The Claims about Test Driven Development

Mingsheng Bai

Lyngby/Denmark 2007

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

In the traditional software development process, unit- and functional tests are written after the code is implemented. However, recently agile software development methods were introduced which also change traditional testing practice. Test driven development (TDD) is a practice of eXtreme Programming (XP) where unit- and functional tests drive the development of the code. This means that the tests are written before the actual code that is going to be tested. It is claimed, among others, that TDD produces better code quality.

The goal of this thesis is to collect all the claims. Evaluation is done in two steps. The first step is studying the literature for supporting or contradictory evidences. The second step is implementing two case studies: a GUI based cinema reservation system and a GUI based shop stock management system. One case study is done using a traditional software process, where the tests are written after the implementation, and the second case study is done using XP and TDD. The results of the case studies are then compared with the results from the literature.

Preface

This thesis was prepared at Informatics and Mathematical Modelling, the Technical University of Denmark in partial fulfilment of the requirements for acquiring the master degree in Computer System Engineering. This thesis was supervised by Hubert Baumeister at IMM.

This thesis introduces the eXtreme Programming and Test Driven Development. The main focus is on collecting claims about Test Driven Development, and evaluated those claims.

I would like to thank my supervisor, the professor Hubert Baumeister. Thanks for Hubert's guidance, correction and advice throughout the whole project.

Copenhagen, Denmark, 2007

Mingsheng Bai

Contents

1. Introduction

Test Driven Development is a new practice of t eXtreme Programming (XP). Some studies or researches have been executed with the aim of understanding or comparing it with a traditional practice. Some claims concerning TDD which are positive or negative were emerged. In this thesis I will collect some claims about TDD, and the evaluation on these claims will be done.

1.1 Background

Test-Driven Development has been invented by Kent Beck and is a development practice which is part of a software development methodology called eXtreme Programming (XP) [1]. TDD began to receive publicity in the early twenty-first century as an aspect of Extreme Programming [28].

TDD is based on the idea to create tests for the program before you develop the program code. This is the opposite of what is usual in current software development methodologies. The availability of tests before actual development ensures rapid feedback after any change. Practitioners emphasize that test-driven development is a method of designing software, not merely a method of testing [26].

As a member of the eXtreme Programming best practices, TDD is most often associated with agile software development process [6].

The TDD practice starts with thoughts on how to test the required functionality. After writing automated test cases that generally will not even compile, the programmers write implementation code to pass these test cases [25]. It follows steps like: write a test case quickly, run the test case to see it failed, write a little production code, run the test case and see it succeed, refactor the code. Such kind of iterations will go through all the user stories.

1.2 Thesis scope

In this thesis I will collect some claims concerning Test Driven Development, which are both supporting TDD and non-supporting TDD. Evaluation of those claims will be done in two steps. The first step, evaluation is to be done by literature study. And the second step, evaluation is to be done by experiment. Due to some claims concern the comparison between XP with TDD and waterfall model, the experiment is designed to develop two systems by using XP with TDD and waterfall model respectively. The

two systems to be developed in the experiment are cinema reservation system and shop stock management system. The cinema reservation system will be developed by using the waterfall model. The shop stock management system will be developed using XP with TDD. Finally, this thesis will evaluate those claims by the thesis finding and the combination of literature study and experiment result.

1.3 Outline

This thesis consists of 7 main parts. The chapter 1 is the introduction part, gives the introduction for the background of the TDD and the scope of the thesis. The chapter 2 will introduce agile software development. The chapter 3 will introduce eXtreme Programming and the benefits & shortcomings. The chapter 4 will introduce the Test Driven Development by details and the claims are collected from literature papers. The chapter 5 is the literature research part, contains three sub sections. The first section introduces 2 of the researched papers for this thesis. The second section introduces the evaluation criteria for the evaluation. The third section gives the evaluation on claims by literature research. The chapter 6 is the experiments part, contains three main parts. The first part documents the experiment's description. The second part documents the experiment's process. The third part gives the evaluation on those claims based on the experiments' result. Finally, the chapter 7 is the conclusion part, gives the achievement of this thesis, final evaluation on the claims and the thoughts & further work.

2. Agile software development

Agile software development is a conceptual framework for software engineering that promotes development iterations throughout the life-cycle of the project. There are many agile development methods; most minimize risk by developing software in short amounts of time. Software developed during one unit of time is referred to as an iteration, which may last from one to four weeks [10]. Iteration consists of the whole software develop process: requirement analysis, design, implementation and test. Iteration may not achieve the full functionalities of the software but the goal is to have an available release (without bugs) at the end of each iteration. At the end of each iteration, the team re-evaluates project priorities.

This chapter will introduce the principles of agile software development and comparison to other methods.

2.1 The principle of agile method-The Agile Manifest

In 2001, 17 prominent[1] figures in the field of agile development came together at the Snowbird ski resort in Utah to discuss ways of creating software in a lighter, faster, more people-centric way. They created the Agile Manifesto, widely regarded as the canonical definition of agile development, and accompanying agile principles [7].
Some of the principles behind the Agile Manifesto are [8]:

- Customer satisfaction by rapid, continuous delivery of useful software
- Working software is delivered frequently (weeks rather than months)
- Working software is the principal measure of progress
- Even late changes in requirements are welcomed
- Close, daily cooperation between business people and developers
- Face-to-face conversation is the best form of communication
- Projects are built around motivated individuals, who should be trusted
- Continuous attention to technical excellence and good design
- Simplicity
- Self-organizing teams
- Regular adaptation to changing circumstances [8]

[1] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas

2.2 Comparison with other method

Agile methods are sometimes characterized as being opposite to the plan-driven or disciplined methodologies. This distinction is misleading, as it implies agile methods are unplanned or undisciplined. A more accurate distinction is to say that methods exist on a continuum from "adaptive" to "predictive". Agile methods exist on the "adaptive" side of this continuum [9].

Adaptive methods focus on adapting quickly to changing realities. When the needs of a project change, an adaptive team changes as well. An adaptive team will have difficulty describing exactly what will happen in the future [7].

Predictive methods, in contrast, focus on planning the future in detail. A predictive team can report exactly what features and tasks are planned for the entire length of the development process. Predictive teams have difficulty changing direction [7].

