriately named method. Next, the larger method is *pulled up* "(Fowler, 1999)" into the superclass, along with any operations that all the subclasses implement the same way. Next, abstract methods for the operations that each subclass implements differently are created, pulling them up into the superclass. The result is a superclass with some abstract methods, declaring abstract operations. The main goal is testing that the template method works, no matter how any subclass decides to implement these abstract operations. Testing all the existing subclasses duplicates testing effort and misses the point that the goal is testing **all possible subclasses** no matter how they implement the abstract operations.

There are two aspects of a template method to test, behavior and structure. The structure is tested by verifying that it invokes the expected primitive operations in the expected order. Since a template method consists of invoking a desired set of methods in a desired order, a spy object which records the method invocations in order is needed. The technique can be illustrated using JUnit's own behavior in running a test. JUnit first invokes *setup()*, then the test method and then *teardown().* Below is the source code of the method which runs a test from the class *junit.framework.TestCase.*

```
public void runBare() throws Throwable {
	setUp();
	try {
		runTest();
	} finally {
		tearDown();
	}
}
```

Now, an object is needed which will verify that JUnit does indeed invoke these methods in the correct order. Below is an implementation of a *SpyTestCase* which collects the method names *runBare(), setUp(), runTest(), tearDown()* in the order in which they are invoked.

```
public class SpyTestCase extends TestCase {

	private List invokedMethodNames = new ArrayList();

	protected void runTest() throws Throwable {
```

```
			invokedMethodNames.add("runTest");
		}

		protected void setUp() throws Exception {
			invokedMethodNames.add("setUp");
		}

		protected void tearDown() throws Exception {
			invokedMethodNames.add("tearDown");
		}

		public List getInvokedMethodNames() {
			return invokedMethodNames;
		}
	}
```

A test is also needed to verify the order of those methods;

```
	public class TestCaseTest extends TestCase {
		public void testRunBareTemplate() throws Throwable {
			SpyTestCase spyTestCase = new SpyTestCase();
			spyTestCase.runBare();
			List expectedMethodNames = new ArrayList() {
				{
					add("setUp");
					add("runTest");
					add("tearDown");
				}
			};
			assertEquals(expectedMethodNames,
		spyTestCase.getInvokedMethodNames());
		}
	}
```

The second aspect (behavior) of a template method can be tested using the techniques in section **5.2.3.3 (Testing An Interface)**. Returning to the current example, it could be verified that invoking *TestCase.runBare()* causes a test to be executed, throwing an *AssertionFaledError* in the event of a failure.

```
	public void testRunBareExecutesAFailingTest() throws Throwable {

		TestCase testCase = new TestCase() {
			protected void runTest() throws Throwable {
				fail("Intentional failure");
			}
		};

		try {
			testCase.runBare();
			fail("Test should have failed!");
		} catch (AssertionFailedError expected) {
			assertEquals("Intentional failure", expected.getMessage());
		}
	}
```

Because the template method has no control over how a subclass implements a primitive operation, it should be assumed that the primitive operations can fail. A strategy for writing this kind of test can be overriding the spy and having it simulate the

desired failure in the appropriate primitive operation. Returning to JUnit, it must be verified that runBare() invokes tearDown() even when the test fails;

```
public void testRunBareInvokesTearDownOnTestFailure() throws Throwable {
        SpyTestCase spyTestCase = new SpyTestCase() {
                protected void runTest() throws Throwable {
                        super.runTest();
                        fail("I failed on purpose");
                }
        };

        try {
                spyTestCase.runBare();
        } catch (AssertionFailedError expected) {
                assertEquals("I failed on purpose", expected.getMessage());
        }

        List expectedMethodNames = new ArrayList() {
                {
                        add("setUp");
                        add("runTest");
                        add("tearDown");
                }
        };
        assertEquals(expectedMethodNames, spyTestCase.getInvokedMethodNames());
}
```

5.3 Testing in Dependent Environments

Designing and developing enterprise applications comes with its own challenges and complexities, such as managing huge amounts of complex data, complex business rules that generally fail all tests of logical reasoning, concurrent data access from multiple clients, complex and enormous number of user interface screens to handle the data and integration with other enterprise applications that were developed using diverse technologies.

In order to reduce the costs of enterprise application design and development, the **Java 2 Platform Enterprise Edition(J2EE)** "(WEB_19, 2005)" technology provides a component based approach to the design, development, assembly and deployment of enterprise applications. J2EE technology is comprised of several components and frameworks such as **Servlets** "(WEB_20, 2005)", **Java Server Pages** "(WEB_21, 2005)", **Enterprise Java Beans** "(WEB_22, 2005)". These technologies all consist of objects that are coded within a component framework and execute in the context of containers. This raises two main issues; performance and dependency; and a testable design should manage both effectively.

The first problem with testing in container environments is performance. Executing code inside a container brings an overhead that affects the execution speed of the tests. This has a negative effect on testing experience. One of the key practices in refactoring is to execute the tests after every change, to be sure a defect is not slipped into the code. Because many refactorings involve several changes a slow test suite discourages the developers from executing the tests frequently enough to realize most of the benefits of the refactoring safety net.

The second problem is the dependency problem. Frameworks are based on a different principle than class libraries; the framework code generally calls the developer code. That means the developer code depends directly on the framework code and framework design. Developers write code that extend the framework classes or implement the framework interfaces. Besides, the framework generally controls the lifecycle of classes or components and it decides on the flow of the code execution. This logic leads to some testing problems. There are two main approaches for solving this problem, **mock objects** and **reducing dependency**.

A **mock object** can be defined as an object that can be used in a test to stand in place of a more expensive to use or difficult to use object. The benefit to using mock objects is avoiding the expensive, external resource and test execution speed problems that come with it.

Rather than using a mock object right away, reducing the dependency on the frameworks should be the first choice to dependency problem. Briefly, this is accomplished by coding objects in such a way that a J2EE based implementation of the services it provides can be plugged in. If customer information is stored in a database, creating a *CustomerStore* interface and making the business logic depend only on this interface can be given as an example to this approach. Providing an in memory implementation of the *CustomerStore* interface and testing the business logic using that implementation removes the dependency to the database.

5.3.1 Testing and XML

XML documents have a significant space in enterprise applications. The configuration files, deployment descriptors, XHTML and XSL transformations, web services and many other technologies are based upon XML. Testing J2EE applications means writing tests involving XML documents. A practical way of accomplishing this is using **XPath** "(WEB_23, 2005)"**.** The XPath language defines a way to refer to the content of an XML document with query expressions that can locate anything from simple XML attribute values to large complex collections of XML elements. The overall strategy when testing with XML documents is to make assertions about those documents using XPath to retrieve the actual content as listed below;

```
assertEquals("Algan", document.getTextAtXpath("/person/lastName"));
```

This assertion says, the element *lastName* inside element *person* at the root of the document should have the text *Algan*. Following is an XML document that satisfies this assertion;

```
<?xml version="1.0?">
<person>
        <firstName>Fatih</firstName>
        <lastName>Algan</lastName>
</person>
```

XPath can be used to obtain the text of an element, to get the value of an attribute or to check whether an element exists. For example, an XML document with many person elements, checking a person with last name *Algan* can be done as below;

```
assertFalse(document.getNodesAtXpath("//person[lastName='Algan']")
        .isEmpty());
```

Here, the method *getNodesAtXpath()* works rather like a database query: "find all the person elements having a lastName element with the text *Algan* and return them as a collection." The assertion says, "The collection of person elements with the lastName of *Fatih* should not be empty." These are the two kinds of XPath statements that are used most often in testing XML data.

There is an XPath API that can be used to execute XPath queries on a parsed XML document. The two widely used implementations of the XPath API are found in **Xalan** "(WEB_24, 2005)" and **jaxen** "(WEB_25, 2005)"**.** These APIs can be used to make assertions about the structure and content of XML documents. A beter alternative can be using **XMLUnit** "(WEB_26, 2005)"**.** This package provides *XMLTestCase*, a base test case that adds custom assertions built on XPath. To use XMLUnit, a test case class named **org.custommonkey.xmlunit.XMLTestCase** is needed to be created. The tests are written as ususal. Various XMLUnit assertion classes help in testing whether than XML elements exist, in verifying their values and even in comparing entire XML documents.

```
public class MarshalPersonToXmlTest extends XMLTestCase {
        public void testFatihAlgan() throws Exception {
                Person person = new Person("Fatih", "Algan");
                XmlMarshaller marshaller = new XmlMarshaller(
                Person.class, "person");
                StringWriter output = new StringWriter();
                marshaller.marshal(person, output);
                String xmlDocumentAsString = output.toString();
                assertXpathExists("/person", xmlDocumentAsString);
                assertXpathEvaluatesTo("Fatih", "/person/firstName", xmlDocumentAsString);
                assertXpathEvaluatesTo("Algan", "/person/lastName",
                xmlDocumentAsString);
        }
}
```

The example above creates an XML document from a value object and tests this operation. Turning Java objects into XML documents and the other way around is called **XML Marshalling.** Web applications that use XSL transformations as their presentation engine typically need to marshal Java beans to XML. Other systems marshal data to and from XML to communicate with other computers in a heterogeneous environment, such as that created by web services. XMLUnit can be a convenient tool for testing in such enviornments. The assertions made in this test are the custom assertions that XMLUnit provides. The first used is *assertXpathExists()*, which executes the XPath query and fails only if there are no nodes in the XML document matching the query. Here this custom assertion is used to verify the existence of a person root element. The other custom assertion used is *assertXpathEvaluatesTo()*, which executes an XPath query and compares the result to an expected value; the assertion fails if the result and the expected value are different. XPath queries generally return either a list of nodes (in which the one expected can be searched for) or a String value (corresponding to the text of an element or attribute). XMLUnit provides the method *assertXMLEqual()*, which checks whether two XML documents are equal. The way that XMLUnit defines equality, though, requires some explanation. With XMLUnit, documents may be *similar* or *identical*. Documents are identical if the same XML elements appear in exactly the same order. Documents are similar if they represent the

same content, but perhaps with certain elements appearing in a different order. This way of interpreting whether documents are equal, viewing them as data, is common in applications, so *assertXMLEqual()* compares documents for similarity, rather than identity.

5.3.1.1 Verifying the Order of Elements in an XML Document

Sometimes the order in which the elements appear in an XML document changes the value the document represents. For instance, in a book, the order in which chapters, sections, and even paragraphs appear determines the meaning of the book. If a List of objects are being marshalled out to XML then the corresponding XML elements should appear in the same order as they were stored in the List. Using *assertXMLEqual()* to compare documents with this sensitivity to order, it might be observed that XMLUnit treats certain unequal documents as equal, and this is not the behavior desired. It is needed that XMLUnit be stricter in its definition of equality.

In XMLUnit, by default, two XML documents are *identical* if their node structures are the same, elements appear in the same order, and corresponding elements have the same value. If the elements are at the same level of the node's tree structure, but sibling elements *with different tag names* are in a different order, then the XML documents are not identical, but *similar*. The method *assertXMLEqual()* only verifies the *similarity* of two XML documents.

```
public class BuildArticleTest extends XMLTestCase {
        public void testMultipleParagraphs() throws Exception {
                XMLUnit.setIgnoreWhitespace(true);
                ArticleBuilder builder = new ArticleBuilder("An Article");
                builder.addAuthorName("An Author");
                builder.addHeading("A heading.");
                builder.addParagraph("This is a paragraph.");
                String expected =
                "<?xml version=\"1.0\" ?>"
                + "<article>"
                + "<title>An Article</title>"
                + "<author>An Author</author>"
                + "<p>This is a paragraph.</p>"
                + "<h1>A heading.</h1>"
                + "</article>";
                String actual = builder.toXml();
                assertXMLEqual(expected, actual);
        }
}
```

In the example above, an XML document is created with a title, an author, a heading and a paragraph. However, in the expected XML document the heading and the paragraph sections are switched; the heading ought to come before the paragraph, not after it. However, the test passes because the expected and actual documents are similar, but not identical. In order to avoid the problem the test can be changed as follows;

```
public void testMultipleParagraphs() throws Exception {
        XMLUnit.setIgnoreWhitespace(true);
        ArticleBuilder builder = new ArticleBuilder("An Article");
        builder.addAuthorName("An Author");
        builder.addHeading("A heading.");
```

```
		builder.addParagraph("This is a paragraph.");
		String expected =
		"<?xml version=\"1.0\" ?>"
		+ "<article>"
		+ "<title>An Article</title>"
		+ "<author>An Author</author>"
		+ "<p>This is a paragraph.</p>"
		+ "<h1>A heading.</h1>"
		+ "</article>";
		String actual = builder.toXml();
		Diff diff = new Diff(expected, actual);
		assertTrue("Builder output is not identical to expected document",
		diff.identical());
	}
```

First, XMLUnit is asked to return an object representing the differences between the two XML documents. Then an assertion is made on the *Diff*, expecting it to represent identical documents, that is, the corresponding elements appear in the expected order.

5.3.1.2 Testing an XSL Stylesheet

In spite of the wide use of Java Server Pages in the J2EE community, it is not the only way to build the presentation layer of a web application. One strong alternative is XSL transformation, which takes data in XML format and presents it as HTML to the end user. This approach has one great benefit over using Java Server Pages, XSL transformation does not require an application server. This means that such a presentation layer can be tested entirely outside any kind of J2EE container. All that is needed is parser and transformers such as Xerces and Xalan.

The general approach to testing an XSL stylesheet consists of transforming a representative set of XML documents and verifying the result. Simple XML documents can be hardcoded in the tests as Strings and then XSL transformation is applied. Assertions can be made on the resulting XML document. As with any other XML document tests, XMLUnit provides two main strategies for verifying content; XPath based assertions on parts of the actual document, or comparing an expected document against the actual document. XMLUnit provides some convenience methods for doing either.

The example below presents a shopcart document from an imaginary on line shopping application. The structure of the shopcart is simple; a shopcart contains items and a subtotal. Each item descirbes its own details; the item name, quantity, unit price and total price. The snippet below shows a sample XML document in described format;

```
<?xml version="1.0" encoding="UTF-8"?>
	<shopcart>
		<item id="762">
			<name>Turkish Coffee</name>
			<quantity>1</quantity>
			<unit-price>$7.25</unit-price>
			<total-price>$7.25</total-price>
		</item>
		<item id="903">
			<name>French Vanilla</name>
```

```
                <quantity>2</quantity>
                <unit-price>$6.50</unit-price>
                <total-price>$13.00</total-price>
        </item>
        <item id="001">
                <name>Java</name>
                <quantity>3</quantity>
                <unit-price>$8.00</unit-price>
                <total-price>$24.00</total-price>
        </item>
        <subtotal>$44.25</subtotal>
</shopcart>
```

As a first test, it can be verified that the stylesheet correctly renders an empty shopcart. The test should ideally verify the content, not the look and feel of the resulting web page. If the tests depend too much on the layout of web pages they become overly sensitive to purely cosmetic changes. The test below expects to see an HTML table for the shopcart, no rows representing shopcart items and a $0.00 subtotal.

```
public class DisplayShopcartXslTest extends XMLTestCase {

private String displayShopcartXslFilename =
"../CoffeeShopWeb/Web Content/WEB-INF/template"
+ "/style/displayShopcart.xsl";

private Source displayShopcartXsl;

protected void setUp() throws Exception {
        displayShopcartXsl = new StreamSource(
        new FileInputStream(displayShopcartXslFilename));
}

public void testEmpty() throws Exception {
        String shopcartXmlAsString =
                "<?xml version=\"1.0\" ?>"
                + "<shopcart>"
                + "<subtotal>$0.00</subtotal>"
                + "</shopcart>";
        Document displayShopcartDom =
        doDisplayShopcartTransformation(shopcartXmlAsString);
        assertXpathExists("//table[@name='shopcart']",
        displayShopcartDom);
        assertXpathEvaluatesTo("$0.00",
        "//table[@name='shopcart']//td[@id='subtotal']",
        displayShopcartDom);
        assertXpathNotExists("//tr[@class='shopcartItem']",
        displayShopcartDom);
}

public Document doDisplayShopcartTransformation(
        String shopcartXmlAsString) throws TransformerConfigurationException,
        TransformerException {
        Source shopcartXml = new StreamSource(
                new StringReader(shopcartXmlAsString));
        Transform transform = new Transform(shopcartXml, displayShopcartXsl);
        return transform.getResultDocument();
}
}
```

The method `doDisplayShopcartTransformation()` performs the transformation to be tested. It uses the XMLUnit class *Transform* to simplify applying the transformation and to retrieve the resulting document. The test builds an empty shopcart XML document as a String, applies the transformation, then makes XPath-based assertions on the resulting document. In particular, it expects the following things:

- A table representing the shopcart, which the test finds by examining its name.
- A table data cell (<td>) inside the table containing the shopcart subtotal amount, which the test finds by examining its ID.
- No table rows (<tr>) inside the table using the stylesheet class *shopcartItem,* which is how the test detects the existence of shopcart items on the page.

An alternative technique may be comparing an expected web page against the actual web page generated by the transformer. This kind of a test is easier to write but can be difficult to maintain because purely cosmetic changes to the presentation layer requires the expected web pages to be updated as well.

5.3.2 Testing and Databases

Almost all enterprise applications require persistent data. Persistence is one of the fundamental concepts in application development. The dominant technology today used in persisting data is relational databases. Relational databases are the most flexible and robust approach to data management in hand today, thus it is a common denominator of many disparate systems and technology platforms. Hence, the relational data model is often the common enterprise wide representation of business entities.

Although relational technology adapts greatly to storage, organization and retrieval of data as well as concurrency and data integrity, an object oriented application with a domain model does not work directly with the tabular representation of the business entities; the application has its own, object oriented model of the business entities. The business logic interacts with this object oriented domain model and its runtime realization as a graph of interconnected objects. Thus, the business logic is never executed in the database as an SQL stored procedure. This allows business logic to make use of sophisticated object oriented concepts such as inheritance and polymorphism.

Using relational and object technology together in an application brings an impedance mismatch and the solution for the problems resulted by this mismatch can require a significant time and effort. One of the major costs is in the area of modeling. The relational and object models must both encompass the same business entities. However, an object oriented purist will model these entities in a very different way than an experienced relational data modeler. The usual solution to this problem is to bend and twist the object model until it matches the underlying relational technology. This can be done successfully, but only at the cost of losing some of the advantages of object orientation. Relational modeling is underpinned by relational theory. Object orientation has no such rigorous mathematical definition or body of theoretical work. Thus, mathematics cannot help to bridge the gap between the two paradigms. A further cost is the API to SQL databases in object languages. They provide a command oriented approach to moving data to and from an SQL database. A structural relationship must be specified at least three times (Insert, Update, Select), adding to the time required for

design and implementation. The unique dialect for every SQL database does not improve the situation.

All the problems mentioned above makes handcoding a persistence layer in an object system costly and unmaintainable. However, in recent years, a new method for interacting with databases, called object relational mapping(ORM), has become popular, especially among the Java community. ORM automates persistence of objects in an object application to the tables in a relational database, using metadata that describes the mapping between the objects and the database. An ORM framework generally intends to provide services listed below;

- An API for performing basic Create/Read/Update/Delete operations on objects of persistent classes.
- A language or API for specifying queries that refer to classes and properties of classes.
- A facility for specifying mapping metadata
- A technique for the ORM implementation to interact with transactional databases to perform dirty checking, lazy association fetching and many other optimization functions.

One of the key qualities of an ORM implementation is its transparency. Frameworks that imposes special programming models and dependent environments such as containers for persistent objects(domain entities), which couple the framework code to the application code prevents developers from concentrating on the business problem at hand. Besides, they make unit testing the domain model of the application in isolation much harder. In addition to these, testing persistence logic brings a dependency to the database system. Initialization of the database with adequate data everytime and bringing it to an initial consistent state after each test run should be carefully organized.

5.3.2.1 A Persistency Framework

The ORM implementation chosen for this study is Hibernate “(WEB_26, 2005)”. Hibernate has been accepted as one of the most flexible, powerful and popular open source ORM solution among the Java community. One of the most valuable features of Hibernate is that it allows testing the domain model of an application completely in isolation. The class *Category* listed below is a domain object from a tree structure of categories which might include other categories or items that both implement the interface *ICategorizable*;

```
public class Category implements ICategorizable, Serializable {
protected Long id;
protected String definition;
protected Set subCategories = new TreeSet(new CategoryComparator());
protected ICategorizable parentCategory;
protected int version;

private Category() {
        super();
}
```

```
public Category(String definition) {
this.definition = definition;
}

public Long getId() {
return id;
}

public void setId(Long id) {
this.id = id;
}

public String getDefinition() {
return definition;
}

public void setDefinition(String definition) {
this.definition = definition;
}

public void addCategory(ICategorizable category) {
if(category == null)
throw new IllegalStateException("Cannot add empty category.");
if(category.getParentCategory() != null)
            category.getParentCategory().getSubCategories()
                  .remove(category);
category.setParentCategory(this);
getSubCategories().add(category);
}

public void removeCategory(ICategorizable category) {
if (category!=null){
      getSubCategories().remove(category);
      category.setParentCategory(null);
}
}

public Set getSubCategories() {
return subCategories;
}

public ICategorizable getParentCategory() {
return parentCategory;
}

public void setSubCategories(Set subCategories) {
this.subCategories = subCategories;
}

public void setParentCategory(ICategorizable parentCategory) {
this.parentCategory = parentCategory;
}

public String toString() {
return getDefinition();
}

private int getVersion() {
      return version;
}
```

```
private void setVersion(int version) {
        this.version = version;
}

public int hashCode() {
        return new HashCodeBuilder(-11241015, -
        1392753393).append(getDefinition()).toHashCode();
}

public boolean equals(final Object other) {
        if (this == other) return true;
        if (!(other instanceof Category)) return false;
        Category castOther = (Category) other;
        return new EqualsBuilder().append(getDefinition(), castOther.getDefinition()).append(
        getParentCategory(), castOther.getParentCategory()).isEquals();
}

}
```

Although no persistence concern leaks into the code of domain logic, there are some subtle rules to obey;

- An identifier property(the *id* attirbute in class *Category*) is needed for database identity. Identifier properties should not be primitive types, Hibernate requires Object types for identifiers.
- The association *subcategories* is a Collection type. However, Hibernate requires that association properties must be of interface types(*Set*, *Map*, *List*) rather than concrete implementations(*Hashset*, *Hashtable*, *ArrayList*) because Hibernate wraps the association types within its own implementations of *Set*, *Map* and *List* under the covers.
- An empty parameter constructor is required for each persistent object. However, it can be private.
- All persistent attributes must conform to JavaBeans conventions. They need accessor and mutator methods as defined in JavaBeans specification. However, the accessor or mutator methods can be private if information hiding is desired.
- All persistent objects must override *equals()* and *hashcode()* methods if optimistic locking and support for automatic dirty checking in application transactions is needed.
- A version property must be defined and mapped to a database column if optimistic locking in application transactions is needed.

The Category class can easily be tested within its domain context as can be seen below;

```
public void testAddCategory() {
        ICategorizable newFirstLevel=getObjectFromProperties("newFirstLevel");
        root.addCategory(newFirstLevel);
        assertTrue(root.hasInChildren(newFirstLevel));
}

public void testAddNullCategory() {
        try {
                ICategorizable subCategory = null;
                root.addCategory(subCategory);
```

```
            fail("Adding a null Category should throw an IllegalStateException");
        } catch(IllegalStateException e) {

        }
    }
```

5.3.2.2 Testing Persistency Logic

A typical object oriented architecture comprises layers that represent different concerns. Consequently, it is a best practice to group all classes and components responsible for persistence into a seperate *persistence layer* in *a layered system architecture*. The **Data Access Object** "(WEB_8, 2005)" pattern is commonly used as a best practice among the Java community. The authors of Hibernate also recommend using this pattern in the persistency layer. However, since Hibernate makes use of transitive persistence, the data access logic generally consists only of a set of finder methods which loads a set of persistent object instances from the database into the memory.

```
public class CategoryDAO implements ICategoryDAO {
public ICategorizable findRootCategory() {
ICategorizable cat = null;
try {
Session session = HibernateUtil.getSession();
cat = (ICategorizable)session.createQuery("from
                Category as cat " +
"where cat.parentCategory is null").uniqueResult();
} catch(HibernateException he) {
throw new InfrastructureException(he.getMessage());
}
return cat;
}

public ICategorizable findCategoryById(Long catId) {
ICategorizable cat = null;
try {
Session session = HibernateUtil.getSession();
                cat=(ICategorizable)session.get(Category.class,
                catId, LockMode.UPGRADE);
} catch(HibernateException he) {
throw new InfrastructureException(he.getMessage());
}
return cat;
}

public Collection findCategoryByMatchingDef(String defString) {
Collection returnVal = null;
try {
Session session = HibernateUtil.getSession();
        returnVal=session.createCriteria(Category.class).
                add(Expression.like("definition",
                defString, MatchMode.ANYWHERE)).list();
} catch(HibernateException he) {
throw new InfrastructureException(he.getMessage());
}
return returnVal;
}
```

```
}
```

When the persistence logic is isolated from the domain classes as seen in class *CategoryDAO* above, it can easily be tested outside of any domain logic context. One alternative in how the tests will be written is providing the mock versions of Hibernate specific classes such as *Session*, *Query*. The mock versions will not query a real database and dependency to the database will be removed completely. Although this method isolates framework specific code from the data access logic of the application, it may not be the most practical solution. An in memory running database would prevent the performance problems that emerges from testing with a real database. Hibernate allows automatic creation of a database schema from the mapping metadata files so the database schema can be created and dropped between test runs automatically. Initializing the database with test data can also be accomplished using the transitive persistence capabilities of Hibernate just with a few lines of code;

```
protected void setUp() throws Exception {
super.setUp();
categoryType="Item Defintions";
root = new Category("Items");
child1 = new Category("Items Level 1");
child2 = new Category("Items Level 2");
child11 = new Category("Items Level 11");
child111= new Category("Items Level 111");
root.addCategory(child1);
root.addCategory(child2);
child1.addCategory(child11);
child11.addCategory(child111);
Session session = HibernateUtil.getSession();
HibernateUtil.beginTransaction();
session.save(root);
HibernateUtil.commitTransaction();
HibernateUtil.closeSession();
}

public void testFindRootCategory() {
ICategorizable temp = getCategoryDAO().findRootCategory();
assertEquals(temp.getDefinition(),root.getDefinition());
}


public void testFindCategoryById(){
ICategorizable root = getCategoryDAO().findRootCategory();
assertEquals(root,getCategoryDAO().findCategoryById(root.getId()));
}

…//Other test methods here

public void tearDown() {
//delete all categories in the database here with a regular SQL query
…
}
```

The persistence layer above does not include any functionality for any create update or delete operations. This is because the Hibernate framework automatically synchronizes the state changes of persistent objects to the database. Such functionality

can be tested with the integration tests which are built above the domain logic. These tests validate the application functionality from a higher granularity.