This section will compare the agile software development with an iterative and Incremental development and a waterfall model.

Compare with iterative and Incremental development

Iterative and Incremental development is a cyclical software development process developed in response to the weaknesses of the waterfall model. It is an essential part of the Rational Unified Process, the Dynamic Systems Development Method, Extreme Programming and generally the agile software development frameworks [11].

The iterative development and Agile development have the same trait, as they emphasis on building software release in short time period. However, Agile development differs from other development models as in this model time periods are measured in weeks rather than months and work is performed in a highly collaborative manner, and most agile methods also differ by treating their time period as a strict timebox [7].

Compare with waterfall model

The waterfall model is a sequential software development model (a process for the creation of software) in which development is seen as flowing steadily downwards (like a waterfall) through the phases of requirements analysis, design, implementation, testing (validation), integration, and maintenance [12].

As of 2004, the waterfall model is still in common use [13]. The waterfall model is a typical predictive method, which predicts the future and stepping through requirement analysis, design, coding and test in a pre-planned sequence. The progress is general measured by the requirement specification, design documents, test strategy and etc.

The main problem of the waterfall model is the inflexible nature of the division of a project into separate stages, so that commitments are made early on, and it is difficult to react to changes in requirements. Iterations are expensive. This means that the waterfall model is likely to be unsuitable if requirements are not well understood or are likely to change radically in the course of the project [14].

The agile methods produce developed feature in short time period, and phases on obtaining small piece of function to deliver business value early. The agile methods don’t fear the requirement changing. However, some agile teams use the waterfall model on a small scale, repeating the entire waterfall cycle each iteration [15].

3. Extreme Programming

Extreme Programming (XP) is one of several Agile Software Development methodologies in Software Engineering. XP is created based on observations on what made software development faster and what made it slower [16]. Despite the many arguments for and against this kind of methodology, Extreme programming has been embraced by the commercial sector during the last ten years [17].

XP emphasizes on close collaboration between the developer team and the customer through face-to-face communication, frequent delivery, self-organizing teams, and rapid response to changes in requirements [16]. XP has advantage in adapting in changing user requirement at any point of project lifecycle. There is no full prescribed activity sequence specified in XP. According to the adaptive approaches, implementation of the projected product starts quickly leading to an incremental delivery of the product [16]. XP initially start with getting a rough requirement involve the importance of the system. And the overall very general architecture of the system and implementation will be built up. With getting more requirements of the system, the full architecture of the system with full functionalities accomplished. The project will complete after several iterations. In predictive system development methods the requirements for the system are determined at the beginning of the development project and often fixed from that point on. This obviously is different from the predictive approach.

The table 3.1 shows the substantial difference between the predictive approach and the XP.

Methodology	XP	Predictive
Iteration	Short(weeks)	Long(months)
Design	During process	Upfront
Test	Test first	Test last
Costumer involvement	During whole process	At initial and final phases

Table 3.1 the difference between predictive approach and XP

The XP carried out based on the basic values, principles and practices. In the section 2.1, the values, principles and practice will be introduced.

3.1 Values, Principles and Practice

XP is built up based on several values, principles and practice. And principles have to be in accordance with values, practice have to be in accordance with principles. The

values are the central part of XP. XP could only be used under agreement of values. There are five values were introduced at present.

Communication
XP promotes communication between your team and your project stakeholders as well as between developers on your team. To achieve this, Extreme Programming favors simple designs, common metaphors, collaboration of users and programmers, frequent verbal communication, and feedback.

Simplicity
XP encourages starting with the simplest solution and refactoring to better ones [18]. This more emphasize on designing or coding for the needs of today instead of tomorrow even future. Due to the requirements possibly changed any time, spending resources on something may not be needed is unwise.

Feedback
With the XP, the feedback should concern these three aspects:

- **The feedback from system**: Write a unit test, the programmer can directly get the feedback from system after implementing changed.
- **The feedback from costumer**: The costumer or the end user should communicate with developers periodically to get the up-to-date requirement.
- **The feedback from team**: After the requirement changed, the team member should communicate each other to get the news.

Courage
Developers should dare to face anything, includes throw the source code away.

Respect (Humility)
People under a project team should care about each other and about the project.

The principles that form the basis of XP are based on the values just described and are intended to foster decisions in a system development project. The principles are intended to be more concrete than the values and more easily translated to guidance in a practical situation [25].

Assuming simplicity is about treating every problem as if its solution were "extremely simple". Traditional system development methods say to plan for the future and to code for reusability. Extreme programming rejects these ideas.

The advocates of Extreme Programming say that making big changes all at once does not work. Extreme Programming applies incremental changes: for example, a system might have small releases every three weeks. By making many little steps the customer has more control over the development process and the system that is being developed.

The principle of embracing change is about not working against changes but embracing them. For instance, if at one of the iterative meetings it appears that the customer's requirements have changed dramatically, programmers are to embrace this and plan the new requirements for the next iteration.

There are 12 practices categorized into 4 areas, in the Extreme Programming, derived from the best practices of software engineering. They were shown at table 3.2:

Area	Fine scale feedback	Continuous process	Shared understanding	Programmer wefare
Practice	Pair Programming Planning Game TDD Whole Team	Continuous Integration Design Improvement Small Releases	Coding Standard CollectiveCodeOwnership Simple Design System Metaphor	Sustainable Pace

Table 3.2 the practices of Extreme Programming

3.2 Benefits

eXtreme Programming do has benefits of using it. This thesis will introduce the benefit of using XP from three aspects, which are from developers, customers and management [56].

For **Developers**, XP allows developers to focus on coding and avoid needless paperwork and meetings. It provides a more social atmosphere, more opportunities to learn new skills.

For the **Customer**, XP creates working software faster, and that software tends to have very few defects. It allows customer to change your mind whenever you need to, with minimal cost and almost no complaining from the developers. XP do has strong adaptability of changing requirement.

For **Management**, XP delivers working software for less money, and the software is more likely to do what the end users actually want.

3.3 Limitation

The exact limits of XP aren't clear yet. But there are some controversial aspects.

Unstable Requirements: Proponents of Extreme Programming claim that by having the on-site customer request changes informally, the process becomes flexible, and

saves the cost of formal overhead. Critics of XP claim this can lead to costly rework and project scope creep beyond what was previously agreed or funded [18].