5.3.3 Testing Enterprise Java Beans

It is a common practice to design enterprise applications in accordance with the **Model View Controller(MVC)** "(WEB_9, 2005)" pattern. A typical J2EE application does much of its work in the model components; executing business logic and providing access to business data. Because the majority of the effort is spent on the model when building an application, it must be built to be tested easily. Beyond the effort to fix the problems in the model, those problems tend to have more critical impact on the system.

Enterprise Java Beans(EJB) "(WEB_22, 2005)" is a J2EE technology for implementing model components. EJB is not a suitable technology for unit testing because EJB components are tightly coupled with a runtime container and requires a special programming model. The most flexible design strategy to make EJB components easier to test is to make the EJBs themselves as thin as possible, just as one might do with view components. Placing most of the code in **Plain Old Java Objects(POJO)** "(WEB_10, 2005)" and using EJBs as simple wrappers that do little more than delegating to other Java objects minimizes the code and runtime dependency to the EJB container. The overall strategy is to be able to execute as much model component code as possible without involving an EJB container, because that allows tests to be simple and execute quickly.

5.3.3.1 Testing Stateless Session Beans

Each unit of discrete business logic is implemented as a method on a stateless session bean. This discrete unit of logic corresponds to a **Transaction Script** "(Fowler, 2002)". Because each transaction script is meant to execute in isolation from the others, it makes sense to test each session bean method independently. In order to emphasize independency, each transaction script should be tested by a different test fixture in the beginning. Afterwards, similar test fixtures, that is methods with same or similar parameters can be moved into a single fixture as they are recognized. The fact that several transaction scripts might be packaged into a single session bean class does not imply that their tests should be similarly organized. The general rule of "one test case class per production class" makes sense when the classes participate in object-oriented relationships and implement cohesive objects; however, stateless session beans are not really objects in the pure object oriented sense, they are collections of procedures.

Among the different types of EJBs, stateless session beans are the easiest to test. They are excellent candidates for refactoring. The goal is to extract all the business logic out of the session bean method into a POJO. It then becomes possible to test the logic without involving the EJB container. Some session beans invoke other objects they retrieve through JNDI lookups. For these beans, first performing the lookup in the session bean method and then passing the resulting object as a parameter to the newly extracted POJO's business logic method is the way to go. This is another way to keep business logic unaware of its environment.

When the business logic is extracted away from a session bean, it merely plays the role of a **Remote Facade** "(Fowler, 2002)", and generally becomes too simple to break.

5.3.3.2 Testing Stateful Session Beans

Stateful session beans are generally used only in the cases where Access to an HTTP session object is not available, such as when building a rich client application that works over **RMI/IIOP** "(WEB_27, 2005)" instead of HTTP. In general, testing a stateful session bean is not much different than testing a stateless session bean. One fundamental difference between testing a stateful session bean and a stateless session bean has to do with scale; each stateless session bean method is treated as a separate unit to test whereas a stateful session bean (and all its methods together) is generally organized as a separate unit to test. The techniques for writing these tests remain the same as for stateless session beans; decouple the business logic from the session bean when possible.

The following example demonstrates a simple shoppingcart session bean. The *create()* method creates an empty shopcart and the methods *addToShopcart()* and *getShopcartItems()* behave as their names suggest.

```
public class ShopcartOperationsBean implements
        javax.ejb.SessionBean {

        private Map coffeeQuantities;

        public void ejbCreate() throws javax.ejb.CreateException {
                coffeeQuantities = new HashMap();
        }

        public void addToShopcart(Vector requestedCoffeeQuantities) {
                for (Iterator i = requestedCoffeeQuantities.iterator();
                        i.hasNext();) {
                        CoffeeQuantity each = (CoffeeQuantity) i.next();
                        String eachCoffeeName = each.getCoffeeName();
                        CoffeeQuantity currentQuantity;
                        if (coffeeQuantities.containsKey(eachCoffeeName)) {
                        currentQuantity = (CoffeeQuantity)
                        coffeeQuantities.get(eachCoffeeName);
                        } else {
                        currentQuantity = new CoffeeQuantity(0, eachCoffeeName);
                        coffeeQuantities.put(eachCoffeeName, currentQuantity);
                        }
                        coffeeQuantities.put(eachCoffeeName,
                        currentQuantity.add(each));
                }
        }

        public Vector getShopcartItems() {
                return new Vector(coffeeQuantities.values());
        }
}
```

The test for the shoppingcart bean should simulate two transactions, in order to verify that conversational state is preserved. The test scenario adds coffees A and B to shopcart first, and then adds coffees A and C.

```
public void testComplexCase() throws Exception {
        // Create a shopcart somehow!
        Vector coffeeQuantities1 = new Vector();
        coffeeQuantities1.add(new CoffeeQuantity(2, "A"));
        coffeeQuantities1.add(new CoffeeQuantity(3, "B"));
        shopcart.addToShopcart(coffeeQuantities1);
        assertEquals(coffeeQuantities1, shopcart.getShopcartItems());
        Vector coffeeQuantities2 = new Vector();
        coffeeQuantities2.add(new CoffeeQuantity(1, "A"));
        coffeeQuantities2.add(new CoffeeQuantity(2, "C"));
        shopcart.addToShopcart(coffeeQuantities2);
        Vector expectedTotalQuantities = new Vector();
        expectedTotalQuantities.add(new CoffeeQuantity(3, "A"));
        expectedTotalQuantities.add(new CoffeeQuantity(3, "B"));
        expectedTotalQuantities.add(new CoffeeQuantity(2, "C"));
        assertEquals(expectedTotalQuantities, shopcart.getShopcartItems());
}
```

The test case above does not include the EJB related code. Only the business logic related code is shown. This is the part which must be extracted out from the session bean. A new domain model object representing the shopcart business logic is listed below;

```
public class ShopcartLogic {
        private Map coffeeQuantities;
        public ShopcartLogic() {
                coffeeQuantities = new HashMap();
        }

        public void addToShopcart(Vector requestedCoffeeQuantities) {
                for (Iterator i = requestedCoffeeQuantities.iterator();
                i.hasNext();) {
                        CoffeeQuantity each = (CoffeeQuantity) i.next();
                        String eachCoffeeName = each.getCoffeeName();
                        CoffeeQuantity currentQuantity;
                        if (coffeeQuantities.containsKey(eachCoffeeName)) {
                        currentQuantity =(CoffeeQuantity)coffeeQuantities.get(
                        eachCoffeeName);
                        } else {
                        currentQuantity =new CoffeeQuantity(0,eachCoffeeName);
                        coffeeQuantities.put(eachCoffeeName,currentQuantity);
                        }
                        coffeeQuantities.put(eachCoffeeName,
                        currentQuantity.add(each));
                }
        }

        public Vector getShopcartItems() {
                return new Vector(coffeeQuantities.values());
        }
}
```

Since the business logic in the session bean is extracted into a seperate class, the session bean becomes too simple to break, it does nothing more than delegate method invocations to another object.

```
public class ShopcartOperationsBean implements javax.ejb.SessionBean {
private javax.ejb.SessionContext mySessionCtx;
private ShopcartLogic shopcart;

public void ejbCreate() throws javax.ejb.CreateException {
        shopcart = new ShopcartLogic();
}
public void addToShopcart(Vector requestedCoffeeQuantities) {
        shopcart.addToShopcart(requestedCoffeeQuantities);
}
public Vector getShopcartItems() {
        return shopcart.getShopcartItems();
}
}
```

This example demonstrates clearly how a testable design is a beter design. By following the refactoring presented, an EJB have become what it should have been all along; merely a wrapper that provide the benefits and services of the container to the application.

5.3.3.3 Testing CMP Entity Beans

One of the main features of Container Managed Enterprise Java Beans is that the application server provides the vast majority of entity bean code. Because the entity bean code is tightly coupled with the runtime container and most of the code belongs to the platform it may be seen reasonable not to test an entity bean at all. The programmer's task in developing entity beans is mostly defining interfaces and methods. However, the programmer still has to specify the entity beans in enough detail so that the container can fill in the rest. Consequently, testing the CMP bean meta data rather than the bean itself saves the programmer from trying to test the platform. However, there are times when testing only the meta data does not give enough confidence. One of these cases is when the application server is not a mature one and might produce bugs.

The most direct way to test a CMP entity bean is to test it inside the container. **Apache Cactus** "(WEB_28, 2005)" is the best known choice for testing J2EE components inside the application server container. Testing a CMP entity bean requires testing the following features;

- The mapping between the entity bean class and the database table.
- The mapping between the entity bean container managed fields and the table columns.
- Container managed relationships(CMR).
- All the finder methods of the entity bean.

Testing security and transaction attributes should go into the integration tests because these issues apply to more than just the entity beans. The class/table and field/column mappings can be tested by storing a few entity beans and then loading

them back. The CMR relationships can be tested by storing an empty collection of related objects, loading it back, adding one or two related entity beans to the collection and then storing once again. As for the finding methods, the only way is setting up test data, invoking the finder and verifying the results.

The application server writes code to handle many situations in the case of CMP entity beans such as verifying the entity bean in the presence of null values, its unique indices, duplicate keys or invalid foreign keys. In order to stay away from testing the platform, the programmer ought to focus on testing how to react to those situations. Presumably, there are session beans that invoke the CMP entity beans. Rather than testing whether the EJB container correctly throws a *DuplicateKeyException*, testing the session bean on how it reacts when the entity bean throws a *DuplicateKeyException* is the way to go.

One alternative that comes to mind when trying to test CMP entity beans is through end to end tests. However, end to end tests are not well suited to isolate defects, as they involve the entire application. Secondly, testing all the boundary conditions with a web container, an EJB container, EJB to EJB communication which relies on a distributed object model, all the business logic around them and the database would take a considerably long time to test, which makes continious testing practically infeasible. If CMP entity beans are to be tested inside the container, they should be tested in isolation. One more point to be careful in testing entity beans is setting up and tearing down test data in the database. If the tests uses JDBC or a tool such as DBUnit to create and remove test data while the EJBs are running, then the entity beans should not be deployed using the "**commit option A**" which means the entity beans have exclusive access to the database. The entity beans would not have exclusive access in such a scenario, so if the tests change live data in the middle of a transaction the tests will fail because of unexpected duplicate key or foreign key problems. Besides, the entity beans will appear to have stale data during the tests, but not in production.

5.3.3.3.1 Testing CMP Metadata Outside the Container:

If testing the entity bean in a live container takes too much time, then what can be done is testing the entity bean meta data instead. This meta data is typically expressed in XML, so it is simple to test and **XMLUnit** "(WEB_26, 2005)" is a suitalbe tool for testing it.

```
public class CoffeeCatalogItemEntityBeanMetaDataTest extends XMLTestCase {
private static final String META_DATA_FILENAME =
        "../CoffeeShopEJB/ejbModule/META-INF/jbosscmp-jdbc.xml";
private static final String ENTITY_BEAN_XPATH =
        "/jbosscmp-jdbc/enterprise-beans/" +
        "entity[ejb-name='CoffeeCatalogItem']/";
private Document metaDataDocument;

protected void setUp() throws Exception {
        XMLUnit.setIgnoreWhitespace(true);
        metaDataDocument = XMLUnit.buildTestDocument(
        new InputSource(new FileReader(META_DATA_FILENAME)));
}
public void testTableMapping() throws Exception {
        assertXpathEvaluatesTo("catalog.beans",
        ENTITY_BEAN_XPATH + "table-name",
```

```
			metaDataDocument);
	}

	public void testFieldMapping() throws Exception {
			assertXpathEvaluatesTo("productId", ENTITY_BEAN_XPATH
			+ "cmp-field[field-name='productId']/column-name",
			metaDataDocument);
			assertXpathEvaluatesTo("coffeeName", ENTITY_BEAN_XPATH
			+ "cmp-field[field-name='coffeeName']/column-name",
			metaDataDocument);
			assertXpathEvaluatesTo("unitPrice", ENTITY_BEAN_XPATH
			+ "cmp-field[field-name='unitPrice']/column-name",
			metaDataDocument);
	}

	public void testDataSource() throws Exception {
			assertXpathEvaluatesTo(
			"java:/jdbc/mimer/CoffeeShopData",
			"/jbosscmp-jdbc/defaults/datasource",
			metaDataDocument);
	}
	}
```

The test case above tests the metadata of an imaginary *CoffeCatalogItem* entity bean, representing an item in the catalog of the CoffeShop application. The *catalog.beans* database table provides the persistent storage for this entity bean.

5.3.3.4 Testing Message Driven Beans

The asynchronous nature of message driven beans makes it difficult for the test to make assertions on the result. In section 5.2.3.1, "Testing the return value of a method ", how to test a method that has no return value was explained. The same issue apply to message driven beans as well because the message handling method *onMessage()* returns no value. However, in the case of a message driven bean, even the method under test is not invoked by the programmer code. The program code sends the container a message, and then the container invokes the appropriate message driven bean. There is no way to obtain a reference to the message driven bean through a JNDI lookup, so there is no way to observe the side effects *onMessage()* has. The object under test is in another JVM, executing on another thread, and there is no way to obtain a reference to it. This probably might be a worst case scenario for object testing. There is no practical way to write an isolated object test for a message driven bean running in a live container. All that can be done is to send a message to the appropriate destination, wait long enough for the message to be delivered and processed, and then observe whatever external side effects comes from processing the message. If the message driven bean updates the database, then a test against the live database is needed. If the bean sends an e-mail, then testing against a live e-mail server is required. Consequently, testing message driven beans outside the container would be the only practical choice in hand.

A message driven bean is two things at once, a JMS message consumer and an EJB. Therefore, the first thing that should be done is to identify the code that requires the container and seperating it from the rest of the code. This part of the code either performs a JNDI lookup or uses the *MessageDrivenContext* object provided by the

container. The container specific code should be kept in the message driven bean and the rest should be moved to a new class. Here, mock objects approach can be taken to testing whatever the message driven bean does with the *MessageDrivenContex* and JNDI lookups. **MockEJB** "(WEB_29, 2005)" is a suitable tool for implementing this approach. The next step is to seperate the JMS consumer behavior into three main tasks; receiving the message, processing the message and replying to the message. Here, replying stands for *replying with another JMS message*. If the bean processes the incoming message and for example, writes the result of some transaction to the database, than that act is considered a part of processing the message, not replying to it. Extracting the message processing code into a seperate class and leaving the rest behind will create a new class with methods entirely unaware of messaging or EJBs. It will just be a POJO(Plain Old Java Object) and can easily be tested by using one of the techniques explained before.

The resulting design is essentially a thin EJB wrapper around a medium sized JMS wrapper around the message receiving logic as seen in **Figure 5.2.**

Although the JMS message consumer is merely a wrapper around the message processing logic, it is almost never *too simple to break*. In the minimum, the message consumer will have to cast the *Message* to the appropriate type and unpack data from the message. If there is a possibility that type cast will fail, the message producer has to be tested, not the consumer. It is the responsibility of the producer to comply with the type of message the consumer expects. If unpacking the data might fail, then that logic has to be tested by invoking *jmsConsumer.handleMessage()* and passing in a Spy version of the messge processing logic class.

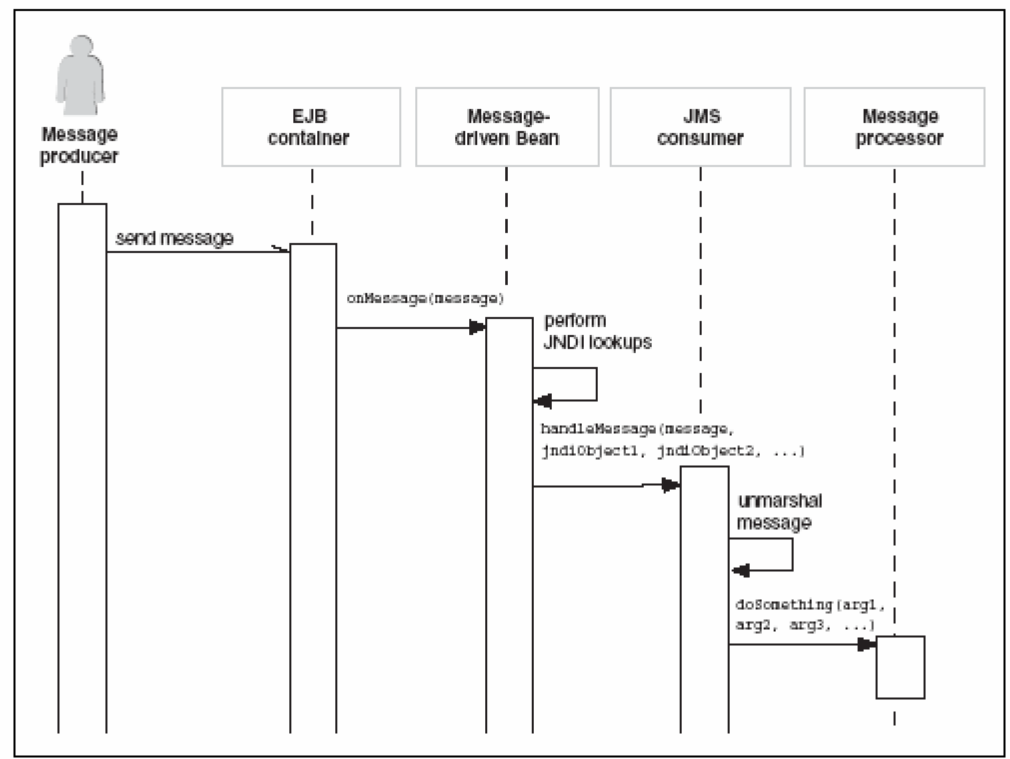

Figure 5.2 A proposed design to test a Message Driven Bean

5.3.3.4.1 Testing a JMS Message Consumer

The difficulties in testing a message driven bean can be overcome by testing it in isolation as a JMS message consumer. The simplest solution is simply instantiating the message listener directly and use a mock objects approach to substitute test objects in place of the message consumer's collaborators. However, there still are some issues that is needed to be aware of during testing;

- Communication via JMS is still entirely asynchronous. For example, a message consumer cannot throw an exception and expect the producer to receive it.
- JMS message consumers are typically deployed in a J2EE application server as message driven beans. Although it is not mandatory to do so, wrapping the message consumer in a message driven bean helps the programmer leverage the EJB container's services such as participating in transactions, guaranteed delivery, etc.

In section 5.3.3.4, *"Testing Message Driven Beans"* an overall design for a message driven bean was propsed. It was a **Decorator** "(Gamma, Helm, Johnson, Vlissides, 1995)" like approach, starting with message processing logic, wrapping it with a JMS message consumer and then wrapping both again in a message driven bean. The same pattern can be applied here as well, ignoring the last layer. This allows testing the message receiving logic, interaction with the JMS API, without having to rely on the correctness of the business logic that responds to the message.

The following example test case tests an order processing message consumer, *ProcessOrderSubmissionMessageListener*. The consumer retrieves a collaborator from the JNDI directory, unmarshals the message, and then executes a *ProcessOrderSubmissionAction* which has the message processing logic. Because the message listener is instatiated by the test, a Spy version of the action can be substituted. This allows to verify that the message consumer invokes the action with the parameters it received from the message.

```
public class ProcessOrderSubmissionMessageListenerTest extends TestCase
        implements MailService {
private boolean invoked;

protected void setUp() throws Exception {
        invoked = false;
        MockContextFactory.setAsInitial();
        new InitialContext().bind("java:comp/env/service/Mail", this);
}

public void testHappyPath() throws Exception {
        ProcessOrderSubmissionAction spyAction =
        new ProcessOrderSubmissionAction() {
                public void processOrder(MailService mailService,
                String customerEmailAddress) {
                        invoked = true;
                        assertEquals("fatihalgan@iyte.edu.tr",
                        customerEmailAddress);
                }
        };
        ProcessOrderSubmissionMessageListener consumer =
                new ProcessOrderSubmissionMessageListener(spyAction);
        MapMessage message = new MapMessageImpl();
```

```
		message.setString(ProcessOrderSubmissionMessageListener
		.CUSTOMER_EMAIL_PARAMETER_NAME, "fatihalgan@iyte.edu.tr");
		consumer.onMessage(message);
		assertTrue("Did not invoke the processing action.", invoked);
	}

	public void sendMessage(String fromAddress, String toAddress,
		String subject, String bodyText) {
		fail("No-one should invoke me.");
	}
	}
```

The test creates a Spy version of the “process submitted order” action, passes it to the JMS message consumer, simulates sending a message, and then verifies that the message consumer invoked the action with the correct customer e-mail address. The test case also implements a dummy e-mail service interface named *MailService()*, however the test fails by default when it is called since sending e-mail is part of the message processing action and should not be accomplished by the message consumer.

5.3.3.4.2 Testing the Message Processing Logic

In section 5.3.3.4, *“Testing Message Driven Beans”* a recommended design was presented; The EJB container delivers the messge to a message driven bean which performs JNDI lookups, and then passes the message to a JMS message consumer which unmarshals the message and passes it to a message processor. This section focuses on testing the message processor logic. The message processor should have no knowledge of JMS or messaging. If this principle can be preserved, the message processor can be treated as any other POJO(Plain Old Java Object) and can be tested in a straight and simple manner.

Returning to the example of receiving and processing an order, the message processor should send an e-mail to the customer saying that her order is received. As explained in the former section(5.3.3.4.2), the mail service was already isolated into an interface called *MailService.* This simplifies the corresponding tests.

```
	public class ProcessOrderSubmissionActionTest extends TestCase {
	private boolean spyMailServiceInvoked = false;
	public void testToAddress() throws Exception {
		ProcessOrderSubmissionAction action =
		new ProcessOrderSubmissionAction();

		MailService spyMailService = new MailService() {
			public void sendMessage(String fromAddress,
			String toAddress, String subject,String bodyText) {
				assertEquals("fatihalgan@iyte.edu.tr", toAddress);
				spyMailServiceInvoked = true;
			}
		};
		action.processOrder(spyMailService, "fatihalgan@iyte.edu.tr");
		assertTrue(spyMailServiceInvoked);
	}
	}
```

5.3.3.4.3 Testing the Message Producer

The majority of the code written to send a JMS message is what can be called as a *“JMS noise”*. There is a large, repetitive structure of code to write before even sending a message as simple as “Hello”. The most direct way to test a JMS producer is to start a messaging server, connect a listener to the appropriate message queue, create a message, send it and then verify that the listener received it. However, this kind of test is only good for testing if the messaging server is configured correctly. After writing one small set of deployment and configuration tests, the focus will shift on different key points; Is the right message being sent? Are they being sent to the right place?

When testing a message producer, first the JMS noise part should be refactored out to a seperate class. There ought to be a simpler API for sending simple messages. Following is an interface, which just sends a *MapMessage* to a particular destination.

```
public interface MapMessageSender {
void sendMapMessage(String destinationQueueJndiName,
        Map messageContent) throws MessagingException;
}
```

To send a *MapMessage* only the JNDI name of the destination *Queue* and a *Map* containing the message content has to be provided. An implementation of this interface will do the rest. In order to test the message producer without running the message server its key reponisibilities must be seperated; creating the message content(not the *Message* object itself but its contents), specifying the message destination and using the JMS server.