User Conflicts: Change control boards are a sign that there are potential conflicts in project objectives and constraints between multiple users. XP's expedited methodology is somewhat dependent on programmers being able to assume a unified client viewpoint so the programmer can concentrate on coding rather than documentation of compromise objectives and constraints. This also applies when multiple programming organizations are involved, particularly organizations which compete for shares of projects [18].

XP might be limited in an environment where a long time is needed to gain feedback. For example, if the system takes 24 hours to compile and link, developer will have a hard time integrating, building, and testing several times a day [56].

4. Test Driven Development

Studies indicate that testing accounts for at least 50% of the total development time [20], [21]. One reason for this is that the verification activities late in development projects tend to be loaded with defects that could have been prevented or at least removed earlier (when they are cheaper to find and remove [19], [20], and [23]). When many defects remain to be found late in a project, schedules are delayed and the verification lead-time increases [22]. Test Driven Development was popularized as a practice of defects-reduced, defect-detection-early and high flexibility. It (TDD) has emerged as a novel software development approach that involves writing automated unit tests in an iterative Test-First manner. When applying TDD, a software developer writes one small automated unit test [24]. The developer then writes just enough code to make the test pass. After possible refractory, the cycle then quickly repeats with the developer writing another test and code to satisfy the test. This chapter will introduce the Test Driven Development in three subsections. The first section is to introduce the Test Driven Development as a software development practice, the second section is to introduce the benefit of the TDD and the last one is to give the shortcoming of using TDD.

4.1 TDD, a software development practice

Test Driven Development (TDD), a software development practice used sporadically for decades [36, 37], has gained added visibility recently as a practice of Extreme Programming (XP) [1]. The practice involves the implementation of a system starting from the unit test cases of an object. Writing test cases and implementing that object or object methods, triggers the need for other objects/methods. An important rule in TDD is: ‘If you can’t write a test for what you are about to code, then you shouldn’t even be thinking about coding’ [38].

Test-driven development should be used combine with Unit Testing. TDD requires that an automated unit test, defining requirements of the code, is written before each aspect of the code itself. These tests contain assertions that are either true or false. Running the tests gives rapid confirmation of correct behavior as the code evolves and is refactored. Testing frameworks based on the xUnit concept provide a mechanism for creating and running sets of automated test cases [26].

The followed section will introduce the formal development cycle of TDD. And R. Martin defined “three laws of using TDD”. The three-laws help the new practitioners to use TDD. The second section will introduce the three laws of using the TDD.

4.1.1 Test-Driven Development Cycle

Kent Beck proposed a sequence of using TDD in his book. The TDD process start with add a test quickly, and run all the tests and see the new one fail, make a little change of the code, then run the automated tests and see them succeed, at last refactor to remove duplication. The each step of TDD will be introduced as follow [52]:

Quickly add a test
In test-driven development, each new feature begins with writing a test. This test must inevitably fail because it is written before the feature has been implemented. In order to write a test, the developer must understand the specification and the requirements of the feature clearly. This may be accomplished through use cases and user stories to cover the requirements and exception conditions. This could also imply an invariant, or modification of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes you focus on the requirements before writing the code, a subtle but important difference.

Run all tests and see the new one fail
This validates that the test harness is working correctly and that the new test does not mistakenly pass without requiring any new code.
The new test should also fail for the expected reason. This step tests the test itself, in the negative. A "negative test" is something familiar to testers, to make sure a feature fails when it should fail (e.g. bad input data). It typically follows or is "paired" with one or more "positive tests" that make sure things work as expected (e.g. good input data). ("Make sure it works, and then change one thing to make it break and make sure it breaks.") The entire suite of unit tests act to serve this need, cross-checking each other to make sure "negative tests" fail for the expected reasons.
This technique avoids the syndrome of writing tests that always pass, and therefore aren't worth much. Running the new test to see it fail the first time is a vital "sanity check".

Make a little change
The next step is to write some code that will cause the test to pass. The new code written at this stage will not be perfect and may, for example, pass the test in an inelegant way. That is acceptable because later steps will improve and hone it. It is important that the code written is only designed to pass the test; no further (and therefore untested) functionality should be predicted and 'allowed for' at any stage.

Run the automated tests and see them succeed
If all test cases now pass, the programmer can be confident that the code meets all the tested requirements. This is a good point from which to begin the final step of the cycle.

Refactor to remove duplication

Now the code can be cleaned up as necessary. By re-running the test cases, the developer can be confident that refactoring is not damaging any existing functionality. The concept of removing duplication is an important aspect of any software design. In this case, however, it also applies to removing any duplication between the test code and the production code — for example magic numbers or strings that were repeated in both, in order to make the test pass in step 3.

Repeat
Starting with another new test, the cycle is then repeated to push forward the functionality. The size of the steps can be as small as the developer likes, or get larger if s/he feels more confident. If the code written to satisfy a test does not fairly quickly do so, then the step-size may have been too big, and maybe the smaller testable steps should be used instead. When using external libraries it is important not to make increments that are so small as to be effectively merely testing the library itself [10].

4.1.2 Three laws of using TDD

The TDD practitioners follow a so-called three laws by using the TDD, which are as follows:

1) *You may not write production code unless you've first written a failing unit test.*
2) *You may not write more of a unit test than is sufficient to fail.*
3) *You may not write more production code than is sufficient to make the failing unit test pass.*

These three laws lock you into a tight loop in which you first write a portion of a unit test that fails, and then you write just enough production code to make that test pass. This loop is perhaps two minutes long and almost always ends in success [27].

4.2 Claims concerning TDD

This thesis is aimed to collect claims concerning Test Driven Development, and evaluate these claims. Due to the limitation of time, this thesis only collects 7 of claims. And the 7claims are 6 positive and 1 negative. This section will introduce these 7 claims.

1. XP with TDD has better productivity than waterfall model [53]
This claim concerns programmer's productivity, which means programmers who practice XP with TDD, will develop code faster than programmers who develop code with a more traditional waterfall-like practice. The higher productivity is, the faster programmer developing code. Programmers' productivity will be measured by the lines Source Line of Code (SLOC) per hour.

2. TDD has advantage in defect reduction [44]
This claim concerns defect reduction, which means using TDD, will reduce the number of defects injection. Compare with a test-after technique, the number of the defects is less.

3. XP with TDD has better Flexibility than waterfall model [39]
This claim concerns the flexibility, which means XP with TDD handles changes in requirements better than traditional approaches, a waterfall model like.

4. TDD has a nearly 100% code coverage for test cases [39]
This claim concerns test quality, which means using TDD, will improve quality of the code. The quality of code here is measured by code coverage for test cases. And the code coverage for test cases nearly 100%.

5. Test Driven Development drives the design [43]
It claims that using TDD can drive the design.

6. XP with TDD detects defect earlier [26]
This claim concerns the defect injunction detection, which means using XP with TDD can detect defect earlier.

7. TDD is limited on applicability of practice [25]
It claims that TDD has limitation of applicability in practice, which means TDD may not suitable for some project, e.g. the GUI application, large system etc.

5. Literature Research

This chapter consists of three subsections. The first section will briefly introduce 2 of researched papers in this thesis. The second section will introduce the criteria of evaluating the claims in this thesis. The last section will evaluate the claims by literature research based on evaluation criteria.

5.1 The Literature Research

While some practitioners have applied some form of TDD for several decades [42], academic and industry studies have only more recently emerged [43]. These studies have examined the effects of TDD on external quality and programmer productivity with somewhat mixed results. This section will briefly introduce 2 of researched papers, which performed case study concern the comparison between using XP with TDD and the waterfall model. Literature research is purpose on giving a literary evaluation on those claims.

5.1.1 Boby George and Laurie Williams

The Boby George and Laurie Williams's paper examine two hypotheses which are: 1. The TDD practice will yield code with superior external code quality when compared with code developed with a waterfall-like practice. External code quality will be assessed based on the number of functional, black-box test cases passed. 2. Programmers who practice TDD will develop code faster than programmers who develop code with a more traditional waterfall-like practice. Programmers' productivity will be measured by the time (hours) to complete the development [25]. The hypotheses were evaluated by carrying out an experiment. The experiment and the conclusion of the paper can be seen as follow:

Experiment Design
In Boby George and Laurie Williams's paper, the experiment was aimed to evaluate the External code quality, Productivity, Correlating productivity and quality and Code coverage between using the TDD and the classical model.

Experiment Details
The experiment consists of two trials:

1) Professional programmers randomly assigned to two groups: TDD (Test first) and Control (Test after). The two groups were asked to develop a bowling game application with a same set of requirements. All programmers used the pair programming. Participants were asked to turn in their programs upon completing the activities as outlined.

2) All programmers with same organizing into team. But all programmers were asked to handle error conditions gracefully and were not provided acceptance test cases. Additionally, the control group were asked to write automated test cases after development.

External Validity

The strength of the experiment is that the experiment was done with practitioners in there own environment.

The experiment also has following limitations:

1) Sample size was very small(six TDD pairs and six Control pairs)
2) The experiment requirement changed in the second trial
3) The experiment carried out by the combination of TDD and pair programming
4) The application used in the evaluation process was very small
5) The programmers have different background with using TDD [25].

Experiment Result

The experiment results were introduced based on:

1) External code quality:

The TDD pairs' code passed approximately 18% more test cases than the control group pairs.

2) Productivity:

On average, the TDD pairs took approximately 16% more time to develop the application than the control group pairs.

3) Correlating productivity and quality:

The two-tailed Pearson correlation had a value of 0.661, which was significant at the 0.019 level. This analysis indicates that the higher quality may be the result of the increased time taken by the TDD pairs and not solely due to the TDD practice itself.

4) Code coverage:

The TDD programmers' test cases achieved a mean of 98% method, 92% statement and 97% branch coverage.

Conclusion

1) TDD practice appears to yield code with superior external code quality, as measured by conformance to a set of black-box test cases, when compared with code developed with a more traditional, waterfall-like model practice.

2) The experiment results showed that TDD programmers took more time (16%) than control group programmers. However, the variance in the performance of the teams was large and these results are only directional. Additionally, the control group pairs did not primarily write any worthwhile automated test cases, making the comparison uneven.

3) On an average, survey results indicate that, 80% of the professional programmers thought TDD was an effective practice and 78% believed the practice improves programmers' productivity. The survey results are statistically significant.

4) Survey results also indicated that TDD practice facilitates simpler design and that

lack of upfront design is not a hindrance. However, for some, transitioning to the TDD mindset is difficult. [25]

Obviously, the hypothesis 1 is hold from the experiment and the hypothesis 2 is not hold.

5.1.2 Lei Zhang, Shunsuke Akifuji, Katsumi Kawai, and Tsuyoshi Morioka

In order to popularize the Test Driven Development (TDD) practice in Chinese offshore companies, an experimental research was firstly conducted to compare TDD with the traditional waterfall development in a small-scale project. The result of the experiment can be seen as follow:

Experiment Design
The experiment was designed to evaluate the efficiency of the TDD.

Experiment Details
8 students divided into 2 groups, which were group “T” and “C”. The two groups were assigned same project of “Working Attendance Management System”.

External Validity
The experiment size was small

Experiment Result
1) The superiority of TDD to Waterfall on productivity is 10%.
2) The superiority of TDD to Waterfall on reliability is 28%.
3) The superiority of TDD to Waterfall on maintainability is 8%.
4) The superiority of TDD to Waterfall on flexibility is 30%.
5) The superiority of TDD to Waterfall on efficiency is 33%.
6) The superiority of TDD to Waterfall on Tester quality is 10%.

Conclusion
(1) The TDD developers took less time (10%) than the traditional developers. This stated that the TDD approach had higher productivity.
(2) The TDD approach appeared to yield code with the superior reliability, maintainability, flexibility and efficiency. The bugs found during the developing process were 28% less than those of the traditional group. The average time used to remove one bug in the TDD group was 8% shorter than that of the traditional group. The time used for the requirement variation of TDD was 30% shorter, and the code size was 33% smaller than those of the traditional group, respectively.
(3) The test code coverage of the TDD approach was about 10% higher than that of the traditional group [45]

5.2 The evaluation criteria

As mentioned before, this thesis focusing on evaluating those claims about the TDD. Some of those claims involve the comparison between the using XP with TDD and Waterfall model. And the comparison concerns 4 aspects in productivity, defect reduction, flexibility and test quality as parameters. Those 4 parameters are introduced as follows:

1. Productivity: Programmers' productivity is measured by productivity metric, e.g. Function Points (FPs) per month or Source Line of Code (SLOC) per month. The higher productivity is, the faster programmer developing code.