Testing the message content generator should be done in isolation. The more complex the message cnotent, the more important testing becomes. With a *MapMessage* for example, the ability to add data to a *MapMessage* from a Map can be extracted into a seperate method or class and reused afterwards. With *MapMessage*, it can be overlooked that *Message.setObject()* only support the primitive wrapper classes(*Integer, Long*, etc.) and String, but not arbitrary objects. Overlooking this constraint is enough to get wrong, so it is enough functionality to test on its own. The following code snippet shows an example of such a test, which tries to add an *ArrayList* object to a *MapMessage*.

```
public class BuildMapMessageTest extends TestCase {
private MapMessageImpl mapMessage;
private MessageBuilder messageBuilder;

protected void setUp() throws Exception {
        mapMessage = new MapMessageImpl();
        messageBuilder = new MessageBuilder();
}

public void testGenericObject() throws Exception {
        Map singleton = Collections.singletonMap("b", new ArrayList());
        try {
                messageBuilder.buildMapMessage(mapMessage, singleton);
                fail("Added a generic object to a MapMessage!");
        } catch (MessagingException expected) {
                Throwable throwable = expected.getCause();
                assertTrue("Wrong exception type",
                throwable instanceof MessageFormatException);
        }
```

```
}
}
```

Depending on what types of messages are used, similar tests may be needed. When building an *ObjectMessage* it must be checked that the object is *Serializable*. When building a *StreamMessage,* it must be tested that the contents are being streamed in the expected order. These tests helps to be sure that the message is built correctly. All that is left to verify is whether each message producer passes in the correct content depending on the content of the message they intend to send. The following code presents a happy path test.

```
public class SubmitOrderTest extends TestCase {
private MockControl mapMessageSenderControl;
private MapMessageSender mapMessageSender;
private Customer fatihalgan;
private Order order;

protected void setUp() throws Exception {
	mapMessageSenderControl =
	MockControl.createControl(MapMessageSender.class);
	mapMessageSender = (MapMessageSender) mapMessageSenderControl.getMock();
	fatihalgan = new Customer("fatihalgan");
	fatihalgan.emailAddress = "fatihalgan@iyte.edu.tr";
	Set orderItems = Collections.singleton(
	new CoffeeQuantity(3, "Special Blend"));
	order = new Order(new Integer(762), fatihalgan, orderItems);
}

public void testHappyPath() throws Exception {
	Map expectedMessageContent =
	Collections.singletonMap("customer-email",
	fatihalgan.emailAddress);
	mapMessageSender.sendMapMessage("queue/Orders",
	expectedMessageContent);
	mapMessageSenderControl.setVoidCallable();
	mapMessageSenderControl.replay();
	SubmitOrderCommand command = new SubmitOrderCommand();
	command.setOrder(order);
	command.execute(mapMessageSender);
	mapMessageSenderControl.verify();
}
}
```

Here, **EasyMock** “(WEB_30, 2005)” is used to mock the *MapMessage* sender, because it already had been tested seperately. The message content (the *Map* object)is verified by examining the parameter that the *SubmitOrderCommand* passes to the *MapMessageSender.* This test also verifies the destination queue for the message using EasyMock.

5.3.4 Testing J2EE Web Components

Testing web J2EE web components can be accomplished with a number of approaches. The first approach is testing the components in a container. With this approach the tests initialize a web container and invoke servlet methods as needed. The servlets should be designed in a way that their methods are small enough and decoupled from the business logic of the application. The second approach in testing web components is simulating the web container. Rather than using a live container the tests can use a lightweight container simulation to manage the web components. Two main benefits of this approach are faster tests and more control. The tests will execute more quickly because the simulated container does not provide all the same value added functions of a production quality application server. Besides, the simulated container will provide an API to access many of the HTTP objects externally which a production container does not. These objects can be used to setup a text fixture and later to verify the results. This is very useful for example while handling a request and rendering a response. The tool used in this study as a simulated Java web container framework is ServletUnit “(WEB_31, 2005)”, which provides *ServletRunner* object as its lightweight container, capable of processing web deployment descriptors and registering servlets programmatically. The last approach is avoiding the container at all. The web components used in web applications are just Java objects, so they can certainly be tested without a container.

5.3.4.1 Testing the Session Data

Temporary client data that persists between client requests is stored in the *HttpSession* object of a servlet container. Thus, a way is needed to test if the session is updated correctly, without involving the entire web application in the process. End to end tests are typically quite long, as they can require many steps just to get the application to the desired point and they poor at locating defects. Many web applications scatter their interaction with the session all over the place, either duplicated within the servlet or in a variety of places outside the servlet. This logic needs to be refactored to be made available outside the container, yet still capable of interacting with the session. There are two key responsibilities in this interaction, updating session data and then updating the HTTP session object. A distinction between the session and its data provides the decoupling needed for testing. The business logic does not need to know where the data comes from, some might come from the request, some from the session or other place. Thus, the following refactoring can be applied to achieve the desired decoupling;

- Moving the logic that updates the session data to a seperate class, which may be called an *action*.
- Keeping the logic that updates the HTTP *session* with the *session data* in the servlet.
- At each request, making the servlet take a snapshot of the session data and pass that to the *action* for processing.

To illustrate the scenario, a coffeshop shopping cart implemented with an HTTP session is presented below. When the user submits the form to add a few kilograms of coffee to her shopcart, the following code runs;

```
HttpSession session = request.getSession(true);
for (Iterator i = requestedQuantities.iterator(); i.hasNext();) {
        CoffeeQuantity each = (CoffeeQuantity) i.next();
        Integer currentQuantityInKilograms =
                (Integer)session.getAttribute(each.getCoffeeName());
        if (currentQuantityInKilograms == null) {
                session.setAttribute(each.getCoffeeName(),
                new Integer(each.getAmountInKilograms()));
        } else {
                int newQuantityInKilograms =
                currentQuantityInKilograms.intValue()
                + each.getAmountInKilograms();
                session.setAttribute(each.getCoffeeName(),
                new Integer(newQuantityInKilograms));
        }
}
```

The code above lives inside the servlet and is invoked by the method *doPost()*. The r*equestedQuantities* is a collection of *CoffeeQuantity* objects, each of which describes the amount of a certain type of coffee. When it is time to display the customer's shopcart, the following code takes over;

```
public static ShopcartBean create(HttpSession session,
        CoffeeCatalog catalog) {
        ShopcartBean shopcartBean = new ShopcartBean();
        for (Enumeration e = session.getAttributeNames(); e.hasMoreElements();)
        {
                String eachCoffeeName = (String) e.nextElement();
                Integer eachQuantityInKilograms =
                (Integer) session.getAttribute(eachCoffeeName);
                ShopcartItemBean item = new ShopcartItemBean(eachCoffeeName,
                eachQuantityInKilograms.intValue(), catalog.getUnitPrice(eachCoffeeName));
                shopcartBean.shopcartItems.add(item);
        }
        return shopcartBean;
}
```

This code lives within the *ShopCartBean*, the object that contains all the shopcart data to be displayed on a web page. The ShopCartBean interacts directly with the http session, despite it has the potential to be used outside the context of a web application. This indicates a high coupling in the design. A pure and simple test that validates updating the shopcart should look like the following;

```
public void testAddToEmptyShopcart() {
        String coffeeProductId = "0";
        String coffeeName = "Java";
        int requestedQuantity = 5;
        CoffeeCatalog catalog = new CoffeeCatalog();
        catalog.addCoffee(coffeeProductId, coffeeName, Money.dollars(7, 50));
        ShopcartModel model = new ShopcartModel();
        List requestedQuantities = Collections.singletonList(
                new CoffeeQuantity(requestedQuantity,
```

```
		catalog.lookupCoffeeById(coffeeProductId)));
	model.addCoffeeQuantities(requestedQuantities);
	assertEquals(5, model.getQuantity("Java"));
	assertEquals(5, model.getTotalQuantity());
}
```

This test primes the catalog with data, creates a new shopcart, adds a certain quantity of coffee to the shopcart, then verifies both the amount of java coffee and the amounts of all coffees. The last assertion ensures that the java is the only coffee in the shopcart. This is much more to the point. To write this test, a design change indicated in figure 5.3 is needed. The final ShopcartModel is as following;

```
public class ShopcartModel implements Serializable {
	private Map coffeeQuantities = new HashMap();

	public void addCoffeeQuantities(List requestedQuantities) {
		for (Iterator i = requestedQuantities.iterator(); i.hasNext();) {
			CoffeeQuantity each = (CoffeeQuantity) i.next();
			String coffeeName = each.getCoffeeName();
			CoffeeQuantity currentQuantity =
			getCoffeeQuantity(coffeeName);
			Quantity sum = each.add(currentQuantity);
			coffeeQuantities.put(coffeeName, sum);
		}
	}

	private CoffeeQuantity getCoffeeQuantity(String coffeeName)
		CoffeeQuantity currentQuantity =
		(CoffeeQuantity) coffeeQuantities.get(coffeeName);
		return (currentQuantity == null) ?
		new CoffeeQuantity(0, coffeeName) : currentQuantity;
	}


	public int getQuantity(String coffeeName) {
		return getCoffeeQuantity(coffeeName).getAmountInKilograms();
	}

	public int getTotalQuantity() {
		int totalQuantity = 0;
		for (Iterator i = coffeeQuantities.values().iterator();
		i.hasNext();)
		{
			CoffeeQuantity each = (CoffeeQuantity) i.next();
			totalQuantity += each.getAmountInKilograms();
	}
	return totalQuantity;
	}

	public Iterator items() {
		return coffeeQuantities.values().iterator();
	}

	public boolean isEmpty() {
		return coffeeQuantities.isEmpty();
	}

	public boolean equals(Object other) {
		if (other != null && other instanceof ShopcartModel) {
```

```
			ShopcartModel that = (ShopcartModel) other;
			return this.coffeeQuantities.equals(that.coffeeQuantities);
		} else {
			return false;
		}
	}

	public int hashCode() {
		return coffeeQuantities.hashCode();
	}

	public String toString() {
		return "a ShopcartModel with " + coffeeQuantities;
	}
	}
```

The servlet can also be refactored so that interaction with the *HttpSession* object is reduced to a single method;

```
	public ShopcartModel getShopcartModel(HttpServletRequest request) {
		HttpSession session = request.getSession(true);
		ShopcartModel model =
			(ShopcartModel) session.getAttribute("shopcartModel");
		if (model == null) {
			model = new ShopcartModel();
			session.setAttribute("shopcartModel", model);
		}
		return model;
	}
```

This is the entire interface between the business logic and http session. It is so simple that defects in the business logic will not affect the interaction with the session. To test the interaction with the session, only the following tests are needed;

- Start with an empty session. Issue a request. Expect a shopcart model in the session.
- Start with a session containing a shopcart model. Issue a request. Expect the shopcart model to be there.

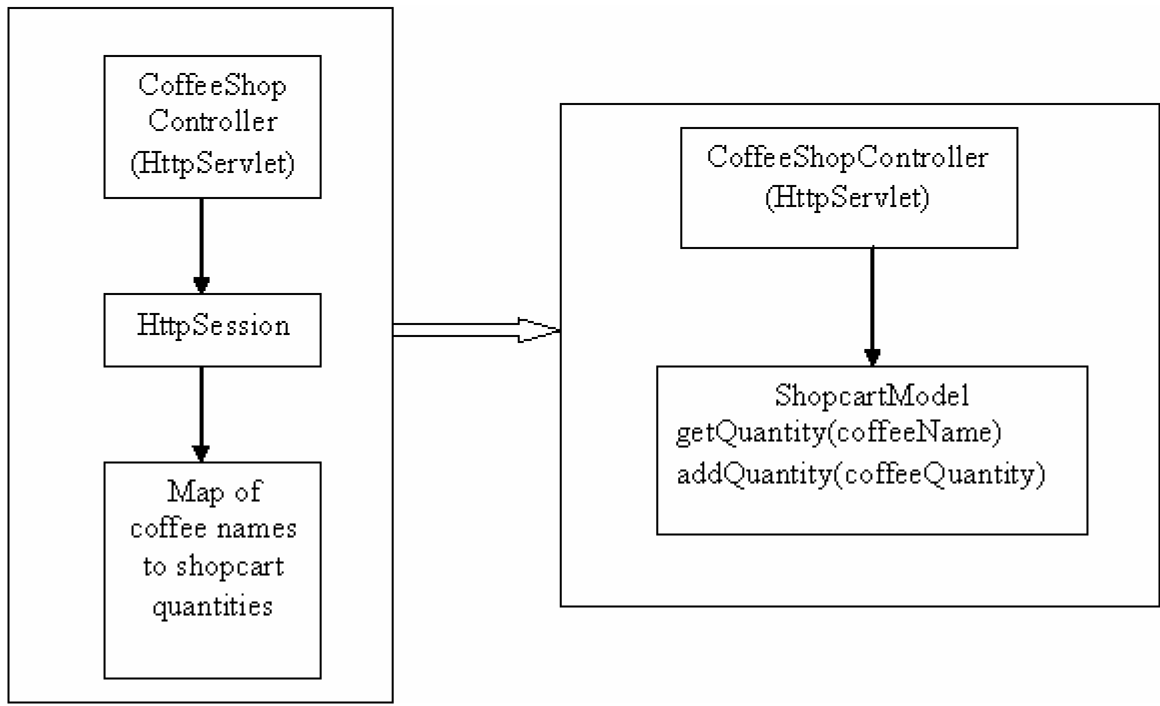

Figure 5.3 Refactoring design for tests

5.3.4.2 Testing Processing a Request

The typical way one tests a servlet is manually, and through the end-user interface. Often, the general strategy is to click through a maze of pages until arriving at the right one, fill in some text, push a button, and determine from the end result whether the servlet did the right thing. There are two principal downfalls with this approach. First, the tests are manual, so they are expensive to execute and prone to error. Second, the result is verified *indirectly* by observing side effects of correct behavior, rather than making assertions on the servlet itself. Clearly some automated tests are in order, tests that verify the correct handling of the request without relying on the correctness of the rest of the application.

In a typical web application, a servlet behaves as follows;

- Receive a request from the web container.
- Choose some business logic to execute based on the request.
- Extract data from the request and pass it to the business logic as parameters.

Each of these aspects of a servlet's behavior is to be tested. To keep the tests isolated, a few principles must be followed. The first is, "the servlet does not know where the request comes from." A request should be simulated without involving a live web container. The next is, "the business logic does not have to behave correctly." The production business logic should not be invoked corresponding to the request, but rather verify that the appropriate method is chosen and the correct parameters are sent to it. The business logic will be tested elsewhere. At this point, the servlet becomes nothing more than a data transfer bus between the network and the business objects.

If there is a legacy servlet in hand, with all the logic in the request handling methods, then the apprach would be testing it like any other method that has no return value as described in section **5.2.3.1**. For a servlet, the side effect will most likely be invoking the business logic, something which must be avoided. Therefore, the code that processes the request parameters must be refactored into a seperate method. Then the new method can be tested in isolation, in addition to verfying that the servlet correctly invokes it. Testing the former is straightforward, the latter can be accomplished with mock objects.

Returning to the imaginary shoppingcart scenario, a shopper adds a quantity of coffee to her shoppingcart. The application presents the catalog information to the user with a text field to specify the quantity of a given coffee and a buton to add it to the shopcart. When the buy buton is pressed, the servlet instantiates an *AddToShopcartCommand*, containing the *ShopcartModel* for the customer's shopcart and the *CoffeeQuantity* object that corresponds to their choice of coffee name and amount. It then executes this command to update the shopcart. Testing whether the command is created from the request without actually invoking the command is needed. To do this, the command creating logic is extracted into a new method named *makeAddToShopcartCommand().* Rather than taking the *HttpServletRequest* as a parameter, it takes as parameters the request parameters(as a *Map*) and the session attributes(as a *Map*). When this design is applied, by the time the servlet invokes *makeAddToShopcartCommand()*, there will be no servlet related interfaces to deal with.

```
public void testMakeCommandValidRequest() {
        CoffeeShopController controller = new CoffeeShopController();
        CoffeeShopModel coffeeShopModel = new CoffeeShopModel();
        coffeeShopModel.getCatalog().addCoffee("0", "Java",
        Money.dollars(7, 50));
        controller.setModel(coffeeShopModel);
        Map parameters = new HashMap();
        parameters.put("quantity-0", new String[] { "2" });
        parameters.put("addToShopcart-0", new String[] { "Buy!" });
        ShopcartModel shopcartModel = new ShopcartModel();
        Map sessionAttributes = Collections.singletonMap("shopcartModel",
        shopcartModel);
        AddToShopcartCommand actualCommand =
        controller.makeAddToShopcartCommand(parameters, sessionAttributes);
        AddToShopcartCommand expectedCommand = new AddToShopcartCommand(
                new CoffeeQuantity(2, "Java"), shopcartModel);
        assertEquals(expectedCommand, actualCommand);
}
```

The request parameter values are set as arrays of String objects beacause that is how the servlet API presents them when *HttpServletRequest.getParameterMap()* is invoked. Besides, a programmer provided coffee catalog is substituted for the one the servlet would initialize to avoid a dependency between this test and the servlet.

The next step is to verify that given a request to add coffee to the shopcart, the servlet attempts to invoke *makeAddToShopcartCommand()* with the expected parameters. To determine how to create the request, inspecting the page that presents the appropriate form will be helpful. The names of the request parameters needed depend on the *productID* of the coffee the customer wants to buy. If she chooses coffee with product id *100*, then the name of the quantity parameter will be *quantity-100*, and the name of the submit buton for adding that coffee to the shopcart is *addToShopcart-100*. The shopcart is stored in the session under the name *shopcartModel*, and that is all

the information needed to simulate a request. If a mock request and response is created, *doPost()* is invoked and then verified that whether *makeAddToShopcartCommand()* is invoked, the test will be complete. To do that, subclassing the class under test, intercepting that method invocation, asserting that it happened and that the parameters were the correct ones is enough.

The following code shows the test in question;

```
public void testServletInvokesMakeAddToShopcartCommand() throws Exception {
        final Map expectedRequestParameters = new HashMap() {
        {
                put("quantity-0", new String[] { "2" });
                put("addToShopcart-0", new String[] { "Buy!" });
        }
};

ShopcartModel shopcartModel = new ShopcartModel();
final Map expectedSessionAttributes=
        Collections.singletonMap("shopcartModel", shopcartModel);
CoffeeShopModel coffeeShopModel = new CoffeeShopModel();
coffeeShopModel.getCatalog().addCoffee("0", "Java",
        Money.dollars(7, 50));


/**
 * Fake making the command
 */
CoffeeShopController controller = new CoffeeShopController() {
        public AddToShopcartCommand makeAddToShopcartCommand(
                Map parameters, Map sessionAttributes) {
                makeAddToShopcartCommandInvoked = true;
                assertEquals(expectedRequestParameters, parameters);
                assertEquals(expectedSessionAttributes,
                        sessionAttributes);
                return null;
        }
};

controller.setModel(coffeeShopModel);

/**
 * Use EasyMock to mock the HttpServletResponse
 */
MockControl httpServletResponseControl =
        MockControl.createNiceControl(HttpServletResponse.class);

HttpServletResponse httpServletResponse =
        (HttpServletResponse) httpServletResponseControl.getMock();

/**
 * A way to fake HttpServletRequest
 */
final HttpRequestBase httpServletRequest = new HttpRequestBase() {
        public HttpSession getSession(boolean create) {
                return new FakeHttpSession(expectedSessionAttributes);
        }

        public RequestDispatcher getRequestDispatcher(String path) {
                return new RequestDispatcherAdapter();
        }
```

```
		};

		/**
		 * Copy the request parameters into the fake request
		 */
		httpServletRequest.clearParameters();
		CollectionUtil.forEachDo(expectedRequestParameters,
			new MapEntryClosure() {
				public void eachMapEntry(Object key, Object value) {
					httpServletRequest.addParameter((String) key,
						(String[]) value);
				}
			}
		);

		controller.doPost(httpServletRequest, httpServletResponse);

		/**
		 * Check if the method is invoked.
		 */
		assertTrue("Did not invoke makeAddToShopcartCommand()",
			makeAddToShopcartCommandInvoked);

}
```

There are two key parts to this test; intercepting the invocation of *makeAddToShopCartCommand()* and asserting that the method was invoked. A field in the test case class named *makeAddToShopcartCommandInvoked* is created to store whether the method was invoked, initialized to *false* in the *setUp()*, set to *true* when the method is invoked and then verified at the end of the test. When the test invokes *doPost()* on the servlet, the spy version of *makeAddToShopcartCommand()* says someone invoked me and makes assertions about the parameters passed into it. These tests verifies that the servlet correctly extracts the relevant data from the request, and chooses the correct business logic to execute. Now, it must be verified that it actually invokes the command. The same technique used in makeAddToShopcartCommand() can be used here as well; extract the method executeCommand(), intercept the method invocation, assert that it happened, and that the correct command was executed.

```
	public void testServletInvokesExecuteCommand() throws Exception {

		final AddToShopcartCommand expectedCommand = new AddToShopcartCommand(
		new CoffeeQuantity(200, "Special Blend"), new ShopcartModel());

		CoffeeShopController controller = new CoffeeShopController() {
		/**
		 * Fake making the command
		 */
		public AddToShopcartCommand makeAddToShopcartCommand(
			HttpServletRequest request) {
			return expectedCommand;
		}

		/**
		 * Record the invocation of executeCommand
		 */
		public void executeCommand(AddToShopcartCommand command) {
			executeCommandInvoked = true;
			assertEquals(expectedCommand, command);
```

```
		}
		};

		final HttpRequestBase httpServletRequest = new HttpRequestBase() {
		public HttpSession getSession(boolean create) {
			return new FakeHttpSession(Collections.EMPTY_MAP);
		}
		public RequestDispatcher getRequestDispatcher(String path) {
			return new RequestDispatcherAdapter();
		}
		};

		httpServletRequest.clearParameters();
		Map requestParameters = new HashMap() {
		{
			put("quantity-0", new String[] { "2" });
			put("addToShopcart-0", new String[] { "Buy!" });
		}
		};

		CollectionUtil.forEachDo(requestParameters, new MapEntryClosure() {
			public void eachMapEntry(Object key, Object value) {
				httpServletRequest.addParameter((String) key,
				(String[]) value);
			}
		});

		MockControl httpServletResponseControl =
			MockControl.createNiceControl(HttpServletResponse.class);
		HttpServletResponse httpServletResponse =
			(HttpServletResponse) httpServletResponseControl.getMock();

		/**
		 * The actual test
		*/
		controller.doPost(httpServletRequest, httpServletResponse);
		assertTrue("Did not invoke executeCommand()", executeCommandInvoked);

	}
```

The test consists of three basic actions;

- **Fake making the command:** Fake *makeAddToShopcartCommand()* to return a hard coded command. This avoids worrying about whether the servlet does this correctly.
- **Record The Invocation Of *executeCommand()*:** Intercept the executeCommand so that the test knows whether the servlet invoked it. This information is stored in the field *executeCommandInvoked*, which is declared on the test case class.
- **The actual test:** Invoke *doPost()* and verify that the servlet invoked *executeCommand()*.

Testing a servlet this way leads to intercepting a number of the servlet's methods, substituting some test-only behavior in their place. Repeatedly subclassing the class under test indicates that it is time to consider moving that behavior to a separate class.

5.3.4.3 Testing Servlet Initialization

The trouble verifying a servlet's initialization behavior, is most likely because the behavior has no directly observable side effect. Often the servlet reads some startup data or builds some internal lookup tables and that information is not available outside the servlet object, instead they are *private* parts. Other times, servlet initialization consists of initializing other resources, such as a data source, messaging server, or security server. It is often difficult to verify that the servlet initializes those resources correctly, as doing so involves both having those resources online and making assertions on their state. The most direct approach to the problem is invoking *init()* and verifying the results.

```
public class InitializeCoffeeShopControllerTest extends TestCase {
	public void testCatalogInitialized() throws Exception {
		//Hard coded initialization parameters
		final Map initParameters = new HashMap();
		initParameters.put("A", "$7.25");

		CoffeeShopController controller = new CoffeeShopController() {
			public Enumeration getInitParameterNames() {
				return new Vector(initParameters.keySet()).elements();
			}

			//Substitute hard coded data
			public String getInitParameter(String name) {
				return (String) initParameters.get(name);
			}
		};
		controller.init();
		//Verify servlet processed hard coded data
		assertEquals(1, controller.getCatalog().size());
		}
	}
```

To invoke *init()*, a *ServletConfig* object to this method must be passed, in which case a simple implementation of this interface is needed, possibly an offtheshelf mock. An easy to use implementation of *ServletConfig* can be found in the reference implementation of J2EE, called *org.apache.catalina.core.StandardWrapper*. With this class, initialization parameters can be set by invoking *addInitializationParameter(key, value)*, then it can be verified that servlet processes those parameters correctly.

```
public void testProcessedParameters() throws Exception {
	FrontControllerServlet servlet = new FrontControllerServlet() {
		public void log(String message) {
			// Intentionally disable logging
		}
	};
	StandardWrapper config = new StandardWrapper();
	config.addInitParameter("serverImplementationClass",
	"myclasses.servlet.RmiServerImpl");
	servlet.init(config);
	assertEquals("myclasses.servlet.RmiServerImpl",
		servlet.getServerImplClassName());
}
```

To summarize, if the servlet implements *init()* with no parameters then the servlet methods that retrieve initialization parameters must be overriden. This is the most direct way to control the initialization parameters the servlet tries to process during a test. If the servlet implements *init(ServletConfig)* then the *StandardWrapper* can be used to hard code initialization parameters. This second option is generally less confusing, so it might be a beter choice implementing *init(ServletConfig)* in servlets, rather than the no-parameter version.

5.3.4.3 Testing the ServletContext

It is common to store lookup tables in the ServletContext. In applications with multiple servlets that look up data at initialization, it is common to cache this data in the ServletContext, as this object is available to all servlets in a web application. Certainly if the cache is incorrect on startup, the application does not stand much of a chance of working. There are two different behaviors here, the servlet asks an object to load the data, and that indeed the object loads the data correctly. Testing them seperately will provide more flexibility and isolation.

Testing the ability to load the lookup data depends on how it is stored. If the data is stored on disk, then simply loading it into memory, presenting the data as objects and then verifying that the right objects are loaded is enough. In the end a “lookup data provider” with methods to lookup data by some key will emerge in most cases. To find the unit price for a given coffee in the CoffeeShop application an interface is provided as the following;

```
public interface CoffeeBeanUnitPriceProvider {
        Money getUnitPrice(String coffeeName);
}
```

For the second part of the problem, verifying that the servlet asks to load that data and stores it in the *ServletContext*, a possible solution can be using **ServletUnit** “(WEB_31, 2005)” and servlet context initialization parameters. In the web deployment descriptor of the web application, it can be specified which implementation of the *CoffeeBeanUnitPriceProvider* to instantiate and place in the *ServletContext*. The application can then obtain this object and ask it for unit prices whenever needed.

```
public class InitializeUnitPricesTest {
        private ServletRunner servletRunner;
        private ServletUnitClient client;
        public void testInitializeUnitPrices() throws Exception {
                servletRunner = new ServletRunner(
                        getWebContentPath("/WEB-INF/web.xml"), "/coffeeShop");
                client = servletRunner.newClient();
                WebRequest request = new PostMethodWebRequest(
                        "http://localhost/coffeeShop/coffee");
                request.setParameter("browseCatalog", "catalog");
                client.sendRequest(request);
                InvocationContext invocationContext=client.newInvocation(request);
                CoffeeShopController controller =
                        (CoffeeShopController) invocationContext.getServlet();
                assertTrue(controller.getServletContext().getAttribute(
                        "unitPriceProvider")
                instanceof CoffeeBeanUnitPriceProviderFileImpl);
```

```
		}
	}
```

The key parts to this test are triggering servlet initialization and verifying what kind of unit price provider ends up in the servlet context. When this test passes, it is verified that a unit price provider is available on servlet startup. By testing the implementation of CoffeeBeanUnitPriceProvider interface the tests will be complete.

5.3.4.4 Testing Rendering a JavaServer Page

JavaServer Pages(JSP) is a J2EE view technology that produces dynamic web content during runtime. Since the content of the JSP page is generated by the JSP compiler of the application server platform, writing end to end tests that runs within the container platform might look as a first choice. However, another option might be hard coding some data for the JSP, then rendering it directly using a JSP engine. The JSP engine's output can be compared with a verison of the JSP output that have been checked by hand once and then filed away as correct.

The JSP in question can be rendered using **ServletUnit** "(WEB_31, 2005)" along with **Jasper**. ServletUnit also allows to intercept the request on the way to the JSP so that additional data can be added into it in the form of request or session attributes. Finally, the current JSP output may be compared to the known, correct output.

```
public class RenderShopcartJspTest extends JspTestCase {
	private ShopcartBean shopcartBean;
	private ServletRunner servletRunner;
	private ServletUnitClient client;

	protected void setUp() throws Exception {
		shopcartBean = new ShopcartBean();

		//Register an entire web deployment descriptor.
		servletRunner = new ServletRunner(getWebContentPath("/WEB-
			INF/web.xml"), "/coffeeShop");

		//A dummy dervlet to help serve up JSPs
		servletRunner.registerServlet("/forward",
			ForwardingServlet.class.getName());
		client = servletRunner.newClient();
	}

	public void testEmptyShopcart() throws Exception {

		//Check against the known correct JSP
		checkShopcartPageAgainst(new File("test/gold",
			"emptyShopcart-master.txt"));
	}

	public void testOneItemInShopcart() throws Exception {
		shopcartBean.shopcartItems.add(new ShopcartItemBean(
		"Java", "762", 5, Money.dollars(7, 50)));
		checkShopcartPageAgainst(new File("test/gold",
			"oneItemInShopcart-master.txt"));
	}
```

```
// Helper code omitted for now
```

The superclass *JSPTestCase* provides some useful methods for locating JSPs on the file system and deciding where on the file system the JSP engine should generate servlet source code. The *ForwardingServlet* is a simple servlet that does two things. First, it lets a test put data into the request or session. Second, it forwards the request to the URI specified. This simulates what CoffeeShopController does in production after all the business logic and database updates are complete.

```
public class ForwardingServlet extends HttpServlet {
    private String forwardUri = "";

    protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException{
        handleRequest(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        handleRequest(request, response);
    }

    protected void handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        getServletContext().getRequestDispatcher(getForwardUri()).forward(
        request, response);
    }

    public void setForwardUri(String forwardUri) {
        this.forwardUri = forwardUri;
    }

    public String getForwardUri() {
        return forwardUri;
    }
}
```

The next part is rendering the JSP and retrieving its content. This is accomplished with using the *ForwardingServlet* in combination with *ServletUnit*, then reading the JSP output as text.

```
public String getActualShopcartPageContent() throws Exception {
    //Create the ServletUnitClient in setUp()
    InvocationContext invocationContext=client.newInvocation(
    "http://localhost/coffeeShop/forward");
    ForwardingServlet servlet=
        (ForwardingServlet)invocationContext.getServlet();
    servlet.setForwardUri("/shopcart.jsp");
    HttpServletRequest request=invocationContext.getRequest();

    //Put the ShopCart data on the request
    request.setAttribute("shopcartDisplay", shopcartBean);

    //Invoke the ForwardingServlet
    servlet.service(request, invocationContext.getResponse());

    //Get the JSP output as text
```

```
        return invocationContext.getServletResponse().getText();
}
```

The steps are straightforward. Put data in the request, render the JSP, and look at the resulting web page.

The last piece of the puzzle comes in two parts. The technique requires first creating the expected page and checking it by visual inspection. Second, future output must be verified against the one created. The file created by hand for comparison will be called as **"Master".** To create the master, the JSP text must be written out to a file;

```
public void generateMaster(File masterFile) throws Exception {
        String responseText = getActualShopcartPageContent();
        new MasterFile(masterFile).write(responseText);
        fail("Writing Master file.");
}
```

When the test is first coded, it invokes the *generateMaster().* This method creates the master file and **fails the test as a reminder to indicate that things have not finished yet**. So the test will look like this the first time it is executed;

```
public void testEmptyShopcart() throws Exception {
        generateMaster(new File("test/master", "emptyShopcart-master.txt"));
}
```

The output of the master file is going to be similar to the following;

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<meta name="GENERATOR" content="IBM WebSphere Studio" />
<meta http-equiv="Content-Style-Type" content="text/css" />
<link href="theme/Master.css" rel="stylesheet" type="text/css" />
<title>shopcart.jsp</title>
</head>
<body>
<h1>your shopcart contains</h1>
<table name="shopcart" border="1">
<tbody>
<tr>
<td><b>Name</b></td>
<td><b>Quantity</b></td>
<td><b>Unit Price</b></td>
<td><b>Total Price</b></td>
</tr>
<tr>
<td colspan="3"><b>Subtotal</b></td>
<td><b>$0.00</b></td>
</tr>
</tbody>
</table>
<form action="coffee" method="POST"><input type="submit"
        name="browseCatalog" value="Buy More Coffee!" /></form>
</body>
</html>
```

Now that the master file is written to the disk, the test needs to be changed so that it checks the response from the server against the master;

```
public void testEmptyShopcart() throws Exception {
	checkShopcartPageAgainst(new File("test/master",
		"emptyShopcart-master.txt"));
}
```

The checkShopcartPageAgainst() will look like the following,

```
public void checkShopcartPageAgainst(File goldMasterFile) throws Exception {
	String responseText = getActualShopcartPageContent();
	new GoldMasterFile(goldMasterFile).check(responseText);
}
```

The class *MasterFile* provides some convenience methods for generating and checking against a **master file**.

```
public class MasterFile extends Assert {
	private File file;
	public MasterFile(String directory, String file) {
		this(new File(directory, file));
	}

	public MasterFile(File file) {
		this.file = file;
	}

	public void write(String content) throws IOException {
		file.getParentFile().mkdirs();
		FileWriter masterWriter = new FileWriter(file);
		masterWriter.write(content);
		masterWriter.close();
	}

	public void check(String actualContent) throws IOException {
		assertTrue("Master [" + file.getAbsolutePath() + "] +
			not found.", file.exists());
		StringWriter stringWriter = new StringWriter();
		PrintWriter printWriter = new PrintWriter(stringWriter);
		BufferedReader masterReader =
			new BufferedReader(new FileReader(file));
		while (true) {
			String line = masterReader.readLine();
			if (line == null) break;
			printWriter.println(line);
		}
		assertEquals(stringWriter.toString(), actualContent);
	}
}
```

Lastly, following is the code for the *JSPTestCase*;

```
public abstract class JspTestCase extends TestCase {
	protected static final String webApplicationRoot=
		"../CoffeeShopWeb/Web Content"; | #1

	protected String getWebContentPath(String relativePath) {
```

```
			return new File(webApplicationRoot, relativePath).getAbsolutePath();
		}

		protected String getCoffeeShopUrlString(String uri)
			throws Exception {
			return "http://localhost/coffeeShop" + uri;
		}

		protected String getJspTempDirectory() {
			return System.getProperty("java.io.tmpdir");
		}

		protected void tearDown() throws Exception {
			File[] files = new File(getJspTempDirectory()).listFiles(
			new FilenameFilter() {
				public boolean accept(File dir, String name) {
				return name.endsWith(".java")
				|| name.endsWith(".class");
			}
			});
			for (int i = 0; i < files.length; i++) {
				File file = files[i];
				file.delete();
			}
		}
	}
```

The technique allows for full testing of JSP pages, both the data and the layout. However, the technique is not very practical to apply and prone to errors since the master file needs to be deleted in each change. Besides, the tests will fail with every cosmetic change to the user interface and all the process needs to be reapplied again. Therefore, testing only the data of the JSP using the technique presented in sections 5.3.1.1 or 5.3.1.2 using XMLUnit "(WEB_26, 2005)" would be the recommended solution in this study.

5.3.4.5 Testing Custom JSP Tags

In order to test a JSP tag handler in isolation, a test that invoke the tag handler's methods with the same order as a JSP engine would is needed. For an example, a custom tag that iterates over the items in a shopcart, presenting each shopcart item as a JavaBean is given.

```
<table name="shopcart" border="1">
<tbody>
<tr>
<td><b>Name</b></td>
<td><b>Quantity</b></td>
<td><b>Unit Price</b></td>
<td><b>Total Price</b></td>
</tr>
<coffee:eachShopcartItem shopcartBean="shopcartDisplay" each="item">
	<tr>
	<td><%= item.coffeeName %></td>
	<td id="product-<%= item.productId %>">
	<%= item.quantityInKilograms %> kg
```

```
		</td>
		<td><%= item.unitPrice %></td>
		<td><%= item.getTotalPrice() %></td>
		</tr>
</coffee:eachShopcartItem>
<tr>
<td colspan="3"><b>Subtotal</b></td>
<td>
<b><jsp:getProperty name="shopcartDisplay" property="subtotal" /></b>
</td>
</tr>
</tbody>
</table>
```

The tag <*coffee:eachShopcartItem*> defines an iterator over the shopcart items, placing each in the scripting variable named by the attribute *each*. This is an *IterationTag,* so referring to the lifecycle for an *IterationTag*, a pseudocode is given below;

```
initialize page context
initialize tag attributes
whatNext := doStartTag()
if (whatNext == EVAL_BODY_INCLUDE)
		do
				evaluate body
		until (doAfterBody() == SKIP_BODY)
endif
whatNext := doEndTag()
if (whatNext == EVAL_PAGE)
		evaluate rest of page
endif
```

The tag takes the specified shopcart and iterates over the items in it. The tag stores each item in a scripting variable, implemented as a page context attribute, so that the JSP can display its properties. The behavior to be tested can be explained in terms of the way that the JSP engine is expected to invoke the tag handler;

1. Set the tag attribute values, shopcartBean and each.
2. Invoke *doStartTag()*. If the shopcart is empty, the tag body should be skipped; otherwise, the first shopcart item should be stored in the scripting variable named by the attribute *each*, and then process the body.
3. If the shopcart is not empty, invoke *doAfterBody()*. If there are more shopcart items, store the next shopcart item in the scripting variable named by the attribute *each*, then process the body again; otherwise, skip the body.
4. Now that all the shopcart items have been processed, invoke *doEndTag()*, then evaluate the rest of the page.

This is essentially a JSP engine simulation, a theorotical one that follows the JSP specification correctly. These steps can be translated into code like the following;

```
public class EachShopcartItemHandlerTest extends TestCase {
		private EachShopcartItemHandler handler;
		private ShopcartBean shopcartBean;
		private MockPageContext pageContext;

		protected void setUp() throws Exception {
```

```
        shopcartBean = new ShopcartBean();
        handler = new EachShopcartItemHandler();

        pageContext = new MockPageContext() {
            private Map attributes = new HashMap();
            public Object getAttribute(String name) {
                return attributes.get(name);
            }
            public void setAttribute(String name, Object value) {
                attributes.put(name, value);
            }
            public void removeAttribute(String name) {
                attributes.remove(name);
            }
        };

        MockJspWriter out = new MockJspWriter();
        pageContext.setJspWriter(out);
        handler.setPageContext(pageContext);
        handler.setParent(null);
        handler.setShopcartBean(shopcartBean);
        handler.setEach("item");
    }

    public void testEmptyShopcart() throws Exception {
        assertEquals(Tag.SKIP_BODY, handler.doStartTag());
        assertNull(getTheEachAttribute());
        assertEquals(Tag.EVAL_PAGE, handler.doEndTag());
    }

    public void testOneItem() throws Exception {
        ShopcartItemBean shopcartItem1 = new ShopcartItemBean(
            "Java", "762", 1, Money.dollars(10, 0));
        shopcartBean.shopcartItems.add(shopcartItem1);

        //Allows to refer to each shopcart item by index.
        List shopcartItemAsList = new LinkedList(
            shopcartBean.shopcartItems);
        assertEquals(Tag.EVAL_BODY_INCLUDE, handler.doStartTag());
        assertEquals(shopcartItemAsList.get(0), getTheEachAttribute());
        assertEquals(Tag.SKIP_BODY, handler.doAfterBody());
        assertNull(getTheEachAttribute());
        assertEquals(Tag.EVAL_PAGE, handler.doEndTag());
    }

    public void testTwoItems() throws Exception {
        shopcartBean.shopcartItems.add(new ShopcartItemBean(
            "Java", "762", 1, Money.dollars(10, 0)));
        shopcartBean.shopcartItems.add(new ShopcartItemBean(
            "Special Blend", "768", 1, Money.dollars(10, 0)));
        List shopcartItemAsList = new LinkedList(
            shopcartBean.shopcartItems);
        assertEquals(Tag.EVAL_BODY_INCLUDE, handler.doStartTag());
        assertEquals(shopcartItemAsList.get(0), getTheEachAttribute());
        assertEquals(IterationTag.EVAL_BODY_AGAIN,
            handler.doAfterBody());
        assertEquals(shopcartItemAsList.get(1), getTheEachAttribute());
        assertEquals(Tag.SKIP_BODY, handler.doAfterBody());
        assertNull(getTheEachAttribute());
        assertEquals(Tag.EVAL_PAGE, handler.doEndTag());
```

```
        }

        public Object getTheEachAttribute() {
                return pageContext.getAttribute("item");
        }
}
```

This test shows iterating over an empty shopcart, a single item shopcart, and a multiple item shopcart. For a tag which writes directly to the JSP using the *JspWriter*, these lines of code should be added to the end of the test;

```
MockJspWriter out = new MockJspWriter();
pageContext.setJspWriter(out);
handler.setPageContext(pageContext);

// The rest of the test
out.setExpectedData("The output you expect");
out.verify();
```

When the test is executed, the *MockJspWriter* verifies its actual output against the expected data specified here.

Although the technique used here seems simple for this particular case, using this technique might go as far as writing a platform simulator from scratch, which is too much effort and error prone. Writing a servlet processing engine for the sake of writing tests like this would oughtweigh the benefit of testing at all. Another option might be writing an end to end test involving a JSP that uses this tag.

5.3.5 Integration Testing

The approach to testing an application in this study is testing its components thoroughly and then integrating those components as simply as possible. Integration in this context is little more than choosing which implementation of various interfaces to use, and then creating an application entry point object with references to those implementations. There are certain aspects of J2EE applications that people associate with end to end tests rather than object tests, such as page flow, using the container services such as security or transactions. The approach used in this study will be based on testing these behaviors in isolation as object tests because this approach disciplines the programming activity and leads to a beter design. However, this techniques does not fully replace end to end testing.

5.3.5.1 Testing Page Flow

Although end to end tests are also widely used to uncover problems with page flow, the Agile community generally prefers verifying page flow without involving business logic, presentation layer, external resources and so on. A web application can be translated into a large state machine where moving from page to page depends on two thisngs; the action the user took and the result of the action. The idea is to extract

the page to page navigation code from the system into a single navigation engine that operates on navigation data. The popular Struts Web Application Framework is based on such a principle. By seperating the navigation logic from other concerns of the application simplifies the navigation test like the one seen below;

```
public void testNavigateToCatalog() {
        assertEquals("Catalog Page", navigationEngine.getNextLocation(
                "Browse Catalog", "OK"));
}
```

This test simply says “if the buton named *Browse Catalog* is pressed and everything goes as expected, then the user will be taken to the C*atalog Page*.” A location corresponds to the URI of a web page or a web page template (Velocity template or JSP). An action corresponds to the URI of a form submit buton or of a hypertext link. This means that some way is needed to translate incoming request URIs into locations and actions, and vice versa. Once each URI is given the name of either a location or an action, the details of which JSP displays the catalog page or which request parameter indicates “add a product to the shopcart” can be ignored.

```
private void handleRequest(HttpServletRequest request, HttpServletResponse
        response) throws IOException, ServletException {

        String forwardUri = "index.html";
        String userName = "fatih";
        try {
                String actionName = actionMapper.getActionName(request);
                log("Performing action: " + actionName);
                if ("Browse Catalog".equals(actionName)) {
                        CoffeeCatalog catalog = model.getCatalog();
                        CatalogView view = new CatalogView(request);
                        view.setCatalog(catalog);
                        forwardUri = view.getUri();
                } else if ("Add to Shopcart".equals(actionName)) {
                        AddToShopcartCommand command =
                        makeAddToShopcartCommand(request);
                        executeCommand(command);
                } else {
                        log("I don't understand action " + actionName);
                }
        } catch (Exception wrapped) {
                throw new ServletException(wrapped);
        }
        request.getRequestDispatcher(forwardUri).forward(request, response);
}
```

Here, the *actionMapper* indicated in bold resolves the action name and determines the action to be performed inspecting the request URI.

```
public class MapRequestToActionTest extends TestCase {
        private HttpServletRequestToActionMapper actionMapper;
        protected void setUp() throws Exception {
                actionMapper = new HttpServletRequestToActionMapper();
        }

        public void testBrowseCatalogAction() throws Exception {
                Map parameters = Collections.singletonMap("browseCatalog",
                        new String[] { "catalog" });
```

```
		doTestMapAction("Browse Catalog", "/coffeeShop/coffee",
			parameters);
	}

	public void testAddToShopcart() throws Exception {
		HashMap parameters = new HashMap() {
		{
			put("addToShopcart-18", new String[] { "Buy!" });
			put("quantity-18", new String[] { "5" });
		}
		};

		doTestMapAction("Add to Shopcart", "/coffeeShop/coffee",
			parameters);
	}

	private void doTestMapAction(String expectedActionName, String uri,
		Map parameters) {
			HttpServletRequest request =
			HttpUtil.makeRequestIgnoreSession(uri, parameters);

		assertEquals(expectedActionName,
			actionMapper.getActionName(request));
	}
}
```

These tests create a fake HttpServletRequest. As the method name implies (*makeRequestIgnoreSession()*) a request is created without worrying about keeping session information, as the session information is not needed in these tests. The request to action mapper is simple, turn a request into the name of an action.

A location to URI mapper can be built in a similar style. It turns location names into URI's. After adding that, the servlet's request handler method will look like the following;

```
private void handleRequest(HttpServletRequest request, HttpServletResponse
	response) throws IOException, ServletException {

	String userName = "fatih";
	String nextLocationName = "Welcome";
	try {
		String actionName = actionMapper.getActionName(request);
		log("Performing action: " + actionName);
		if (knownActions.contains(actionName) == false) {
			log("I don't understand action " + actionName);
		} else {
			String actionResult = "OK";
			if ("Browse Catalog".equals(actionName)) {
				CoffeeCatalog catalog = model.getCatalog();
				CatalogView view = new CatalogView(request);
				view.setCatalog(catalog);
			} else if ("Add to Shopcart".equals(actionName)) {
				AddToShopcartCommand command =
				makeAddToShopcartCommand(request);
				executeCommand(command);
			}
			nextLocationName =
			navigationEngine.getNextLocation(actionName, "OK");
		}
	} catch (Exception wrapped) {
```

```
                throw new ServletException(wrapped);
        }
        String forwardUri = locationMapper.getUri(nextLocationName);
        request.getRequestDispatcher(forwardUri).forward(request, response);
}
```

The overall behavior of the request handler is straightforward;

1. Interpret the incoming request to determine which action the user wants to perform.
2. Perform the action, assuming it goes "OK." If the action fails for some reason, set the value of *result* to some short description of the failure. For example, if the user tries to add -1 kg of Java to his shopcart, set result to *invalid quantity*.
3. Ask the navigator for the next location, based on the action performed and the result of that action.
4. Determine the URI that corresponds to the next location. Provide that URI to the request dispatcher.

5.3.5.2 Testing Navigation in a Struts Application

Struts is a web application framework that is very well known and widely used by the Java community. This section is devoted to verifying the navigation rules in a Struts application. The most direct approach that can be used is to verify the content of ***struts-config.xml*** using XMLUnit "(WEB_26, 2005)". The details of testing XML with XMLUnit can be found in section 5.3.1.1.

```
public class StrutsNavigationTest extends StrutsConfigFixture {
        private static Document strutsConfigDocument;

        public static Test suite() {
                TestSetup setup =
                        new TestSetup(new TestSuite(StrutsNavigationTest.class)) {
                        private String strutsConfigFilename =
                                "test/data/sample-struts-config.xml";
                        protected void setUp() throws Exception {
                                XMLUnit.setIgnoreWhitespace(true);
                                strutsConfigDocument =
                                        XMLUnit.buildTestDocument(
                                new InputSource(new FileReader(
                                new File(strutsConfigFilename))));
                        }
                };
                return setup;
        }

        public void testLogonSubmitActionExists() throws Exception {
                assertXpathExists(getActionXpath("/LogonSubmit"),
                        strutsConfigDocument);
        }

        public void testLogonSubmitActionSuccessMappingExists()throws Exception{
                assertXpathExists(getActionForwardXpath("/LogonSubmit"),
                        strutsConfigDocument);
        }
```

```
		public void testLogonSubmitActionSuccessMapsToWelcome()throws Exception{
			assertXpathEvaluatesTo("/Welcome.do",
			getActionForwardPathXpath("/LogonSubmit", "success"),
				strutsConfigDocument);
		}
	}
```

The Struts configuration file combines navigation rules with the mapping between locations and URIs. When an action forwards to another action, it uses a navigation rule, whereas actions that forward to page templates (a JSP or a Velocity template) are location/URI mapping rules. In this way, the Struts configuration file plays the role of navigation engine as well as location mapper as described in the former section (5.3.5.1).

There are a few things to notice about. First, the struts configuration file is loaded using one time setup. Next, the methods *getActionXpath()*, *getActionForwardXpath()*, and *getActionForwardPathXpath()* translate the concepts of “action” and “action forward” to the corresponding XPath locations in *struts-config.xml*. Not only it is not needed to remember the various XPath expressions for actions and action forwards, but also duplicating those expressions must be avoided in case of future changes in the Struts Configuration file DTD.

```
	public abstract class StrutsConfigFixture extends XMLTestCase {
		protected String getActionForwardPathXpath(String action,
			String forward) {
			return getActionXpath(action) + "/forward[@name='" + forward +
				"']/@path";
		}

		protected String getActionXpath(String path) {
			return "/struts-config/action-mappings/action[@path='"
			+ path + "']";
		}

		protected String getActionForwardXpath(String action) {
			return getActionXpath(action) + "/forward";
		}
	}
```

5.3.5.3 Testing Web Security

J2EE platform provides a role based built in security mechanism for protecting server side resources. Since the main purpose of this study is based on unit testing, test isolation is one of the main preferences while designing the tests. The security settings is declarative in J2EE and is defined in the web application configuration file, *web.xml*. Thus, the approach to testing web resource security is straightforward; testing the configuration file with a tool like XMLUnit “(WEB_26, 2005)” using techniques descirbed in **section 5.3.1.1**. In the case of end to end tests, HtmlUnit provides a *CredentialProvider* API to simulate having a particular user logged in. The class *SimpleCredentialProvider* allows to specify the user name and password to simulate for all requests originating from the same *WebClient* object.