2. Defect reduction: The defect reduction measured by defect rate. The defect rate here is the total defects' number (DN) during the developing process divide to the lines of LOC. It can be simply expressed as DN/nLOC. The lower defect rate leads good defect reduction.

3. Flexibility: The flexibility of system is measured by the time used to adapt the requirement variations per LOC. The number lines of LOC here is calculated by the number of SLOC after modify minus the number lines of LOC before modify. The shorter time spending on per modified LOC, the better flexibility is.

4. Test quality: The test quality of system is measured by the results of the code coverage on methods, blocks and lines for test cases.

So the claim 1 refers the productivity, the claim 2 refers defect reduction, the claim 3 refers the flexibility and the claim 4 refers the test quality.

5.3 The evaluation of claims by literature study

In this section, those claims will be evaluated by literature studies, and the results can be seen as follow:

1. XP with TDD has better productivity than waterfall model

Keith Ray [53] claims that using XP with will produce code faster than using waterfall model. And Keith Ray argued, in Test-Driven Development, testing is part of the design process; it doesn't take much time to write a small test that represents a part of your thinking about a problem. Test-after is slower because the traditional design/code process -- without tests -- takes about the same amount of time as the TDD design/code process, and then the traditional coding time is followed by writing tests that take even more time [53].

K. Beck [24] said no studies have categorically demonstrated the difference between TDD and any of the many alternatives in quality, productivity, or fun. However, the anecdotal evidence is overwhelming, and the secondary effects are unmistakable. Programmers really do relax, teams really do develop trust, and customers really do learn to look forward to new releases.

This thesis also researched several papers concerning comparing productivity between using XP with TDD and waterfall model.

Lei Zhang [5.1.2] performed an experiment to evaluate the productivity between using XP with TDD and waterfall model. Two groups were formed, which are one using XP with TDD and one using waterfall model. The two groups worked with the same subject “Working Attendance Management System”. The productivity were measured by the total time spend for the project. As a result, the less time spend, the higher productivity is. The experiment’s result is the XP with TDD team spend less 10% time than waterfall model, which means using XP with TDD has higher productivity [45].

The David and Hossein [6] also performed an experiment, which contain two groups (TDD and water) doing the same subject “Graph Base”. The experiment results that the XP with TDD team produce almost twice features than the waterfall team in the same time. And the paper concludes that using XP with TDD is more productive than using waterfall model.

So, as K. Beck said, no studies have categorically demonstrated the difference between TDD and any of the many alternatives in quality, productivity, or fun. But the empirical research can also give some valuable information, although it can’t provide hard evidence. So, this claim is supported by literatures.

2. 2 TDD has advantage in defect reduction

The authors [44] claim that using XP with TDD has benefit at reducing defect injection. The authors argued, debugging and software maintenance is often viewed as a low-cost activity in which working code defect is “patched” to alter its properties, and specifications and designs are neither examined nor updated [30]. Unfortunately, such fixes and “small” code changes may be nearly 40 times more error prone than new development [31], and often new faults are injected during the debugging and maintenance. The TDD test cases are a high-granularity low-level regression test. By continuously running these automated test cases, one can find out whether a change breaks the existing system. The ease of running the automated test cases after changes are made should allow smooth integration of new functionality into the code base and reduce the likelihood that fixes and maintenance introduce new permanent defects [44].

The authors [44] also performed a case study at IBM, which two teams (TDD and waterfall) were formed. The two teams developed two different systems separately.

The TDD team developed a legacy system for seventh version on existing platform. And the waterfall team developed the legacy system for first version on new platform. The case study concludes that TDD team has about 40% reduction in FVT detected defect density of new/changed code when compared with waterfall team.

Lei Zhang [5.1.2] performed an experiment to evaluate the productivity between using XP with TDD and waterfall model. Two groups were formed, which are one using XP with TDD and one using waterfall model. The two groups worked with the same subject “Working Attendance Management System”. The experiment results that the TDD group has less 28% bugs than the waterfall group [45].

Patrick [54] also mentioned that the clearest benefit is verification: you get an exhaustive suite of automated unit tests that constantly protect your system from defects, no matter what changes are made. You get drastically fewer defects throughout the system lifecycle, because your tests enable you to find and kill most bugs as soon as they are born [54].

So, the literature study implies that the short loop test – code – test will reduce the defect injection. Since the test cases running all the time, the defect will be remove immediately. This claim is supported by literatures.

3 XP with TDD has better Flexibility than waterfall model

H. Wasmus [39] claims that using XP with TDD has better flexibility than using waterfall model. And H. Wasmus argued that using XP with TDD makes the development process more suitable for changes in requirements. Because of an iterative based process, which results in working prototypes, possible change request can be identified earlier. In the traditional development process, the final prototyping/product delivered to the use at the end. At that point, simple change can require immense amount of time. A result is that once a product is delivered, developers will try to avoid or ignore change requests. XP with TDD makes changes easier to implement with a better suited final product as result [39].

“The flexibility is quit important issue in the software development. The programmers do not like and afraid of changing code. The old maxim, “If it isn’t broke, don’t fix it!” is a common attitude among software developers. TDD alleviates this fear because you can check any change to the software almost instantly. If the tests all pass, it’s unlikely that the change broke something unexpected. The tests make small changes virtually risk free [3]. The TDD is adoptive to change code by correction or requirement changing”. Robert C. argued so.

K. Beck also mentioned that XP shortens the release cycle, so there is less change during the development of a single release. During a release, the customer is welcome to substitute new functionality for functionality not yet completed. The team doesn't even notice if it is working on newly discovered functionality or features defined

years ago [1].

Lei Zhang's [5.1.2] experiment result also shows that the TDD group using the time for requirement variation is 30% shorter than the waterfall group.