```
	webClient.setCredentialProvider(new SimpleCredentialProvider("admin", "adm1n"));
	Page page = webClient.getPage(
```

```
new URL("http://localhost:8080/coffeeShop/admin/"));
assertEquals(HttpServletResponse.SC_OK, page.getWebResponse().getStatusCode());
```

Sometimes, security requirements demand a finer grained check in which a role based model will not be enough. In this case the security mechanism must colloborate with application domain logic in order to perform the desired authorization checks. In this case a mock object approach may be taken to provide the credential information of the current user to the application logic.

5.4 Summary

Unit testing is the core practice of test driven development. The aim of unit testing is not only program code verification. The granularity of unit tests should be small and they should be run in isolation. This makes locating the errors easier. In order to be able to run unit tests in isolation, responsibilities of a small module must be identified and refactored into their own classes. Besides, the fragments that satisfy each responsibility must be designed in such a way that different implementations of the same function can be plugged in. Sometimes this can be a mock object, or a simple stub. This results in a better designed and clean code in which responsibilities are decoupled from each other. The set of best practices and methods discussed in this section are based on this principle. However, achieving test isolation may sometimes be too expensive, and might outweight the benefit of unit testing.

CHAPTER 6

IMPLEMENTATION OF TEST DRIVEN DEVELOPMENT

This section of the study aims putting some of the techniques presented in the study into practice in a comprehensive application and experiencing the whole process of test driven development practice in a real world scenario. The motive behind this challenge is achieving a more concrete understanding on how test driven development can be applied and experiencing its effects on a software project. Different design decisions taken and observations gained will be discussed on the way.

6.1 Requirements of the Ticket Seller Application

The example application is developed for the refectory of a university. Both students and the university staff are expected to have their lunch at the dining hall of the university. In order to have a meal, one must buy a ticket-meal beforehand. The main requirement is to provide all people a convenient way to buy their meal tickets. Furthermore, it is believed that some statistics on the rise and fall of the demand to the refectory at specific intervals would provide the management of the university illuminative information of how satisfied are people with the service provided to them.

6.1.1 Functional Requirements of Ticket Seller:

The functional requirements of Ticket Seller is based around three main use cases. The first and obvious use case is selling tickets. However, the management is also willing to track how many of the sold tickets are really used. Furthermore, to detect any attempt of forgery, it is requested that each consumed ticket must be tracked with a feedback mechanism back into the system, in order to make sure that no ticket is copied and used more than once. Lastly, the management has requested that the feedback mechanism must be further approved by an authority on a daily basis.

The functional requirements in this study is collected in user stories, as in Extreme Programming. Since test driven development is closely related to agile software development processes, it is a convenience to follow up the techniques used in an agile process. The user stories given below has some trivial details which are not explained and is not considered as significant for the purpose of this study, such as differentiating ticket pricing policies according to some status and location criteria of the staff and students, being able to prove that a ticket is sold to a specific person, etc. The non functional requirements as well as some functional requirements will be skipped and not covered for the sake of simplicity.

6.1.1.1 User Story 1: Ticket Sale

A student or staff of the university applies to a clerk located at each faculty to buy a meal ticket. The person has to present his identity to the clerk (either his student number or personel registration number). The clerk enters the presented number and verifies that it is correct by the name that is already registered in the computer system. The person(buyer) then tells for which month he wants to buy the tickets and how many tickets he wants. After feeding that information to the system the clerk is presented with a list of ticket numbers and the total price of the tickets. After the payment, the clerk prints the tickets and hands out to the buyer.

6.1.1.2 User Story 2: Ticket Feedback

On receiving his meal, a person has to deliver a ticket printed for the current month. The used tickets are collected on a daily basis and entered to the system as consumed tickets by an officer. The officer first selects the month (probably the current month) the tickets are printed for. A list of the tickets which were sold for the selected month but have not been used yet is presented to the officer. It is expected that the registration number of the ticket in the officer’s hand is found in that list. If it has not, that means the ticket is a copied or a forged ticket that was either used before or never existed. The ticket numbers that are checked once do not appear again and are considered as used.

6.1.1.3 User Story 3: Feedback Approval

Another officer is responsible to track if the officer who is expected to enter the feedback information to the system has done his job. So she is expected to get a print out of which tickets are checked as consumed tickets on a daily basis. She enters the date for which she wants to a list of consumed tickets and a report of ticket numbers, owner of the ticket, the date on which the ticket was sold and the price of the ticket is presented to the officer.

6.2 Preliminary Design of Ticket Seller

Ticket seller will be implemented as a J2EE Web Application in order to provide easy access from the faculties and from within different locations of the university campus. Inspecting the requirements, ticket seller can be considered a small application and the initial design is comprised of three main architectural layers as seen in **Figure 6.1**.

The web layer is built on top of **Struts Framework** which is a widely used MVC web framework for J2EE applications. The business layer is comprised of simple Command objects. Considering the size and the complexity of the functional scenarious, it would be appropriate to implement the business layer as simple Command objects,

encapsulating simple business logic in Transaction Scripts. The data is passed between the layers up and down as simple data and utility classes. The lack of a domain layer eliminates the need to use and O/R mapping framework in the persistence layer, so the data access layer will be using pure JDBC underneath.

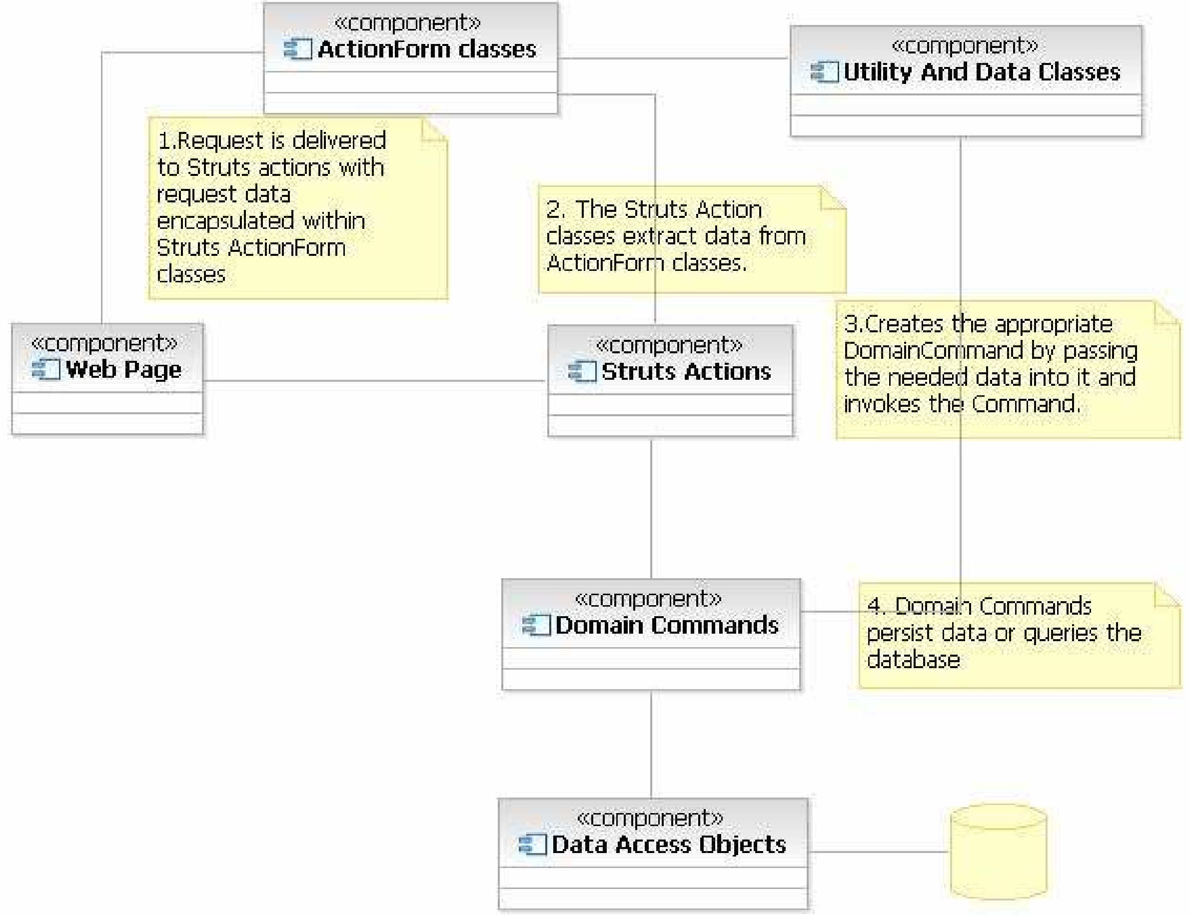

Figure 6.1 The Architectural Layers Of Ticket Seller

6.2.1 Persistence Layer of Ticket Seller

The persistence layer of Ticket Seller is based upon a simple Command pattern implementation. The DatabaseCommand interface defines the contract for concrete DatabaseCommand classes which interact with the database. A Singleton class called PersistenceManager provides convenience methods for handling infrastructural issues such as obtaining a database connection and managing transactions. The persistence layer is summarized in Figure 6.2

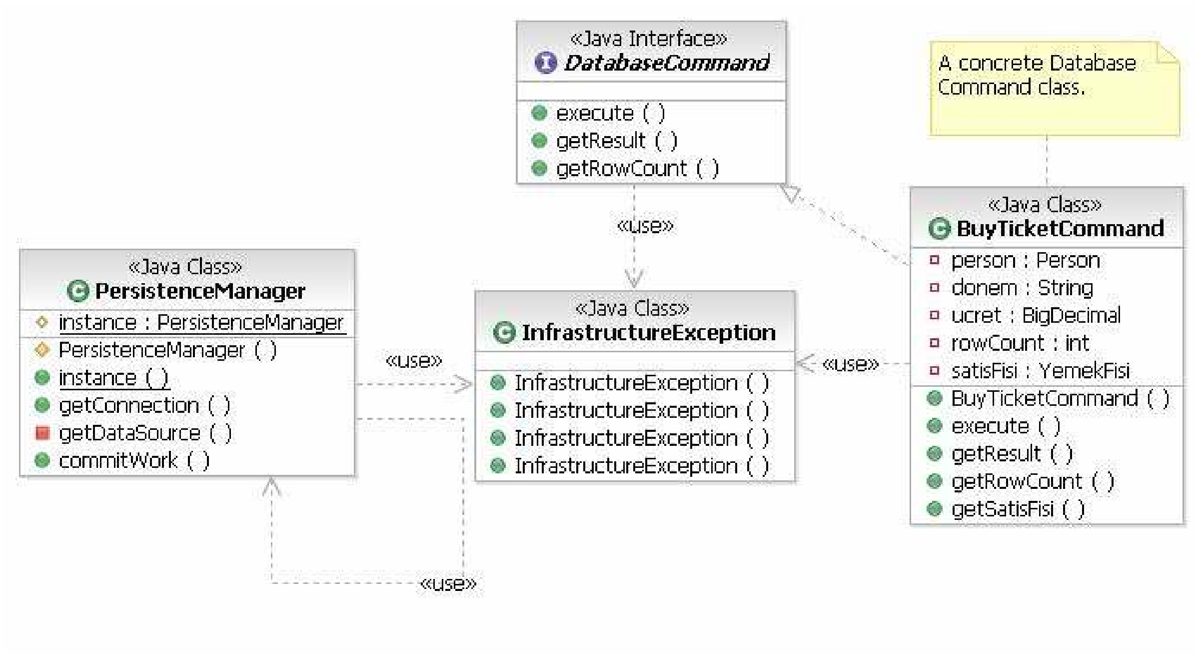

Figure 6.2 The Persistence Layer Of Ticket Seller

6.2.2 The Business Logic Layer of Ticket Seller

The business logic layer of ticket seller is based on a combination of Command and Template method patterns. The DomainCommand interface defines the highest level abstraction for a DomainCommand object, with the DomainCommand.execute() method. The abstract class AbstractBaseCommand provides the base implementation of DomainCommand.execute() method using a template method pattern. Accordiing to this, the execution of a DomainCommand consists of three stages; Firstly, a database transaction is started by AbstractBaseCommand.beginTransaction(). After that the command business logic is executed by the AbstractBaseCommand.invokeCommand() method. Lastly, the transaction is commited or rolled back with the AbstractBaseCommand.commitTransaction() method. The business layer architecture is summarized in **Figure 6.3.**

The business logic layer of Ticket Seller does not implement a domain model. Instead it makes use of simple Command objects which encapsulates pieces of small business logic as Transaction Scripts. The reason of this design decision is due to the simplicity of the application and functional requirements. The absence of a domain layer also eliminates the need for implementing complex architectural infrastructure functionality such as automatic dirty checking, lazy loading of collections and transactional write behind of domain objects.

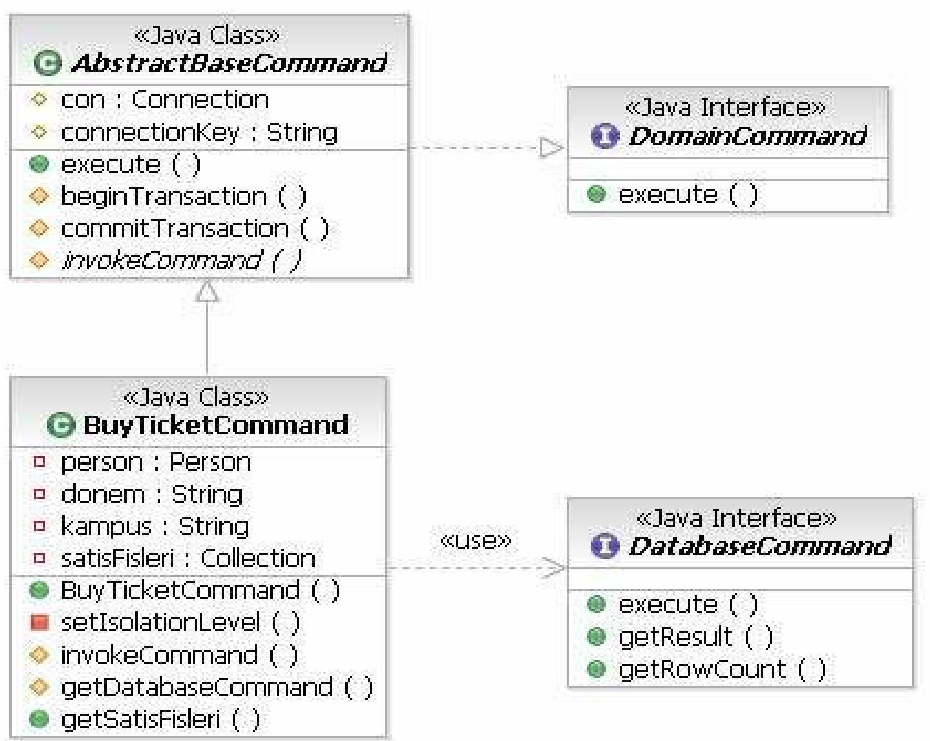

Figure 6.3 The Business Logic Layer Of Ticket Seller

6.2.3 Presentation Layer of Ticket Seller

The presentation layer of Ticket Seller is based on the popular Model View Controller framework Struts. The controller part in Struts based applications is implemented via extending the Struts Action classes. The view is implemented in JSP's using struts tag libraries. The data between the controller and the view is transferred by using subclasses of Struts ActionForm class. The Struts Action classes themseves are an implementation of the Command pattern. The Ticket Seller further enhances that command implementation by introducing a BaseAction class, providing template methods for its subclasses.

The overall architecture of the presentation layer of Ticker Seller is illustrated in **Figure 6-4**.

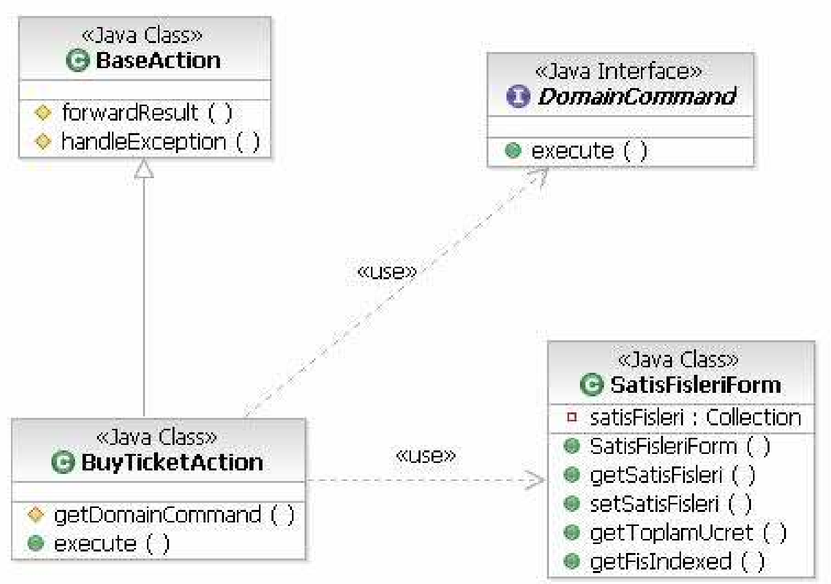

Figure 6.4 The Presentation Layer Of Ticket Seller

6.3 Implementation of the Persistence Layer

The first component of the persistence layer of Ticket Seller is the PersistenceManager class. The PersistenceManager provides some infrastructural convenience methods for managing transactional resources.

```
public class PersistenceManager {
protected static PersistenceManager instance = null;

	protected PersistenceManager() {
	super();
	}

	public static PersistenceManager instance() {
	if(instance == null) {
		instance = new PersistenceManager();
	}
	return instance;
	}

	public Connection getConnection(String key) {
	Connection con;
	try {
		con = getDataSource(key).getConnection();
		con.setAutoCommit(false);
	} catch (SQLException e) {
		throw new InfrastructureException(e.getMessage());
	}
	return con;
	}

	private DataSource getDataSource(String key) {
	DataSource ds = null;
	try {
```

```
                Context ic = new InitialContext();
                ds = (DataSource)ic.lookup(key);
        } catch(NamingException ne) {
                throw new InfrastructureException(ne.getMessage());
        }
        return ds;
        }

        public void commitWork(Connection con) {
        try {
                con.commit();
                con.close();
        } catch(SQLException se) {
                throw new InfrastructureException(se.getMessage());
        }
}

}
```

The PersistenceManager gets the reference to a DataSource via a JNDI lookup, which is dependent on the application server runtime services. So, in order to test it, a mock object library named EJBMock is used.

```
public void setUp() {
      try {
         MockContextFactory.setAsInitial();
         context = new InitialContext();
         context.bind(PluginFactory.getParam("DATA_SOURCE"), new MockDataSource());
      } catch(Throwable t) {
         throw new RuntimeException(t.getMessage());
      }
   }

public void testGetConnection() {
      try {
         Connection con =
PersistenceManager.instance().getConnection(PluginFactory.getParam("DATA_SOURCE"));
         assertNotNull(con);
      } catch(Exception e) {
         fail(e.getMessage());
      }
   }
……
```

While testing the PersistenceManager is straightforward, testing a DatabaseCommand class requires very careful seperation of various concerns. Below is listed a concrete implementation of a DatabaseCommand class;

```
public class GetKullaniciCommand implements DatabaseCommand {
        private int rowCount;
        private ResultSet resultSet;
        private Kullanici kullanici;
        public GetKullaniciCommand(Kullanici kullanici) {
                super();
                this.kullanici = kullanici;
        }

        public void execute(Connection con) throws InfrastructureException {
                PreparedStatement ps = null;
```

```
  try {
    ps = con.prepareStatement(PluginFactory.getParam("GET_KULLANICI_SQL"));
   ps.setString(1, kullanici.getKullaniciAdi());
   ps.setString(2, kullanici.getSifre());
   resultSet = ps.executeQuery();
  } catch(SQLException se) {
        throw new InfrastructureException(se.getMessage());
  }

  }

  public ResultSet getResult() throws InfrastructureException,
    UnsupportedOperationException {
    return resultSet;
  }

  public int getRowCount() throws InfrastructureException,
    UnsupportedOperationException {
    throw new UnsupportedOperationException("Operation Not Supported: A select query
    can return no rowcount value");
  }

}
```

In order to be able to test a database command without using a database, mock implementation of classes Connection, PreparedStatement and ResultSet are needed. Two approaches to mocking these classes may be taken. The first approach is using a dynamic mock library such as EasyMock. Dynamic mock libraries work in a replay approach which does not require the creation of seperate mock objects. The needed mock objects are created by the library using dynamic proxies as long as the required interface definition is provided. The expected calls with their parameters and return values are specified in the setup code;

```
//get mock control
MockControl control = MockControl.createControl(Foo.class);
//Get the mock object from mock control
Foo foo = (Foo) control.getMock();
//get mock control
MockControl control = MockControl.createControl(Foo.class);
//get mock object
Foo foo = (Foo)control.getMock();
//begin behavior definition

//specify which method invocation's behavior
//to be defined.
foo.bar(10);
//define the behavior -- return "ok" when the
//argument is 10
control.setReturnValue("ok");
...
//end behavior definition
control.replay();
```

This apprach seems convenient, however it results in too much set up coding to define the required mock behavior for each class tested. Another approach may be using DBUnit, a unit testing tool for testing database access code. In DBUnit, the test data is saved in XML files, and loaded by DBUnit automatically.

```
public class TestGetKullaniciCommand extends DatabaseTestCase {
   private Connection con;
   IDataSet loadedDataSet;
   private Kullanici kullanici;

          public TestGetKullaniciCommand(String arg0) {
                  super(arg0);
          }

          protected IDatabaseConnection getConnection() throws Exception {
                  Connection con =
MockPersistenceManager.instance().getConnection("DATA_SOURCE");
                  return new DatabaseConnection(con);
          }

          protected IDataSet getDataSet() throws Exception {
                  loadedDataSet = new
FlatXmlDataSet(this.getClass().getClassLoader().getResourceAsStream(PluginFactory.getParam
("KULLANICI_XML")));
                  return loadedDataSet;
          }

          protected DatabaseOperation getSetUpOperation() throws Exception {
                  return DatabaseOperation.CLEAN_INSERT;
          }

          protected DatabaseOperation getTearDownOperation() throws Exception {
                  return DatabaseOperation.NONE;
          }

          public void setUp() throws Exception {
             super.setUp();
                  kullanici = new Kullanici("fatih", "madcoder", "Fatih", "Algan");
          }

          public void testCheckDataLoaded() throws Exception {
             assertNotNull(loadedDataSet);
             int rowCount =
loadedDataSet.getTable(PluginFactory.getParam("KULLANICI_TABLE")).getRowCount();
             assertEquals(1, rowCount);
          }

          public void testExecute() throws Exception {
             GetKullaniciCommand cmd = new GetKullaniciCommand(kullanici);
             Connection con = MockPersistenceManager.instance().getConnection("");
             cmd.execute(con);
             ITable verifyData =
getConnection().createQueryTable("SEARCH_KULLANICI_RESULT",
                  PluginFactory.getParam("IS_KULLANICI_EXISTS_SQL") + "fatih");
                  assertFalse(verifyData.getRowCount() == 0);
          }
}
```

The MockPersistenceManager object used in the method getConnection() here is not really a mock object but instead connects to an in-memory database such as Hsqldb to make the tests run faster insted of a real production type database. The testExecute

method compares the data obtained from the XML file with the data in the in memory database. The database is initialized with the data from the XML file between each test run if CLEAN_INSERT option is specified in the getSetUpOperation() method.

6.4 Implementing the Business Logic Layer

The highest level component of the business layer is the class AbstractBaseCommand, which manages the transaction demarcation boundaries and provides a template for concrete domain commands to execute within the context provided by this template;

```
public abstract class AbstractBaseCommand implements DomainCommand {
        protected Connection con = null;
        protected String connectionKey = null;

        public void execute() {
                con = beginTransaction(connectionKey);
                invokeCommand();
                commitTransaction();
        }

        protected Connection beginTransaction(String key) {
                return PersistenceManager.instance().getConnection(key);
        }

        protected void commitTransaction() {
                PersistenceManager.instance().commitWork(con);
        }

        protected abstract void invokeCommand();
}
```

A concrete domain command class extends from the AbstractBaseCommand, like seen in the example below;

```
public class BuyTicketCommand extends AbstractBaseCommand {
        private Person person = null;
        private String donem = null;
        private String kampus = null;
        private Collection satisFisleri = new ArrayList();

        public BuyTicketCommand(Person person, String donem, String kampus) {
                super();
                this.person = person;
                this.donem = donem;
                this.kampus = kampus;
                connectionKey = PluginFactory.getParam("DATA_SOURCE");
        }

        private void setIsolationLevel() {
                try {

        con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
                } catch(SQLException se) {
                        throw new InfrastructureException(se.getMessage());
```

```
		}
	}

	protected void invokeCommand() {
		setIsolationLevel();
		DatabaseCommand cmd = getDatabaseCommand();
		for(int i = 0; i < person.getFisSayisi().intValue(); i++) {
			cmd.execute(con);
	satisFisleri.add(((edu.iyte.yemekhane.persistence.BuyTicketCommand)cmd).getSatisFis
i());
		}
	}

	protected DatabaseCommand getDatabaseCommand() {
		DatabaseCommand cmd = new
edu.iyte.yemekhane.persistence.BuyTicketCommand(
				person, donem, person.getYemekFisiUcreti(kampus));
		return cmd;
	}

	public Collection getSatisFisleri() {
		return satisFisleri;
	}
}
```

The critical part of a DomainCommand implementation is the invokeCommand() method, which does the real work. In order to provide test isolation and to be able to test without using a database, a stub of the DatabaseCommand must be provided, like the one below;

```
public class TestBuyTicketCommand extends TestCase {
	private MockDatabaseCommand dbCommand;
	private Person person;
	private String donem;
	private String kampus;
	private BuyTicketCommand cmd = null;

	public TestBuyTicketCommand() {
		super();
	}

	public static void main(String[] args) {
		junit.swingui.TestRunner.run(TestBuyTicketCommand.class);
	}

	protected void setUp() throws Exception {
		super.setUp();
		person = new Ogrenci("102001003", "Fatih", "Algan");
		person.setFisSayisi(new Integer(5));
		donem = "Ekim";
		kampus = "Urla";
		dbCommand = new MockDatabaseCommand();
		cmd = new BuyTicketCommand(person, donem, kampus) {
			protected DatabaseCommand getDatabaseCommand() {
				return dbCommand;
			}

			public Connection beginTransaction() {
				return new MockConnection();
```

```
				}
			};
		}

		public void testExecute() {
			cmd.execute();
			assertTrue(dbCommand.invoked());
			assertEquals(cmd.getSatisFisleri().size(), 5);
		}



		public void testGetDatabaseCommand() {
			BuyTicketCommand myCmd = new BuyTicketCommand(person, donem,
	kampus);
			assertTrue(myCmd.getDatabaseCommand() instanceof
	edu.iyte.yemekhane.persistence.BuyTicketCommand);
		}
	}
```

Here, the setUp() method returns an overriden version of BuyTicketCommand which returns stubs of DatabaseCommand and Connection interfaces that makes it possible invoking the DomainCommand.execute() method possible without interacting with the database. The stub version of DatabaseCommand is just a simple spy object that records the invocation and makes sure that the DomainCommand has attempted to invoke it.

When the DomainCommand orders the persistence layer for only an update operation to the database the implementation is straightforward as seen above. However, it order for a query to load objects into the memory, the implementation gets more complicated. The tests must make sure that the ResultSet returned from the persistence layer is processed correctly by the DomainCommand and the in memory objects are correctly constructed in the memory. Unluckily, this operation brings a high coupling to infrastructure code, since the ResultSet must be obtained from a PreparedStatement and iterated over to construct the needed objects. To mock this functionality, a set of mock objects are implemented, which provides the desired decoupling to the database and infrastructural code, as seen in **Figure 6.5**.

While seeming complex, some of the implementation classes such as MockConnection has many empty methods, just in order to satisfy the contract of the java.sql.Connection interface. The highest level class of the hierarchy is MockDatabaseCommand, which is a stub of DatabaseCommand interface. This is the class that the DomainCommand implementations directly interact with, as can be seen in the previous example BuyTicketCommandTest.

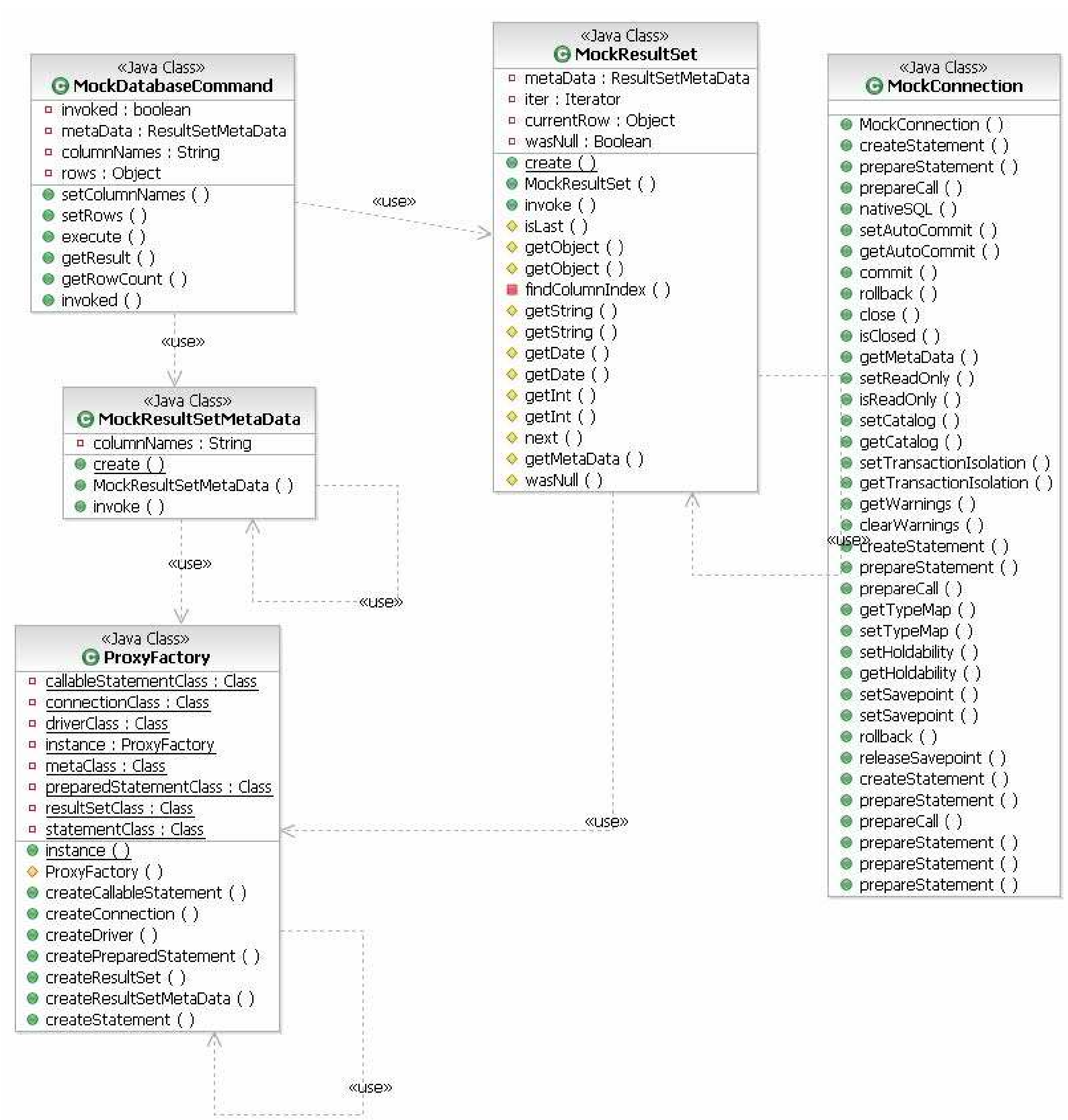

Figure 6.5 The Mock Persistence Layer Implementation

```
public class MockDatabaseCommand implements DatabaseCommand {
        private boolean invoked = false;
        private ResultSetMetaData metaData = null;
        private String[] columnNames = null;
        private Object[][] rows = null;

        public void setColumnNames(String[] columnNames) {
                this.columnNames = columnNames;
        }

        public void setRows(Object[][] rows) {
                this.rows = rows;
        }

        public void execute(Connection con) throws InfrastructureException {
```

```
            invoked = true;
        }

        public ResultSet getResult() throws InfrastructureException,
                    UnsupportedOperationException {
            metaData = MockResultSetMetaData.create(columnNames);
            return MockResultSet.create(metaData, rows);
        }

        public int getRowCount() throws InfrastructureException,
                    UnsupportedOperationException {
            return 0;
        }

        public boolean invoked() {
            return invoked;
        }
    }
```

The second important part of the puzzle is the MockResultSetMetaData class, which makes use of dynamic proxies in order to intercept the calls to methods like getColumnCount() or getColumnName() on itself.

```
public class MockResultSetMetaData implements InvocationHandler {
  private String[] columnNames = null;

  public static ResultSetMetaData create(String[] columnNames) {
    return ProxyFactory.instance().createResultSetMetaData(
      new MockResultSetMetaData(columnNames));
  }

  public MockResultSetMetaData(String[] columnNames) {
    super();
    this.columnNames = columnNames;
  }

  public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    String methodName = method.getName();
    if (methodName.equals("getColumnCount")) {
      return new Integer(this.columnNames.length);
    } else if (methodName.equals("getColumnName") ||
methodName.equals("getColumnLabel")) {
      int col = ((Integer) args[0]).intValue() - 1;
      return this.columnNames[col];
      // stub out other methods for now
    } else {
      Class returnType = method.getReturnType();
      if (returnType.equals(String.class)) {
        return "";
      } else if (returnType.equals(Integer.TYPE)) {
        return new Integer(0);
      } else {
        return Boolean.FALSE;
      }
    }
  }
}
```

The MockResultSetMetaData class makes use of a singleton class named ProxyFactory which creates proxies for the desired interfaces, as seen below;

```
public class ProxyFactory {
private static final Class[] callableStatementClass = new Class[] { CallableStatement.class };
private static final Class[] connectionClass = new Class[] { Connection.class };
private static final Class[] driverClass = new Class[] { Driver.class };
private static final ProxyFactory instance = new ProxyFactory();
private static final Class[] metaClass = new Class[] { ResultSetMetaData.class };
private static final Class[] preparedStatementClass = new Class[] { PreparedStatement.class };
private static final Class[] resultSetClass = new Class[] { ResultSet.class };
private static final Class[] statementClass = new Class[] { Statement.class };

public static ProxyFactory instance() {
        return instance;
}

protected ProxyFactory() {
        super();
}

public CallableStatement createCallableStatement(InvocationHandler handler) {
        return (CallableStatement)
        Proxy.newProxyInstance(handler.getClass().getClassLoader(),
        callableStatementClass, handler);
}

public Connection createConnection(InvocationHandler handler) {
        return (Connection) Proxy.newProxyInstance(handler.getClass().getClassLoader(),
        connectionClass, handler);
}

public Driver createDriver(InvocationHandler handler) {
        return (Driver) Proxy.newProxyInstance(handler.getClass().getClassLoader(),
        driverClass, handler);
}

public PreparedStatement createPreparedStatement(InvocationHandler handler) {
        return (PreparedStatement)
        Proxy.newProxyInstance(handler.getClass().getClassLoader(),
        preparedStatementClass, handler);
}

public ResultSet createResultSet(InvocationHandler handler) {
        return (ResultSet) Proxy.newProxyInstance(handler.getClass().getClassLoader(),
        resultSetClass, handler);
}

public ResultSetMetaData createResultSetMetaData(InvocationHandler handler) {
        return (ResultSetMetaData)
        Proxy.newProxyInstance(handler.getClass().getClassLoader(),
                metaClass, handler);
}

public Statement createStatement(InvocationHandler handler) {
        return (Statement) Proxy.newProxyInstance(handler.getClass().getClassLoader(),
                statementClass, handler);
}
}
```

The last component for mocking the database access logic is the MockResultSet class, which also makes use of dynamic proxies to intercept on calls made onto itself. MockResultSet class makes use of MockResultSetMetaData to interprete the schema of the database;

```
public class MockResultSet implements InvocationHandler {
	private ResultSetMetaData metaData = null;
	private Iterator iter = null;
	private Object[] currentRow = null;
	private Boolean wasNull = Boolean.FALSE;

	public static ResultSet create(
		ResultSetMetaData metaData,
		Object[][] rows) {
		return ProxyFactory.instance().createResultSet(
			new MockResultSet(metaData, rows));
	}

	public MockResultSet(ResultSetMetaData metaData, Object[][] rows) {
		super();
		this.metaData = metaData;
		this.iter = (rows == null) ? Collections.EMPTY_LIST.iterator()
			: Arrays.asList(rows).iterator();
	}

	public Object invoke(Object proxy, Method method, Object[] args)
		throws Throwable {
		String methodName = method.getName();
		if (methodName.equals("getMetaData")) {
			return this.getMetaData();
		} else if (methodName.equals("next")) {
			return this.next();
		} else if (methodName.equals("previous")) {//do nothing
		} else if (methodName.equals("close")) {//do nothing
		} else if (methodName.equals("getObject")) {
			if (args[0] instanceof Integer) {
				int col = ((Integer) args[0]).intValue();
				return this.getObject(col);
			} else if (args[0] instanceof String) {
				return this.getObject((String) args[0]);
			}
		} else if (methodName.equals("getString")) {
			if (args[0] instanceof Integer) {
				int col = ((Integer) args[0]).intValue();
				return this.getString(col);
			} else if (args[0] instanceof String) {
				return this.getString((String) args[0]);
			}
		} else if (methodName.equals("getInt")) {
			if(args[0] instanceof Integer) {
				int col = ((Integer) args[0]).intValue();
				return new Integer(this.getInt(col));
			} else if(args[0] instanceof String) {
				return new Integer(this.getInt((String)args[0]));
			}
		} else if (methodName.equals("getDate")) {
			if(args[0] instanceof Integer) {
				int col = ((Integer) args[0]).intValue();
				return this.getDate(col);
```

```
			} else if(args[0] instanceof String) {
				return this.getDate((String)args[0]);
			}
		} else if (methodName.equals("wasNull")) {
			return this.wasNull();
		} else if (methodName.equals("isLast")) {
			return this.isLast();
		}
		return null;
	}

	protected Boolean isLast() throws SQLException {
		return this.iter.hasNext() ? Boolean.FALSE : Boolean.TRUE;
	}

	protected Object getObject(int columnIndex) throws SQLException {
		Object obj = this.currentRow[columnIndex - 1];
		if (obj == null) {
			this.wasNull = (obj == null) ? Boolean.TRUE : Boolean.FALSE;
		}
		return obj;
	}

	protected Object getObject(String columnName) throws SQLException {
		return this.getObject(this.findColumnIndex(columnName));
	}

	private int findColumnIndex(String columnName) throws SQLException {
		for (int i = 0; i < this.currentRow.length; i++) {
			int c = i + 1;
		if (this.metaData.getColumnName(c).equalsIgnoreCase(columnName)) {
			return c;
		}
	}
		throw new SQLException(columnName + " is not a valid column name.");
	}

	protected String getString(int columnIndex) throws SQLException {
		Object obj = this.getObject(columnIndex);
		return (obj == null) ? null : obj.toString();
	}

	protected String getString(String columnName) throws SQLException {
		Object obj = this.getObject(this.findColumnIndex(columnName));
		return (obj == null) ? null : obj.toString();
	}

	protected Date getDate(int columnIndex) throws SQLException {
		Object obj = null;
		try {
			obj = this.getObject(columnIndex);
			return (obj == null) ? null : (Date)obj;
		} catch(Exception re) {
			throw new SQLException("Bad Date Format: " + obj.toString());
		}
	}

	protected Date getDate(String columnName) throws SQLException {
		Object obj = null;
		try {
```

```
			obj = this.getObject(this.findColumnIndex(columnName));
			return (obj == null) ? null :
			newDate(DateUtils.convertDate(obj.toString()).getTime());
		} catch(Exception re) {
			throw new SQLException("Bad Date Format: " + obj.toString());
		}
	}

	protected int getInt(int columnIndex) throws SQLException {
		Object obj = this.getObject(columnIndex);
		return (obj == null) ? 0 : new Integer(obj.toString()).intValue();
	}

	protected int getInt(String columnName) throws SQLException {
		Object obj = this.getObject(this.findColumnIndex(columnName));
		return (obj == null) ? 0 : new Integer(obj.toString()).intValue();
	}

	protected Boolean next() throws SQLException {
		if (!this.iter.hasNext()) {
			return Boolean.FALSE;
		} else {
			this.currentRow = (Object[]) iter.next();
			return Boolean.TRUE;
		}
	}

	protected ResultSetMetaData getMetaData() throws SQLException {
		return this.metaData;
	}

	protected Boolean wasNull() throws SQLException {
		return this.wasNull;
	}
}
```

Finally, a test case that makes use of the mock persistence infrastructure layer is given below showing how a mock data access layer for the ticket seller can be plugged under the business logic for testing;

```
public class TestSearchOgrenciCommand extends TestCase {
	private Ogrenci ogrenci = null;
	private MockDatabaseCommand dbCommand;
	String[] columnNames = new String[] {"OGRENCI_NO", "AD", "SOYAD"};
	Object[] row1 = new Object[] {"100101001", "UTKAN A.", "KILIÇ"};
	Object[] row2 = new Object[] {"100101002", "GÖKTUĞ", "KARPAT"};
	Object[][] rows = new Object[][] {row1, row2};

	private SearchOgrenciCommand cmd = new SearchOgrenciCommand() {
		protected DatabaseCommand getDatabaseCommand() {
		return dbCommand;
	}

	public Connection beginTransaction() {
		return new MockConnection();
	}
	};
```

```
	public static void main(String[] args) {
		junit.swingui.TestRunner.run(TestSearchPersonelCommand.class);
	}
	protected void setUp() throws Exception {
	super.setUp();
	ogrenci = new Ogrenci();
	ogrenci.setNo("100101001");
	dbCommand = new MockDatabaseCommand();
	dbCommand.setColumnNames(columnNames);
	dbCommand.setRows(rows);
	cmd.setOgrenci(ogrenci);
	}

	protected void tearDown() throws Exception {
	super.tearDown();
	}

	public void testExecute() {
	cmd.execute();
	assertTrue(dbCommand.invoked());
	ArrayList list = new ArrayList();
	list.add(new Ogrenci((String)row1[0], (String)row1[1], (String)row1[2]));
	list.add(new Ogrenci((String)row2[0], (String)row2[1], (String)row2[2]));
	assertEquals(cmd.getPersonList(), list);
	}

	public void testGetDatabaseCommand() {
		SearchOgrenciCommand myCmd = new SearchOgrenciCommand();
		myCmd.setOgrenci(ogrenci);
		assertTrue(myCmd.getDatabaseCommand() instanceof GetOgrenciCommand);
	}
}
```

6.5 Implementing the Presentation Layer

The presentation layer of the Ticket Seller is based on the Struts MVC framework. A high level architectural view of the Struts framework is presented in Figure 6.6. The Controller ActionServlet is a variation of the FrontController pattern, who decides on which controller logic to invoke and determines the next view based on the key returned from the controller logic invoked. Action classes are an implementation of the Command Pattern, which executes the controller logic delegated to them by the ActionServlet. ActionForms represent the model data to be displayed in the view. The view consists of JSP pages and a rich set of special custom JSP tags which injects the request parameters into ActionForms behind the scenes. This architecture has two important consequences from a testing perspective. Firstly, since the controller dispatching rules are kept in a configuration file (struts-config.xml), testing the controller invocation and view dispatching logic can be tested just by testing the configuration file. The framework handles the necessary machinery for page flow and controller execution. The second effect is that since the request and session data are encapsulated in special bean classes (ActionForm), and the request parameters are bound to the properties of these beans by the framework automatically, extracting data from the HttpRequest and HttpSession can be avoided if the application is carefully designed. As noted in section in **5.3.4.2**, a servlet which processes a client request had three seperate responsibilies;

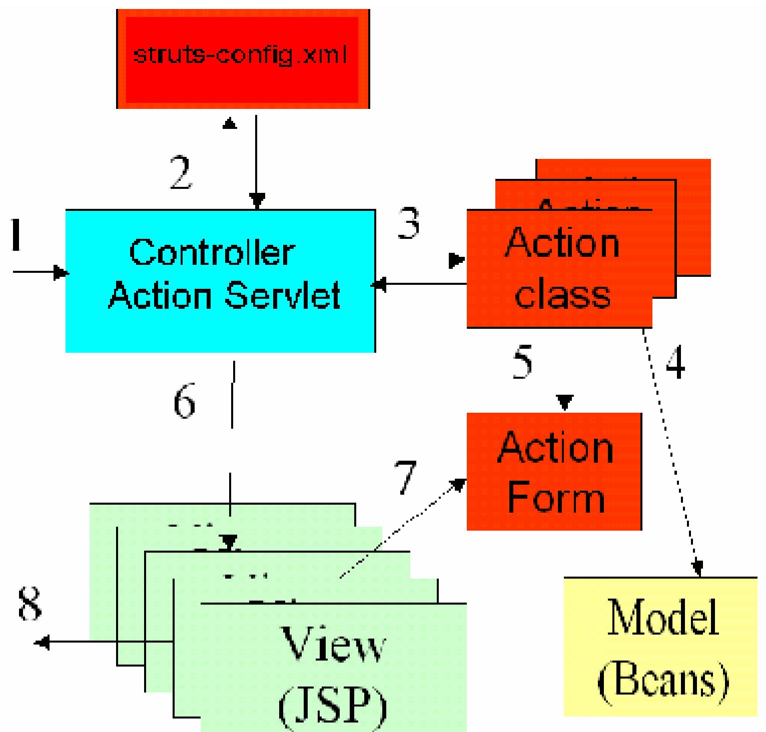

Figure 6.6 High Level Architecture Of The Struts Framework

- Receiving a request from the web container.
- Choosing some business logic to execute based on the request.
- Extracting data from the request and pass it to the business logic as parameters.

By paying attention to the rule of thumbs mentioned, two of these responsibilities may be completely delegated to the framework, leaving the developer the task of choosing the business logic to execute only.

The controller logic of the TicketSeller application has a base Action class into which the common functionality amongst all the Action classes are factored out. The architecture of the TicketSeller application is summarized in **Figure 6.7**

Testing of the presentation layer is done via an open source Struts testing framework called StrutsTest, which is designed to test Struts based web applications. StrutsTest provides out of container testing capabilities by MockStrutsTestCase class. This class also tests the request dispatching and view forwarding logic by inspecting the struts configuration file, which eliminates the need to test the navigation logic seperately. The architecture of the presentation layer testing infrastructure is summarized in **Figure 6.8**

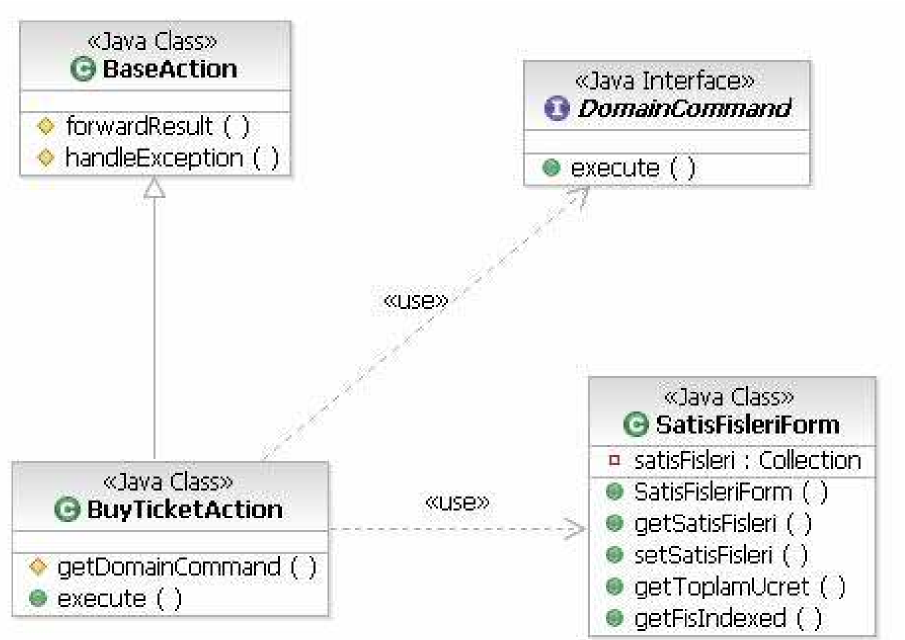

Figure 6.7 Class Hierarchy For Presentation Layer Of The Ticket Seller

The most important component of the presentation testing infrastructure is the class BaseActionTestCase which controls the main flow of testing an Action class;

```
public abstract class BaseActionTestCase extends MockStrutsTestCase  {
        public static final String HAPPY_PATH = "success";
        protected ActionMapping mapping = new ActionMapping();
        protected ActionForward forward = null;
        protected ActionForm form = null;
        protected Action action = null;
        protected List actionListeners = new ArrayList();

        protected void setUp() throws Exception {
                super.setUp();
                setContextDirectory(new File("WebContent"));
        }

        protected void tearDown() throws Exception {
                super.tearDown();
        }

        protected abstract void setAction();
                protected abstract void addActionForwards();
                protected abstract void setActionForm();

                public void addActionListener(ActionListener listener) {
                        actionListeners.add(listener);
                }

                public void removeActionListener(ActionListener listener) {
                        actionListeners.remove(listener);
                }

                public void actionPerform() {
                        try {
                        setAction();
```

```
                addActionForwards();
                setActionForm();
                forward = action.execute(mapping, form, request, response);
                Iterator it = actionListeners.iterator();
                while(it.hasNext()) {
                        ActionListener listener = (ActionListener)it.next();
                        listener.actionExecuted(action);
                }
                } catch(Throwable t) {
                        fail("Unexpected Exception: " + t.getMessage());
                }
        }

        public void checkListenerInvocations() {
                //Check if the the desired business logic invoked
                Iterator it = actionListeners.iterator();
                while(it.hasNext()) {
                        ActionListener listener = (ActionListener)it.next();
                        assertTrue("Registered Domain Command Method Not
                        Invoked", listener.invoked());
                }
        }