Using XP with TDD embraces change, the literatures are supportive for this claims.

4 TDD has a nearly 100% code coverage for test

H. Wasmus claims that using TDD has higher code coverage for test and it is nearly 100%. H. Wasmus also argued that the quality of the code will also improve with TDD. Due to creating test first, a nearly 100% code coverage of tests will be acquired automatically [39].

Paper [5.1.1] [25] analyzed code coverage as an indication of the quality of the test cases written by TDD programmers. The industry standard for coverage is in the range 80–90%, although ideally the coverage should be 100% [46]. The Boby and Laurie found that the TDD programmers' test cases achieved a mean of 98% method, 92% statement and 97% branch coverage from their experiment. This result is excluding the main method from the code coverage. Including the main method into the code coverage analysis will have lowered the TDD programmers' coverage results [25].

"Since the fine granularity of the test-then-code cycle gives continuous feedback to programmer [25], the code were written to pass the test; hence the trashy code won't be written. So the code coverage with TDD will be higher and nearly 100%", Boby argued.

K. Beck also mentioned that statement coverage certainly is not a sufficient measure of test quality, but it is a starting place. TDD followed religiously should result in 100 percent statement coverage [52].

So, the literature study implies to write test before coding, and the production code followed written, this is a way ensures code coverage. The literatures support this claim.

5. Test Driven Development drives the design

David and Hossein claim that TDD is not just for test, and it drives design. They also argued that TDD is the craft of producing automated tests for production code, and using that process to drive design and programming. For every tiny bit of functionality in the production code, you first develop a test that specifies and validates what the code will do. You then produce exactly as much code as will enable that test to pass. Then you refactor (simplify and clarify) both the production code and the test code [43]. The TDD idea that a test can be written before the program or that test can aid in deciding what program code to write and what that program's interface should look like is a radical concept for most software developers [43].

It is also argued from Wiki, Test-driven development can help to build software better and faster. It offers more than just simple validation of correctness, but can also drive the design of a program. By focusing on the test cases first, one must imagine how the functionality will be used by clients (in this case, the test cases). Therefore, the programmer is only concerned with the interface and not the implementation [26].

K. Beck also mentioned that if it's hard to write a test, it's a signal that you have a design problem, not a testing problem. Loosely coupled, highly cohesive code is easy to test [16].

So, the literatures support that TDD drives design.

6. XP with TDD detects defect earlier

This claim is collected from Wiki [26]. And it was argued as large numbers of tests help to limit the number of defects in the code. The early and frequent nature of the tests helps to catch defects early in the development cycle, preventing them from becoming endemic and expensive problems. Eliminating defects early in the process usually avoids lengthy and tedious debugging later in the project [26].

Boby and Laurie [25] also argued that the fine granularity of the test-then-code cycle gives continuous feedback to programmer. With TDD, faults are identified quickly as new code is added to the system; hence the source of the problem is more easily determined.

So, with the TDD, the test goes alone with the whole development process, even the very beginning phase or the end. And the defect will be discovered immediately when running test failed. So the literatures support that using XP with TDD may detect fault earlier, compared with waterfall model, the test executes after implementing the whole system.

7. TDD is limited on applicability of practice

Boby and Laurie claimed that TDD is limited on applicability, and argued that some codes are inherently hard to test using TDD (for example GUIs [32]). Further, the TDD practice requires considerable effort to be expended on writing mock test objects. Additionally, since no formal documentation takes place, the rationale behind important decisions is not documented and can get lost [25].

For the TDD with GUI, it is argued from Wiki, TDD is difficult to use in some situations, such as graphical user interfaces or relational databases, where systems with complex input and output were not designed for isolated unit testing or refactoring. [26].

David Astels [54] also mentioned that using TDD to develop the Graphical User

Interface (GUI) of an application is tricky. Part of the problem is that GUIs are, by definition, graphical [54].

For the TDD with mock test object, it is also argued [26], the use of the Mock Object design pattern is necessary in order to control the scope of dependencies involved in unit tests. However, when creating mock objects to interact with the module being tested, it is necessary that the developer have a good understanding of the behavior of the entities that are being mocked. Failure to achieve this understanding can lead to problems when the modules are deployed into a "real life" environment and they receive input they were not expecting or give output the environment can not handle. In many instances, creating mock objects that function realistically enough to allow for appropriate unit testing can be difficult and time consuming [26].

This claim is controversial but supported by literatures.

6. Experiments

This chapter contains three sub sections, a section for describing the experiment, a section for documenting the experiment's process and a section for evaluation of the claims by the experiment's result.

Why Experiment?
Experimentation in software engineering can be difficult. Formal, controlled experiments, such as those conducted with students or professionals, over relatively short periods of time are often viewed as "research in the small" [50]. These experiments may suffer from the external validity limitations (or perceptions of such). On the other hand, case studies can be viewed as "research in the typical" [50]. Concerns with case studies involve the internal validity of the research, or the degree of confidence and generalization in a cause-effect relationship between factors of interest and the observed results [51]. There is also an apprehension with case studies of the ability to make a valid comparison between the baseline and the new treatment, since the same project is generally not replicated. Finally, case studies often cannot yield statistically significant results due to a small sample size. Nonetheless, case studies can provide valuable information on a new technology or practice. By performing multiple case studies and recording the context variables of each case study, researchers can build up knowledge through a family of experiments [5] which examine the efficacy of a new practice. In this thesis, the evaluation of the claims concerning the TDD practice will be made by performing an experiment to give a hand-on perception.

6.1 Experiments Description

This section is literal description of the experiment. It will describes the experiment's purpose, the experiment's subject, the procedure of the experiment, the evaluation strategies for evaluating each claim, the methodology be used in the experiment and the validity of the experiment.

6.1.1 Experiment purpose

The experiment is purposing on collecting data between using XP with TDD and the Waterfall Model in productivity, defects reduction, flexibility, test quality and the external code quality. Also the experimenter's perception during the experiment will also be recorded, because the claims which are not related with those 5 parameters will be evaluate these perceptions. This experiment will give a hand-on experience of evaluation.