}
```

By providing the template method actionPerform() which calls the abstract methods setAction(), addActionForwards() and setActionForm() respectively, all the configuration will be setup. All needed to be done by the testcase classes is to override these three abstract methods appropriately. There’s also a spy listener component in the BaseActionTestCase, which consists of a set of ActionListener classes. The ActionListener makes sure that the Action class execute method has been invoked during the test run, which is expected to delegate the task to be done to a mock implementation of a DomainCommand class.

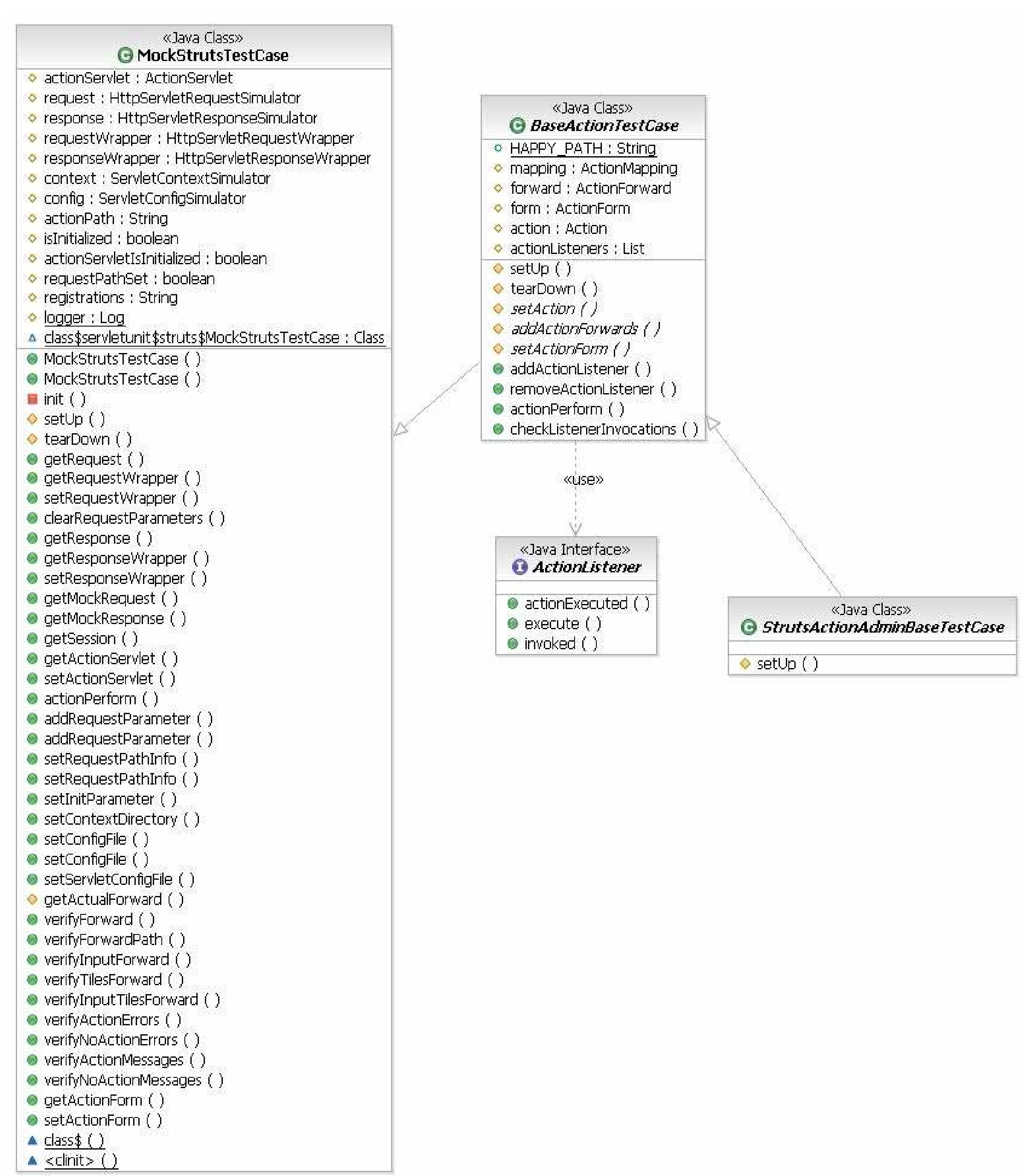

Figure 6.8 Class Hierarchy For Presentation Layer Tests

The following code snippet illustrates how the tests are run;

```
public class TestSearchPersonAction extends StrutsActionClerkBaseTestCase {
	private String sicilNo = null;
	private static final String FORWARD_HAPPY_PATH = "/clerk/listpersons.jsp";

	public void setUp() throws Exception {
		super.setUp();
		sicilNo = "2";
	}
```

```
        protected void setAction() {
                action = new SearchPersonAction() {
                        public DomainCommand getDomainCommand(String sicilNo, String
                        type) {
                                SearchPersonelCommand cmd = new
                                MockSearchPersonelCommand();
                                        cmd.setPersonel(new
                                        Personel(Integer.valueOf(sicilNo), null, null));
                                return cmd;
                        }
                };
        }

        protected void addActionForwards() {
                mapping.addForwardConfig(new ActionForward(HAPPY_PATH,
                FORWARD_HAPPY_PATH, false));
        }

        protected void setActionForm() {
                form = new PersonForm();
                ((PersonForm)form).setType("Personel");
                ((PersonForm)form).setNo(sicilNo);
        }

        public void testListenersInvoked() {
                super.actionPerform();
                checkListenerInvocations();
        }

        public void testExecuteHappyPath() {
                super.actionPerform();
                //Check if the forward action is set.
                assertNotNull(forward);
                //Check if the result is really the happy path.
                assertEquals(FORWARD_HAPPY_PATH, forward.getPath());
                //Check if the view data is set correctly.
                assertTrue(((PersonForm)form).getPersonList().size() > 0);
        }

        public void testGetDomainCommand() {
                SearchPersonAction action = new SearchPersonAction();
                DomainCommand cmd = action.getDomainCommand("12", "Personel");
                assertTrue(cmd instanceof SearchPersonelCommand);
                cmd = action.getDomainCommand("12", "Öğrenci");
                assertTrue(cmd instanceof SearchOgrenciCommand);
        }
}
```

The most critical section of the above test case is where a mock version of a DomainCommand is created, shown in bold letters. This mock domain class is also a spy implementing the ActionListener interface, which records the business logic invocations triggered by the Action under test;

```
public class MockSearchPersonelCommand extends SearchPersonelCommand implements
        ActionListener {
        private boolean invoked = false;
        public void actionExecuted(Action action) {
                invoked = true;
        }
        public boolean invoked() {
                return invoked;
        }
        public void execute() {
                invoked = true;
                personList.add(new Personel(new Integer(2), "ERKAN", "KARABULUT"));
                personList.add(new Personel(new Integer(3), "YILDIRAY", "OKUŞ"));
        }
}
```

6.6 Summary

The design and architectural structure of of an application has a significant impact on how it is tested. The ticket seller has no domain model, based on business domain objects due to its simplicity and small size. The business logic of the TicketSeller consists of a number of TransactionScripts encapsulated in Command Objects. The data access layer is not completely decoupled from the business layer, however it is sufficient flexible to change the underlying RDMS underneath without breaking the business layer or removing the database while unit testing the business layer code. The presentation layer of the TicketSeller application is based on Struts. Struts has been one of the first frameworks that gained wide acceptence in the community. However, Struts reflects its strong coupling with the Java servlet API to the developer directly as well as a high coupling between the application code and the framework API itself, which is a flaw that is avoided in most of the newer generation frameworks. This impact makes testing Struts application without third party mock object libraries out of the container almost impossible. Despite those disatvantages, there are numerous third party extension libraries available for testing Struts applications, because of Struts's wide use in community and industry.

CHAPTER 7

SOFTWARE METRICS AND TEST DRIVEN DEVELOPMENT

The term software metric refers to any measurement related to software development “(McConnell, 1993)”. Metrics are defined for measuring the development process and the produced software. Software metrics can hint at possible anomalies in a negative or positive sense.

7.1 Classification of Software Metrics

Sommerville, “(Sommerville, 1996)” classifies metrics into two classes, control metrics and predictor metrics. Control metrics measure the process attributes to control the development process. Predictor metrics measure product attributes in order to predict the product quality.

Another possible classification of software metrics is the kind of measured entity. This may comprise conceptional, material or ideal entities such as processes, persons or programs respectively “(Fenton, Pfleeger, 1996)”. Consequesntly, there are three kinds of software metrics;

- Process metrics – for software changing activities
- Resource metrics – for process supporting resources
- Product metrics – for artifacts created by the development process.

7.1.1 Software Process Metrics and TDD

Every activity during software development can be seen as a process. The measurement of such processes aims to provide a better understanding of the process efficiency. Typical process metrics include defect discovery rate, defect backlog, test progress “(Bassin, Biyani, Santhanam, 2002)” and communication metrics “(Allen, Dutoit, Bruegge, 1998)”.

SW-CMM defines a set of key process areas(KPA) “(Paul, Weber, Curtis, 1995)”. During a CMM certification, feedback is provided for each KPA evaluating the extent to which the organization meets pre-established criteria in the areas of activities and infrastructure, institutionalization and goals. SW-CMM focuses on measuring the capability of a development organization to perform a process rather than measuring the process itself.
Compared to SW-CMM, agile development practices is more focused on producing high quality and running software than organizational maturity “(Ken, 2002)”. The Personal Software process introduces individual level metrics. The primary goals of PSP are to improve project estimation and quality assurance. The collected metrics include size, time and defect data. The collection of these metrics has to be performed by the developer in parallel with regular development activities. However, observations

show that the adoption of continious data gathering by developers as suggested by PSP, is very low in practice because of the overhead and context switch associated with the data gathering, which requires a very high level of personal discipline “(Johnson, et. al., 2003)”.

Weinberg proposes an observational model to make sense of the observed artifacts “(Gerald, 1993)”. An observation model aims to tell;

- What to measure
- How to interprete the measured data
- How to find the significant observations
- Which control actions to take

An effective observation model should be able to identify the state of a system. Knowing this state allows a controller to take appropriate actions.

TDD, by itself, does not define the software process metrics directly. If it is embedded in a development method such as Extreme Programming, the number of implemented user stories or task cards measure the progress of a development team “(WEB_11, 2005)”.

7.1.2 Software Product Metrics

Software product metrics are calculated from the produced artifacts of a software development process. Typically, their calculation can be performed completely automatically. Within a development project, their goal is to predict attributes of the produced system, such as maintainability and understandability, such as complexity metrics. Some metrics try to rate the design quality, such as coupling metrics. Not only the production code, but also the test harness can be the subject of software product metrics. An example of this can be test coverage metrics.

7.1.2.1 Complexity Metrics

Complexity metrics try to rate the maintainability and understandability of a system. Lines Of Code(LOC) is perhaps the most basic metric. The size of software entities can be seen as an indicator of their complexity. In practice, however, the calculation of LOC bears many ambiguities. For example, LOC could be defined either as a physical line of code or as a logical statement “(Jones, 2000)”.

Weighted method count (WMC) is another complexity metric. This metric sums up the complexity indices for all methods of a class. These complexity measures of the individual methods are usually given by code complexity metrics such as LOC or McCabe cyclomatic complexity “(Chidamber, Kemerer, 1994)”.

Fenton “(Fenton, 1994)” shows that a search for general software complexity metrics is doomed to fail. However, measurement theory helps to define and validate measures for some specific complexity attributes.

One of the goals of TDD is to keep a software maintainable and understandable. New features are added incrementally. The costs for adding new features must not become prohibitively expensive. Therefore a TDD team must keep the complexity of the application under development low. XP expresses this desire in its practices of

Simple Design and Not Implementing Any Functionality until it is needed. This goal is achieved by driving the code with tests and continious refactoring. However, as of today, there is no comprehensive empirical data available to support that claim.

Coupling and Cohesion

Coupling and cohesion metrics try to rate the design quality of an application. The focus of coupling models is to see how strongly components are tied together. Coupling models exist on different levels of granularity, such as objects or packages. It is seen as a good design if software entities that implement seperate concepts have few dependencies between each other.

Cohesion models describe how closely the elements of one component are related to each other. It is seen as good design if software entities that add to the implementation of the same concept have strong relationships among each other.

TDD nearly automatically leads to a decoupled design "(Martin, 2003)". In order for the production code to be testable, it must be callable in isolation and decoupled from its environment and dependencies. The act of driving the production code using tests forces the developer to decouple the application design. While TDD reduces the coupling of the production code, it introduces a coupling between test and production code. This coupling might impact upon developement and production. For example, in order to run the tests against some test data, it must be possible to provide this data programmatically.

TDD also has an important effect on cohesion. The single responsibility principle(every service has only one home) directs a programmer to the seperation of coupled responsibilities. Each responsibility is an axis of change. Mixing responsibilities means mixing axes of change. If the implementation of a new feature touches one such responsibility, then –in theory – it will be abstracted into a new abstraction by removing duplication in the application "(WEB_11, 2005)". Seperated responsibilities would be decoupled, thus increasing the cohesion of the seperated entities.

Test Coverage

Test coverage is the percentage of elements required by a test strategy that have been exercised by a given test harness "(Binder, 1999)" . Test coverage is either specification based or code based. Some of the most well known code-based coverage models are as follows;

- **Statement Coverage** is the percentage of all source code statements that were touched at least once during a run of the test harness. A statement coverage of %100 does not guarantee the absence of errors.
- **Branch Coverage** is the percentage of all possible branches in a program that were touched at least once. It is a stronger criterion than statement coverage but still does not guarantee error free code.
- **Path Coverage** is the percentage of all possible paths – combinations of branches – that were touched at least once. Still a %100 path coverage does not guarantee error free code.

Commercial tools typically calculate statement coverage. The capabilities of more sophisticated test coverage tools are limited by polymorphism and dynamic class loading. The use of test coverage tools is vigorously discussed in TDD community. On the one hand, a program emerging from a TDD process should have a very high test

coverage. On the other hand, if controlled by a tool, people tend to optimize the metric rather than the goal. Even worse, a high test coverage does not guarantee that the unit tests really test the core of an application and express the intent of the user stories appropriately. Low test coverage is, however, an indicator that the developers did not work properly.

Code based test coverage tools can either work dynamically or statically. Dynamic tools observe a program while executing its test harness, such as Clover. Some tools instrument the source code by weaving in coverage measuring aspects. Other possibilities(at least in Java) are instrumentation of byte code or the use of the Java Virtual Machine debugging interface. Static tools, such as NoUnit calculate the test coverage by static analysis of the source code only. They use models such as control flow graphs, program dependence graphs or program slicing.

7.1.2.2Measurement over Time

Some metrics only make sense if they are observed over time. For example, the plain number of touched modules in a new release of a software system contains little valuable information. However, if the fraction of touched modules continiously grows for successive releases, then this observation can be taken as a sign of increasing complexity. Some other metrics may change thier meaning over time, for example, one measurement of user satisfaction is clearly a product measure, but plotting the trend of several measurements of user satisfaction over time or accross different products might indicate to a process measure. A set of tools generally available tools that allow the ongoing collection of metrics over time exist in the market, such as JMetra.

7.2 Software Inspection

Software inspections are peer reviews of a programmer's work to find problems and improve quality. The goal of inspection is finding errors in design and code. The types of software artifacts submitted to an inspection process typically includes requirements documents, design, code or test cases.
The discipline of code reading has established a number of techniques that help with the understanding of large amounts of source code. Spinelli compares code reading with the way an engineer examines a machine "(Diomidis, 2003)". Code reading not only enforces programming standarts, but facilitates a quicker understanding of the existing system. Understanding open source software is one possible area of application.
For a TDD developer, software inspections and reading techniques are specially important because in TDD the source code is the only reliable source of reference. In the context of Extreme Programming, programming is performed by pairs of developers. This is a kind of continious software inspection or peer review. Another important XP practice is collective code ownership, which leads to a continious inspection of other team members' code.
For a TDD process assessor, inspection techniques are ven more important, because he or she is confronted with all the source code as a whole, whereas a TDD developer can build up knowledge of the source code incrementally.

7.3 TDD Process Assessment

Process assessment aims to uncover method or process misuse. For example, a team of developers might state that they are doing Extreme Programming with the arguement that they are not documenting anything.
In a study of documenting the assessment of a TDD Project "(Maximilien, Williams, 2003)", the developers were inexperienced in TDD and needed an introduction to this style of development. The project was closely monitored because it was the first TDD project in this development's organization. One experience of the authors was that the developers needed continious encouragement to add unit tests to the system, for new features and bug fixes. Therefore, the process assessors set up a monitoring system that reported the relative number of unit tests for every subsystem. The developers needed a third of the overall project time until they appreciated the test harness.

7.3.1 Retrospectives

One of the major characteristics of agile methodologies is the emphasis on the team to organize itself. This contrasts to most traditional development methodologies. Cockburn states that the people involved in the development process are the most important factor in guaranteeing the success of a project "(Cockburn, 2002)". Boehm concludes that agile retrospectives and reflections enable continious process and technical improvement "(Boehm, Turner, 2003)"
The self organization aim at harvesting the experiences gained by the team members throughout the project with interviews, workshops, study groups or project reviews. Some agile methodologies advocate pair programming and collective code ownership as an extreme form of such reviews where developers review other team members continiously.

Such retrospectives typically concentrate on the established tracking mechanisms and personal experience of the team members "(Kerievsky, 2004)". Usually, there are only a few documents available describing the project's evolution because agile methods, especially TDD, value running software over comprehensive documentation.

7.3.2 TDD Specific Measurements

The retrospective activities mentioned above provide more qualitative observations about the state of a development process. On the other hand, TDD specific measurements may allow the derivation of more quantitative observations from the produced source code. Wege "(WEB_11, 2005)" proposes the following as TDD specific measurements;

- **Test Coverage for Integration Deltas:** Typically current test coverage tools concentrate on determining the test coverage of the most current integration

version. Such tools neglect the process aspect of building a highly covering test harness, piece by piece. Every change in production code is deriven by tests. Hence every production code change in every integration delta is covered comprehensively by the tests that were added or changed in that integration delta. The only exception are refactorings. They do not have to be covered by tests.

- **Test Code Versus Production Code Ratio:** Beck observes that a TDD developer will likely end up with about the same number of lines of test code as model code "(Beck, 2003)". This test code must be available during daily development. One of the regular requests in TDD process assessments is to show the tests. Some modern project management tools like Maven even have a seperate tag for identifying test code.
- **Continious Integration:** The rule to chek in only clean code leads to the attitude that developers try to integrate and commit as soon as possible. The do not want thier tests get broken by other programmers' changes. Practically, this means that a developer commits rather small amounts of changes. In TDD, the smallest amount of cohesive changes is the implementation of a new feature because in order to allow for a commit all tests must run and all duplications must be removed by refactoring. Thus, an analysis of the commit log can show that the integration deltas are neither too small nor too big.
- **Large Refactorings Stink:** Basically, refactorings should be done in small steps and continiously. If a TDD developer finds himself making large refactorings, that indicates he should had done many smaller refactorings earlier and he got lazy. There are, however, exceptions to this rule. Sometimes a big refactoring might be inevitable due to a customer request or a technical issue.
- **Low Coupling, High Cohesion:** Low coupling and high cohesion is generally regarded as good object oriented design. Writing tests first, helps to achieve that goal. Being able to perform each test in isolation requires the feature under test to be decoupled from the rest of the system. Also the desire to minimize fixture code in the automated tests requires a high degree of cohesion of a feature in production code. Typically, high coupling and low cohesion indicates that the refactoring step was not performed thoroughly enough.
- **Use of Advanced Testing Techniques:** Test driving every single change to a system requires advanced testing techniques in many cases. The literature contains a number of advanced techniques such as MockObjects. The absence of advanced testing techniques might indicate that not all changes to a system were actually driven by tests.

7.4 Evaluating Product Metrics of Ticket Seller Application

In this section, the Ticket Seller application will be analyzed according to some of the product metrics mentioned above. The analysis presented here is applied on a scaled down version of the Ticket Seller application in order to shorten the content of the results and communicate them beter. The packages retrospected are randomly selected from the domain layer of the application.

The tool used for retrospecting the Ticket Seller application is an open source test coverage tool named Cobertura. Cobertura calculates the percentage of code

accessed by tests. It can be used to identify which parts of a Java program are lacking test coverage. Cobertura offer the following features;

- Shows percent of lines coveraged and branches coveraged for each class, package and for the overall project
- Shows the McCabe cyclomatic code complexity for each class, and the average cyclomatic code complexity for each package, and for the overall product.
- Reports the analysis results in HTML or XML format.

The following table shows an example of treshold values for cyclomaticcomplexity taken from Carnegie Melon Software Engineering Institute “(WEB_12)”.

Cyclomatic Complexity	Risk Evaluation
1 – 10	A simple program, without much risk
11 – 20	More complex, moderate risk
21 – 50	Complex, high risk program
Greater than 50	Unstable program (very high risk)

Table 7.1 Cyclomatic Complexity Thresholds “(WEB_12)”.

Cyclomatic Complexity in fact is the classical graph theory cyclomatic number, indicating the number of regions in a graph. As applied to software, it is the number of linearly independent paths that comprise a program. In other words, it represents the number of binary decisions in a program. A low cyclomatic complexity contributes to a program’s understandability, and indicates it’s amenable to modification at a lower risk than a more complex program. It is also a strong indicator to a program’s testability. However, the fact that a code module has high cyclomatic complexity does not, by itself, mean that it represents excess risk, that it can, or should be redesigned to make it simpler; more must be known about specific application. The Ticket Seller Application will be analyzed in terms of test code coverage and cyclomatic complexity in this study.

7.4.1 Setting up the Environment

Cobertura does not provide an Eclipse plug-in that can be integrated into the IDE platform. However, it provides a couple of Ant tasks that can make the build – test – retrospect – deploy cycle in an automated way. Cobertura works by inserting instrumentation instructions directly into the compiled Java classes. When these instructions are encountered by the Java Virtual Machine, the inserted code increments various counters so that it is possible to tell which instructions have been encountered and which have not. An Ant task is responsible for creating the instrumented versions of the classes. Once the classes have been instrumented by Cobertura, the application can be tested normally by JUnit. The instrumented classes found in the classpath will be loaded before the original ones do. The classes to be instrumented can be specified by using Regular Expressions in the Ant task.

7.4.2 Running the Tests

With an appropriate Ant configuration, running all the tests and getting the product metrics analysis reports becomes a straightforward and simple task. The process can easily be monitored within Ant logs directed to console or to a file as seen below;

```
Buildfile: D:\EclipseWorkspaces\Yemekhane\YemekhaneDomain\build.xml
clean:
   [delete] Deleting directory D:\EclipseWorkspaces\Yemekhane\YemekhaneDomain\bin
   [delete] Deleting directory D:\EclipseWorkspaces\Yemekhane\YemekhaneDomain\coverage
   [delete] Deleting directory D:\EclipseWorkspaces\Yemekhane\YemekhaneDomain\instrument
   [delete] Deleting: D:\EclipseWorkspaces\Yemekhane\YemekhaneDomain\cobertura.ser
init:
    [mkdir] Created dir: D:\EclipseWorkspaces\Yemekhane\YemekhaneDomain\bin
    [mkdir] Created dir: D:\EclipseWorkspaces\Yemekhane\YemekhaneDomain\coverage
    [mkdir] Created dir: D:\EclipseWorkspaces\Yemekhane\YemekhaneDomain\instrument
build:
     [echo] YemekhaneDomain: D:\EclipseWorkspaces\Yemekhane\YemekhaneDomain\build.xml
     [copy] Copying 65 files to D:\EclipseWorkspaces\Yemekhane\YemekhaneDomain\bin
    [javac] Compiling 57 source files to D:\EclipseWorkspaces\Yemekhane\YemekhaneDomain\bin
    [javac] Note:
D:\EclipseWorkspaces\Yemekhane\YemekhaneDomain\src\edu\iyte\yemekhane\util\DateUtils.java uses
or overrides a deprecated API.
    [javac] Note: Recompile with -Xlint:deprecation for details.
    [javac] Note: Some input files use unchecked or unsafe operations.
    [javac] Note: Recompile with -Xlint:unchecked for details.
instrument:
[cobertura-instrument] Cobertura null - GNU GPL License (NO WARRANTY) - See COPYRIGHT file
[cobertura-instrument] Instrumenting 19 classes to
D:\EclipseWorkspaces\Yemekhane\YemekhaneDomain\instrument
[cobertura-instrument] Instrument time: 266ms
AllTests:
    [junit] Running edu.iyte.yemekhane.AllTests
    [junit] Tests run: 51, Failures: 0, Errors: 0, Time elapsed: 2,984 sec
coverage:
[cobertura-report] Cobertura null - GNU GPL License (NO WARRANTY) - See COPYRIGHT file
[cobertura-report] Report time: 719ms
BUILD SUCCESSFUL
Total time: 9 seconds
```

7.4.3 Analyzing the Metrics

The results of code metrics produced are simple and sef explaining, as can be seen in **Figure 7.1**

The analysis shows that branch coverage of the tests are all complete, with 100%. However, line coverage falls down to 80% in package **edu.iyte.yemekhane.persistence.** So, inspecting this package by itself first is a reaosonable way to go;

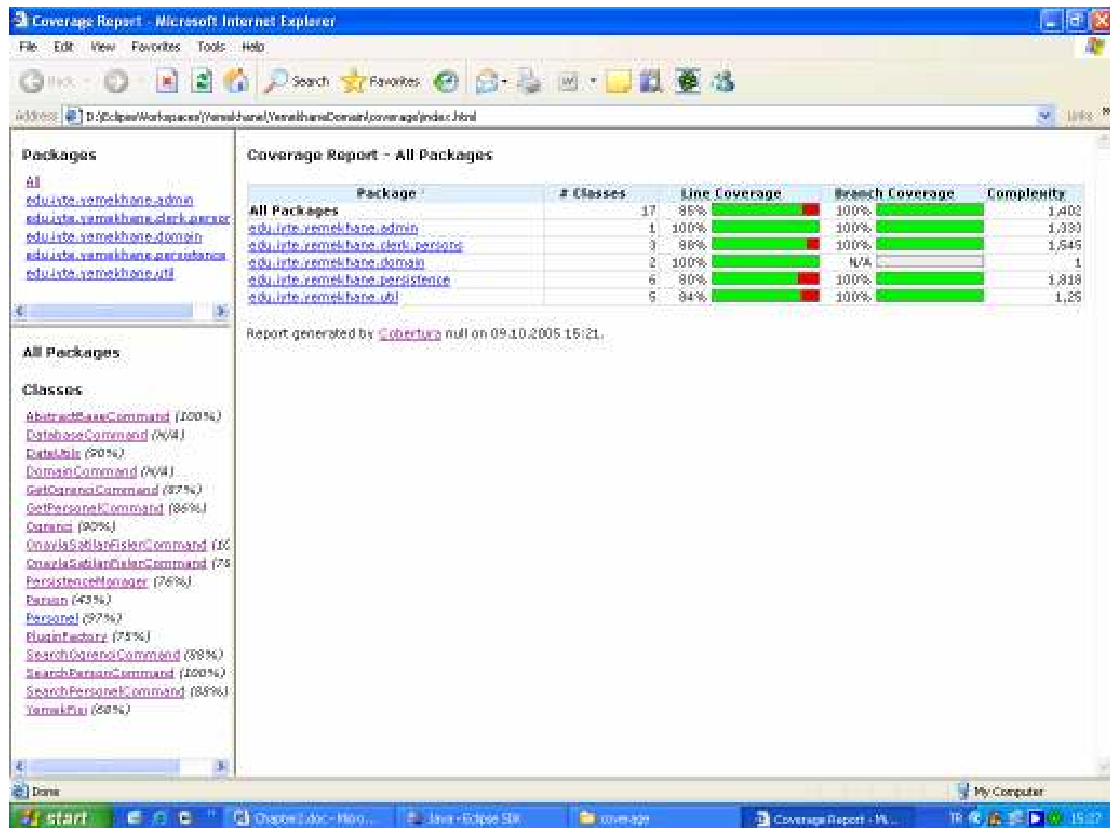

Figure 7.1 First Test Coverage and Complexity Analysis Report

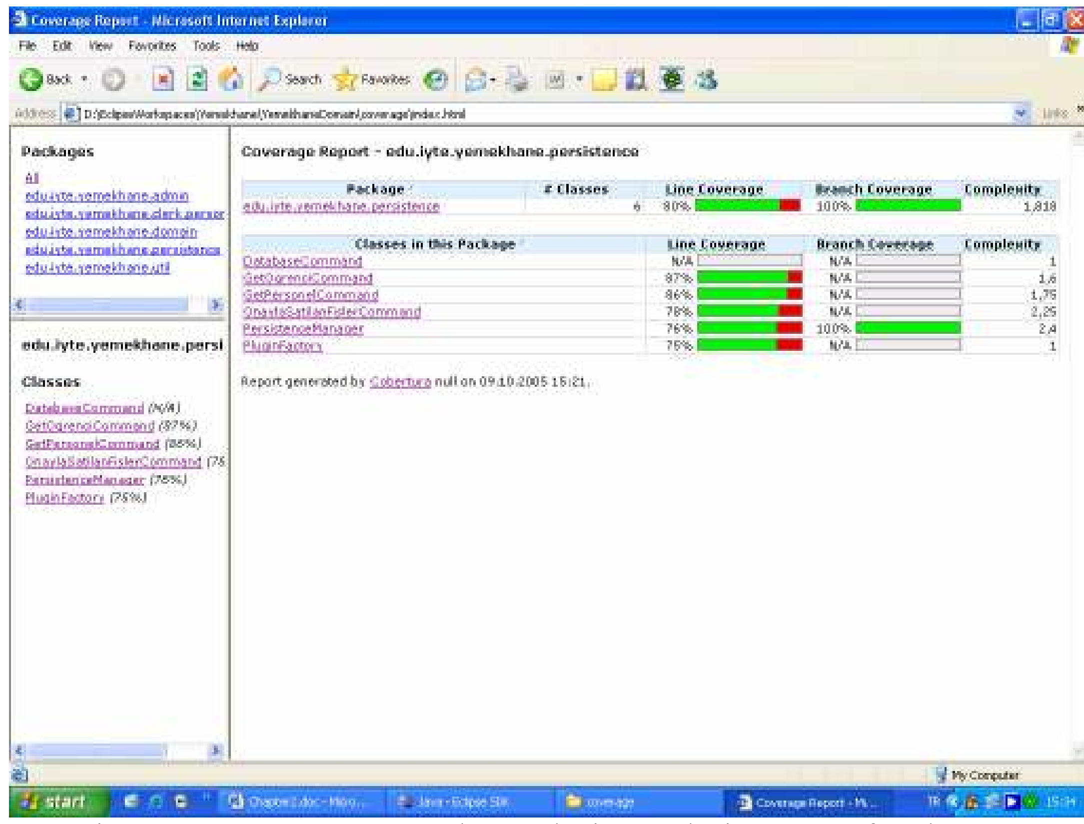

Figure 7.2 Test Coverage And Complexity Analysis Report of package edu.iyte.yemekhane.persistence

Here, the lowest test coverage seems to appear in the class PluginFactory, with 75%. So, a closer inspection to it would give an idea about the problem;

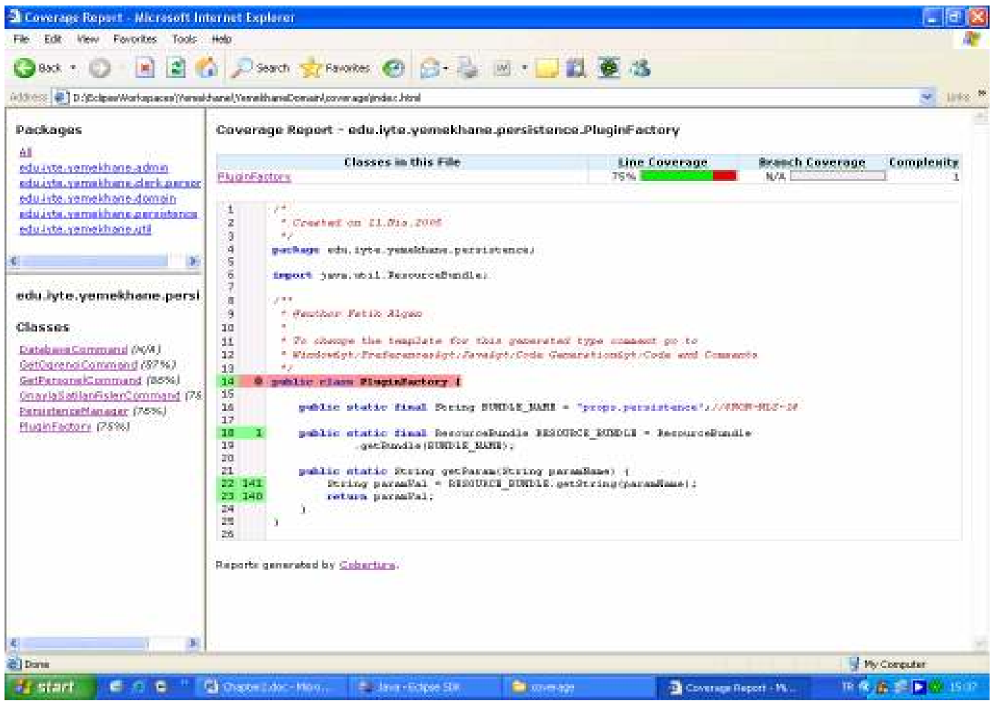

Figure 7.3 The uncovered parts of class PluginFactory, marked with RedLine.

Inspecting the class PluginFactory reveals that the uncovered part is the class definition itself. This is probably because of a small bug in Cobertura itself, maybe due to being all members of the class static, since such a line should not be reported as a code coverage problem.

Further inspection of the tool, reveals that it also reports some classes that are not being tested according to the "**too simple to break**" rule, such as a class with simple data fields which only has simple accessor methods, as seen below.

However, the tool gives valuable clues that may point to potential high coupling and low cohesion points in the code structure, or the parts of the code that the tests neglect to verify. In **Figure 7.5,** the Analysis report of the class PersistenceManager is given. It seems that the tests fail short to verify whether the methods getConnection(), getDataSource() and commitWork() throw the expected kind of exceptions in the case of a system failure. This problem seems to arise from an incomplete test harness, because, the mock versions of a DataSource or Connection class could easily be plugged in that would throw the desired exceptions when they are sent a message. If stub versions of these classes were impossible to be plugged in, this would indicate a potential source of high coupling and low cohesion in the codebase.

Figure 7.4 The Uncovered Code parts that belongs to a Simple Data Class

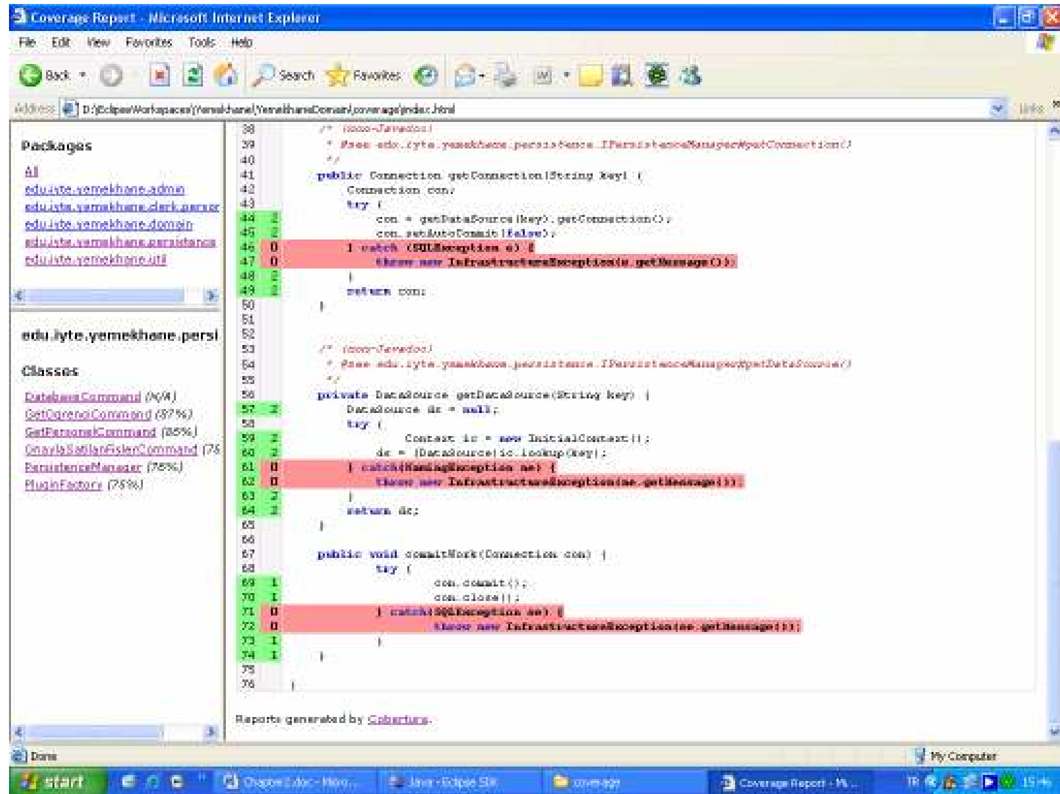

Figure 7.5 Uncovered Code Belonging to class PersistenceManager

The cyclomatic complexity analysis of the TicketSeller application has a very low complexity factor, indicating a low risk at change management as listed in **Table 7.2**

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	17	85%	100%	1,402
edu.iyte.yemekhane.admin	1	100%	100%	1,333
edu.iyte.yemekhane.clerk.persons	3	88%	100%	1,545
edu.iyte.yemekhane.domain	2	100%	N/A	1
edu.iyte.yemekhane.persistence	6	80%	100%	1,818
edu.iyte.yemekhane.util	5	84%	100%	1,25

Table 7.2 Complexity and Coverage analysis of each package

7.5 Summary

Metrics are defined for measuring the development process and the produced software. Software metrics can hint at possible anomalies in a negative or positive sense.

Process metrics aim to evaluate the development process itself whereas the product metrics focus on the evaluate the result of the process, i.e. the product itself.
Test Driven Development by itself does not propose any specific process metrics, however, the development process it is involved in might, just as Extreme Programming defines process metrics based on completed user stories or task cards.

The product metrics can be gathered by analyzing the produced software and can generally be analyzed in complete automation. These metrics can indicate the quality of an application in terms of complexity, coverage and quality. These attributes of a software depends on many parameters such as dependency analysis, lines of code or cyclomatic complexity. However, the requirements, functional and nonfunctional requirements of each application is specific in its own context to a certain degree. Thus, the threshold values of product metrics for each application should be interpreted in its own domain context.

CHAPTER 8

CONCLUSION

Software development has always been a big challenge. As the IT industry grew enormously within the last twenty years, so did the size and complexity of software products. Many development methodologies, processes and tools hace been proposed and evaluated along the way by the academicians, industry or developers. However, the key challenge, managing complexity in software development is still a hot area of research. Reminding a cynical remark of Frederick Brooks, “software developement is stil in its stone age”, as an engieering discipline.

Software verification and validation is an essential part of software development process, and a key factor of improving software product quality. However, in order to keep the testing process in discipline and maintainable as the product sizes and complexity grows, the testing process itself needs to be automated.

During the last four years, I have worked on a number of Java and J2EE based enterprise application development projects of different sizes. One of the key challenges I have faced in those projects was successfully applying a disciplined programming technique and development process to increase the quality of the software while keeping the projects on schedule. Those were the times I first started to read some articles about Extreme Programming. Like many professionals working in software business, it attracted my attention with its human centric, simple and clever approach to solve the problems I faced.

The most interesting and new practice of extreme programming was test driven development. It was a programming method that benefited from automated tests, but was different than the concept of conventional software test automation. In Test Deriven Development, the testing activity itself is an integral part of the programming process, and is considered a part of production code and documentation. In test driven development, a number of small tests for the needed functionality was written first, then the code was written to pass those tests successfully.

The idea behind TDD is not replacing the QA department’s defect detecting role with programmer testing, contrary to the phrase **“test”**. TDD aims to reach a high quality software product starting from requirements definition phase by dividing the requirements to fine grained pieces and expressing the as test cases to achieving high quality, working end product.

The TDD process itself is human centric, contrary to the techniques that were proposed so far, by putting the programming itself as the centric task of software development. The obligation to run the tests in isolation from each other, forces the programmer to write decoupled, high cohesive code. Continious refactoring eliminates the duplication and increases developer confidence.

After reading a few articles on TDD, I investigated further on TDD while trying automated tests in the projects I have worked for the first time. However, these were not unit tests. The goal was then writing performance tests for an application that was being developed for the university and end to end tests were the suitable tool for that purpose. After more investigation it turned out that applying test driven development was harder than it seemed at first. Writing unit tests were not as direct and smooth as writing end to

end integration tests. It required a deep understanding of object oriented concepts, design patterns and refactoring experience. Without continious refactoring, abstraction, and seperation, writing testable code was not possible. The countless number of frameworks, application server containers, databases, external applications and services were not improving the situation at all either.

The main goal of this work is to prove and show that Test Driven Development can be applied as a pragmatic but robust and measurable programming technique in developing software. While putting forth the statement, comprehensive techniques on best practices and patterns to use while applying unit testing is researched. Trying to provide a theoretical background, the solutions presented in the study also aims to be pragmatic, simple and cost effective to apply. In cases where a solution that does not fall short in satisfying those criteria, end to and tests are suggested.

At nearly the same time with this study, I took part in a new software project as the team leader and decided to apply test driven development under the light of my research on the subject. I have found the chance to apply many of the practices presented in this study in the project. Although my initial planning was to include the mentioned project as an application of the techniques presented in this study, the direction and scope of the project followed a different path. The project is still under development and is planned to go into production within a few months later, a reason that makes me reluctant to put forth my experiences regarding test driven development without seeing its end results, the size of the software also grew too big to be included in such a study. Consequently, I have decided to put a smaller application employing test driven development into this study. While being small in scale, this application is also a real and complete application that is about to go into production within a few weeks at the university I am currently working.

The findings and solutions presented in this study could have further been enriched by additional topics, had there been no time constraints. A section on management and configuration of the production and test code was among my plans when I started to work on this thesis. While not seeming a technical subject, test configuration and maintanence is maybe one of the key factors determining success of TDD in practice. Another section on integration and performance tests could also have been added. However, this is a subject more related to software testing, not test driven development itself. TDD limits the testing activity to unit testing, whose application and goal is quite different from the conventional meaning of **“software testing”**.

While claiming that TDD improves the software development process and helps the developer to produce high quality design and code, this proposition would not go further than being only a claim, unless proven by concrete evaluation metrics. The last part of the study is based on the some metrics that can be used to evaluate how Test Driven Development contributes to software development practice and how software assessment can be accomplished in agile practices.

BIBLIOGRAPHY

Allen H., Dutoit, Bruegge, B., 1998, Communication Metrics for Software Development IEEE Transactions on Software Engineering, 24(8), August

Ambler, W. S., 2002, Agile Modeling (Wiley)

Bassin, K., Biyani, S., Santhanam, P., 2002, Metrics to Evaluate Vendor-Developed Software Based on Test Case Execution Results, IBM Systems Journal, 41(1):13–30.

Beck, K., 1999, Extreme Programming Explained (Addison-Wesley) , p 30, 46

Beck K., 2003, Test Driven Development By Example (Addison-Wesley) , p 50

Binder, R. V., 1999, Testing Object-Oriented Systems: Models, Patterns, and Tools(Addison-Wesley)

Boehm, B., Turner, R., 2003, Balancing Agility and Discipline – A Guide for the Perplexed (Addison-Wesley)

Chidamber, S. R., Kemerer, C.F., 1994, A metrics Suite for Object-Oriented Design: IEEE Transactions on Software Engineering, 20(6).

Cockburn, A., 2002, Agile Software Development (Addison-Wesley) , p 44

Diomidis, S., 2003, Code Reading(Addison-Wesley)

Fenton N., 1994, Software Measurement: A Necessary Scientific Basis: IEEE Transactions on Software Engineering, 20(3), March.

Fenton, N. E., Pfleeger, S. L., 1996, Software Metrics: A Rigorous and Practical Approach.(International Thomson Publishing)

Fewster, M., Graham, D., 1999, Software Test Automation (Addison-Wesley), p9

Fowler M., 1999, Refactoring: Improving the Design of Existing Code (Addison-Wesley)

Fowler, M., 2002, Patterns Of Enterprise Application Architecture (Addison-Wesley), p 110

Gamma, Helm, Johnson, Vlissides, 1995, Design Patterns - Elements Of Reusable Object Oriented Software (Addison-Wesley)

Gerald, M. W., 1993, Quality Software Management – First Order Measurement - Volume 2 (Dorset House).

Highsmith, J., Cockburn A., 2001, Agile Software Development: The Business Of Innovation, IEEE Computer, Sept.

Johnson, P. M., Kou, H., Agustin, J., Chan, C., Moore C., Miglani, J., Zhen, S., Doane, W. E. J., 2003, Beyond the Personal Software Process: Metrics Collection and Analysis for the Differently Disciplined. In Proc. Intl. Conf. on Software Engineering (ICSE), Portland, OR, May.

Jones, C., 2000, Software Assessments, Benchmarks, and Best Practices (Addison-Wesley)

Ken, Orr., 2002, CMM Versus Agile Development Religious Wars and Software Development. Agile Project Management, 3(7)

Kerievsky J., 2004, Refactoring to Patterns(Addison-Wesley).

Martin, R. C., 2003, Agile Software Development - Principles, Patterns, and Pracitces (Prentice-Hall)

Maximilien, E. M., Williams, L., 2003, Assessing Test-Driven Development at IBM: In Proc. Intl. Conf. on Software Engineering (ICSE), May.

McConnell, S., 1993, Code Complete: A Practical Handbook of Software Construction (Microsoft Press)

Paulk, M.C., Weber, C.V., Curtis, B., 1995, The Capability Maturity Model, Guidelines for Improving the Software Process(Addison-Wesley).

Sommerville, I., 1996, Software Engineering (Addison-Wesley) , p 446, 466, 471

ςολοκη, E., 1990, ςΕΣΟΦΤ, Automated Testing – WHY AND HOW, INTEREX Conference, Boston, MA, USA

WEB_1, 2005, Ian Rankine, QES Inc, Introduction To SoftwareTesting, 19/04/2005, www.qestest.com/principl.htm

WEB_2, 2005, Automated Testing Specialists Web Page, 22/04/2005, http://www.sqa-test.com/toolpage.html

WEB_3, 2005, Agile Software Development Manifesto, 22/05/2005, http://agilemanifesto.org

WEB_4, 2005, HighSmith J, What Is Agile Software Development, 22/05/2005, www.adaptivesd.com/articles/cross_oct02.pdf

WEB_5, 2005, Kumar Ravi, Lean Software Development, The Project Perfect White Paper Collection, 05/05/2005, www.projectperfect.com.au/downloads/inf_lean_debelopment.pdf

WEB_6, 2005, Abrahamsson, Salo, Ronkainen, Warsta, Agile Software Development Methods, VIT publications, ESPOO 2002, 07/06/2005, www.inf.vtt.fi/pdf/publications/2002/P478.pdf

WEB_7, 2005, Beck Kent, Simple Smalltalk Testing With Patterns, 24/05/2005, ftp://ic.net/users/jeffries/TestingFramework

WEB_8, 2005, Java Blueprints Data Access Object Pattern, 22/04/2005, http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

WEB_9, 2005, Java Blueprints Model View Controller Pattern, 06/06/2005, http://java.sun.com/blueprints/patterns/MVC-detailed.html

WEB_10, 2005, Plain Old Java Object(POJO) Pattern, 12/05/2005, http://www.martinfowler.com/bliki/POJO.html

WEB_11, 2005, Automated Support for Process Assessment in Test-Driven Development, 09/07/2005, Christian Wege, Tubingen 2004

WEB_12, 2005, Sample treshold values for Cyclomatic Complexity, 12/06/2005, http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html

WEB_13, 2005, SUnit Home Page, 16/04/2005, http://sunit.sourceforge.net

WEB_14, 2005, JUnit Home Page, 16/04/2005, http://www.junit.org

WEB_15, 2005, Eclipse Project Home Page, 16/04/2005, www.eclipse.org

WEB_16, 2005 NetBeans Project Home Page, 22/05/2005, www.netbeans.org

WEB_17, 2005, Oracle JDeveloper IDE Page, 05/05/2005, http://www.oracle.com/technology/software/products/jdev/index.html

WEB_18, 2005, GS Base Project Home Page, 12/04/2005, http://gsbase.sourceforge.net

WEB_19, 2005, Java 2 Enterprise Edition Technology Home Page, 22/07/2005, http://java.sun.com/j2ee/index.jsp

WEB_20, 2005, Java Servlet Technology Home Page, 04/05/2005, http://java.sun.com/products/servlet/index.jsp

WEB_21, 2005, Java Server Pages Technology Home Page, 05/05/2005, http://java.sun.com/products/jsp/index.jsp

WEB_22, 2005, Enterprise Java Beans Technology Home Page, 07/05/2005, http://java.sun.com/products/ejb/index.jsp

WEB_23, 2005, XML Path Language(XPath) Home Page, 15/06/2005, http://www.w3.org/TR/xpath20

WEB_24, 2005, Xalan-Java XSLT Processor Home Page, 08/05/2005, http://xml.apache.org/xalan-j/

WEB_25, 2005, Jaxen Java XPath Engine Home Page, 09/05/2005, http://jaxen.org/

WEB_26, 2005, XMLUnit Project Home Page, 12/05/2005, http://xmlunit.sourceforge.net/

WEB_27, 2005, Hibernate Framework Home Page, 22/06/2005, http://www.hibernate.org

WEB_27, 2005, Java RMI over IIOP Technology Home Page, 25/05/2005, http://java.sun.com/products/rmi-iiop

WEB_28, 2005, Apache Cactus Project Home Page, 13/05/2005, http://jakarta.apache.org/cactus/index.html

WEB_29, 2005, MockEJB Framework Home Page, 06/07/2005, http://www.mockejb.org/

WEB_30, 2005, EasyMock Project Home Page, 23/05/2005, http://www.easymock.org

WEB_31, 2005, ServletUnit Project Home Page, 17/08/2005, http://httpunit.sourceforge.net/doc/servletunit-intro.html