6.1.2 Experiment subject

The experiment consists of two small scale project, which going to develop a Cinema reservation system and a Shop stock management system. Those two systems are both local GUI based application. And both the systems use database to store data. The Shop stock management system is going to be developed using XP with TDD. And the Cinema reservation system is going to be developed using waterfall model. The table 6.1 and 6.2 give an overview of the functions of the two systems had respectively.

Function	**Description**
Add	The user can use this function to add new film information into the database.
Remove	The user can use this function to remove film information from the database.
Show all film info	The user can use this function to show all the films information from the database.
Reserve Seat	The user can use this function to reserve a seat for a film.
Show Reservation	The user can use this function to show the reservation information for one film.

Table 6.1 the functionalities of Cinema reservation system

Function	**Description**
Add	The user can use this function to add new product information into the database.
Remove	The user can use this function to remove product information from the database.
Modify	The user can use this function to modify the product information from the database.
Show OOS	The user can use this function to show the products, which are *out of stock*. The product out of stock is interpreted as the amount of the product less than 50.
Show Stock	The user can use this function to show the current stock.

Table 6.2 the functionalities of Shop stock management system

6.1.3 Experiment Tools and Methodology

The below list out the tools and methodology have been used in the experiment:

MySQL 5.0: The both two systems will use the MySQL 5.0 as the database system.
Eclipse: The Eclipse will be the development environment for the both experiments.
JAVA: The java will be the main programming language in the experiment.

JUnit: The test will be carried out under the JUnit framework.
Test Driven Development: The Shop stock management system will be developed under the Test Driven Development.
Waterfall Model: The cinema reservation system will be developed under the Waterfall Model.
EclEmma: EclEmma is a free Java code coverage tool for Eclipse and is based on EMMA code coverage tool. EclEmma is used to calculate the code coverage for test cases.
Metrics 1.3.6: Metrics is a free eclipse plug-in used to gather metrics for both systems.
PSP (Personal Software Process): There are 9 PSP tables will be used during the experiments.

- The table of ***Time recording log for Cinema reservation system*** will record the time spend on each activity of the development in Cinema reservation system.
- The table of ***Bug recording log for Cinema reservation system*** will record the number of bug and the bug injection from Cinema reservation system.
- The table of ***Modify recording log for Cinema reservation system*** will record the time of modify function from Cinema reservation system.
- The table of ***PSP Project Summery Form Cinema reservation system*** will give all the results from the Cinema reservation system.
- The table of ***Time recording log for Shop stock management system*** will record the time spend on each activity of the development in Shop stock management system.
- The table of ***Design changing injection for Shop stock management system*** will recode the times of changing design from the Shop stock management system development.
- The table of ***Bug recording log for Shop stock management system*** will record the number of bug and the bug injection from Shop stock management system.
- The table of ***Modify recording log for Shop stock management system*** will record the time of modify function from Shop Stock Manage System.
- The table of ***PSP Project Summery Form Shop stock management system*** will give all the results from the Shop stock management system.

6.1.4 Evaluation Strategy

This section will introduce how the claims will be evaluated from the experiment. Reflect on the claims, the thesis has the following strategies for each claim:

1) **XP with TDD has better productivity than waterfall model**
The sum time of using XP with TDD and the sum time of using waterfall model will be recorded and calculated by using PSP time log table. And the SLOC for each system will be calculated. Finally, the productivity will be calculated and be compared.

2) **TDD has advantage in defect reduction**
The number of bugs found in both systems will be recorded into PSP table. And the defect rate will be calculated and compared.

3) **XP with TDD has better Flexibility than waterfall mode**l
The experiment was designed to change the requirement once in both systems. For the Cinema reservation system, the function of *Reserve* is required to be modified. For the Shop stock management system, the function of *login* is required to be added. The period time of successful modifying/adding this function will be recorded in PSP table. The number of modified LOC will be calculated. And the comparison on flexibility will be made.

4) **TDD has a nearly 100% code coverage for test cases**
In this experiment, the code coverage for test cases of the Shop stock management system will be calculated by the tool of EclEmma.

5) **Test Driven Development drives the design**
This claim will be evaluated by the perceptions from the experiment. The perception will be recorded.

6) **XP with TDD detect defect earlier**
From this claim, the time when detect the bug will be recorded both in the TDD development and the waterfall development. For instance, the whole project lifecycle will calculate to a sum time, which is like 0 to 50 hours. And the time when detect the bug could be at the 10^{th} hour of the whole project lifetime.

7) **TDD is limited on applicability of practice**
The evaluation will be made by the observation during the experiment.

To achieve the above strategies, the following variables need to observe and record during the experiment.

1. *The sum time spend on from Cinema reservation system*
2. *The sum time spend on from Shop stock management system*
3. *The Source Line of Code for Cinema reservation system*
4. *The Source Line of Code for Shop stock management system*
5. *The number of the bugs from Cinema reservation system during the developing process*
6. *The number of the bugs from Shop stock management system during the developing process*
7. *The time of modify function from Cinema reservation system*
8. *The time of modify function from Shop stock management system*
9. *The modified SLOC for the Cinema reservation system*
10. *The modified SLOC for the Shop stock management system*

11. the code coverage of all test cases the system from Shop stock management system
12. the times of changing design from Cinema reservation system
13. the times of changing design from Shop stock management system

6.1.5 Experiment Procedure

The first system of the experiment is Cinema reservation system. This system is developed using waterfall model, which follows the steps as analysis, design, implementation and test. The project plan is made as the start. And then the functional requirements for the system are addressed by using the use case specification. The class diagram with method and attributes is drawing out after specifying the requirement. The system will be implemented after the finished the class diagram. Then the test strategy is written, and the test based on the test strategy is carried out. The system is well developed after all the bugs removed. All the relevant data for measurement will be recorded into PSP tables.

The second system to be developed is Shop stock management system. The system will be developed using XP with TDD. The project plan for this system is written at the beginning. And then the user story of the function *Add* is made. Then the test case based on the use story is written and write the code to pass the test. The iteration for each function is followed up. The system is integrated from each function and the integration test will be made. All the relevant data for measurement will be recorded into PSP tables.

6.1.6 Experiment Validity

Although experimentation is an accepted approach toward scientific validation in most scientific disciplines, it only recently has gained acceptance within the software development community [49]. In this thesis, the experiment is designed to evaluate those claims concerning the Test Driven Development. So, it is necessary to validate the experiment. The experiment for this thesis has strengths on evaluation strategy and data collection.

Evaluation Strategy
There is one evaluation strategy for each claim. Each evaluation strategy is design to evaluate on specific claim on purpose. The evaluation strategy defines the measurement variables and data should be collected during the experiment. The evaluation strategy also gives the methodology of how to collect the data should be used from the experiment.

Data Collection
The Personal Software Process (PSP) table will be used to collect the data during the

experiment. The PSP is a quantified method aimed to the improvement of the quality and productivity of the personal work of individual software engineers.

6.2 Experiment Process

This chapter contains the development process of the experiment. The first section documents the process of developing the Cinema reservation system, which will be developed using waterfall model. And the second section documents the process of developing the Shop stock management system, which will be developed using XP with TDD.

6.2.1Cinema reservation system

As mentioned previously, the Cinema reservation system will be developed using waterfall model. The system is a simple J2SE GUI based application with few functions. This section will introduce the developing process of the Cinema reservation system following as project plan, analysis, design, implementation, test and maintenance.

a) Project Planning

The figure 6.1 shows the project plan of developing the Cinema reservation system. The project starts from 15-09-2007 to 16-10-2007, and divides into 5 main phases, which are project planning, system analysis, design, implementation and test.

ID	*Task*	*Start*	*Finish*	*Last*
	Project Plan	2007-9-17	2007-9-17	1d
	Requirements Analysis	2007-9-17	2007-9-18	2d
	Use Case Specification	2007-9-18	2007-9-21	4d
	Class Diagram Design	2007-9-24	2007-9-26	3d
	Database Design	2007-9-26	2007-9-28	3d
	Coding	2007-10-1	2007-10-5	5d
	Debugging	2007-10-8	2007-10-11	4d
	Test	2007-10-12	2007-10-16	3d
	Maintenance	2007-10-16	2007-10-16	1d

Figure 6.1 project's plan for Cinema reservation system

b) System Analysis

The functional requirement of the system is the main task during the system analysis phase. And this paper uses the Use Case Specification to describe the user requirement. The system has 6 use case can be seen as follow:

The table 6.3 shows the use case specification of the function *Add*.

USE CASE 1	Add a film information
Summery	The user can use the system to add one film' information into the database, it includes film's name, showing time, date, cinema and the price.
Actors	user
Preconditions	Database connection success
Basic course of events	1. The user presses the ADD button from the GUI. 2. The system show the Add panel out 3. The user enters the film's information (name, showtime, date, cinema, and price) into each text field of Add panel. 4. The user presses the add button 5. the system prompt that add film info successful
Alternative paths	4a. : The user enter wrong format data into the test field 1. The system give a error message of "data format error" 4b. : The film already exist in the database 1. The system prompt that the film already exist
Post conditions	Add a film information successfully
Notes	

Table 6.3 use case specification for add a film

The table 6.4 shows the use case specification of the function *Remove*.

USE CASE 2	Remove film information
Summery	The user can use the system to remove a film's information from the database.
Actors	user
Preconditions	The system should provide the all film's information to the user from a table
Basic course of events	1. The system show all the film's information from a table 2. The user identified the film which to be removed 3. The user clicks the row, which the film to be removed in 4. the user presses the Remove button 5. The system removes the film from database 6. The user presses the refresh button 7. The system shows the new table contains the updated data
Alternative paths	4a. : No film selected, click remove button 1. System print out error
Post conditions	Remove a film's information successfully
Notes	

Table 6.4 use case specification for remove a film

The table 6.5 shows the use case specification of the function *show all film information*.

USE CASE 3	Show all film information
Summery	The system shows all film's information from a table
Actors	
Preconditions	The database load successful.
Basic course of events	1. The system connect to the film database 2. The system load the data from database to vector 3. The system show the film's information from table
Alternative paths	1a. : The database connection failed 1. The system prints out error 2a. : Loading data failed 1. The system prints out error
Post conditions	Show all film information successfully
Notes	This function execute initially

Table 6.5 use case specification for show film

The table 6.6 shows the use case specification of the function *Refresh*.

USE CASE 4	Refresh film's information
Summery	The user can use the system to refresh film's information
Actors	user
Preconditions	
Basic course of events	1. the user presses the refresh button 2. The system connect to the film database 3. The system load the data from database to vector 4. The system show the film's information from table
Alternative paths	2a. : The database connection failed 1. The system prints out error 3a. : Loading data failed 1. The system prints out error
Post conditions	refresh all film information successfully
Notes	

Table 6.6 use case specification for refresh

The table 6.7 shows the use case specification of the function *Reserve a Seat*.

USE CASE 5	Reserve a Seat
Summery	The user can use the system to reserve a seat for one specific film
Actors	user
Preconditions	Data loaded success
Basic course of events	1. The user choose a film from a combo box list to reserve 2. The user enter the customer's name 3. The user choose the Row number of the reservation 4. The user choose the Column number of the reservation 5. The user click the Reserve button 6. The system add reservation into database 7. The system prompt reservation made
Alternative paths	1a. : Empty film list 6a. : The seat has been reserved 1. The system prompt that The seat already booked, try again!
Post conditions	Reserve a seat successfully
Notes	

Table 6.7 use case specification for reserve a seat

The table 6.8 shows the use case specification of the function *show reservation*.

USE CASE 6	Show Reservation
Summery	The user can use the system to show one specific film's reservation
Actors	user
Preconditions	Data load success
Basic course of events	1. The user choose a film from a combo list 2. The user click on the show button 3. The system search from database 4. The system append the reservation to the text area
Alternative paths	1a. : Empty film list 4a. : No reservation The system append to the text area, no reservation found
Post conditions	Add a film information successfully
Notes	

Table 6.8 use case for show reservations

c) Design Part

This part contains the design assumptions the class diagram of the system, the Graphical User Interface and the database design.

Assumption

There are several design assumptions of the system as follow:

1) The film's information is assumed as one film will only show once at the cinema.
2) The cinema has only one show room, which contains row (A-L) and column (1-12).
3) The system assumed that one customer can only reserve one seat at once.

Class diagram

The figure 6.2 shows the class diagram of the Cinema reservation system. The system contains 7classes and one interface.

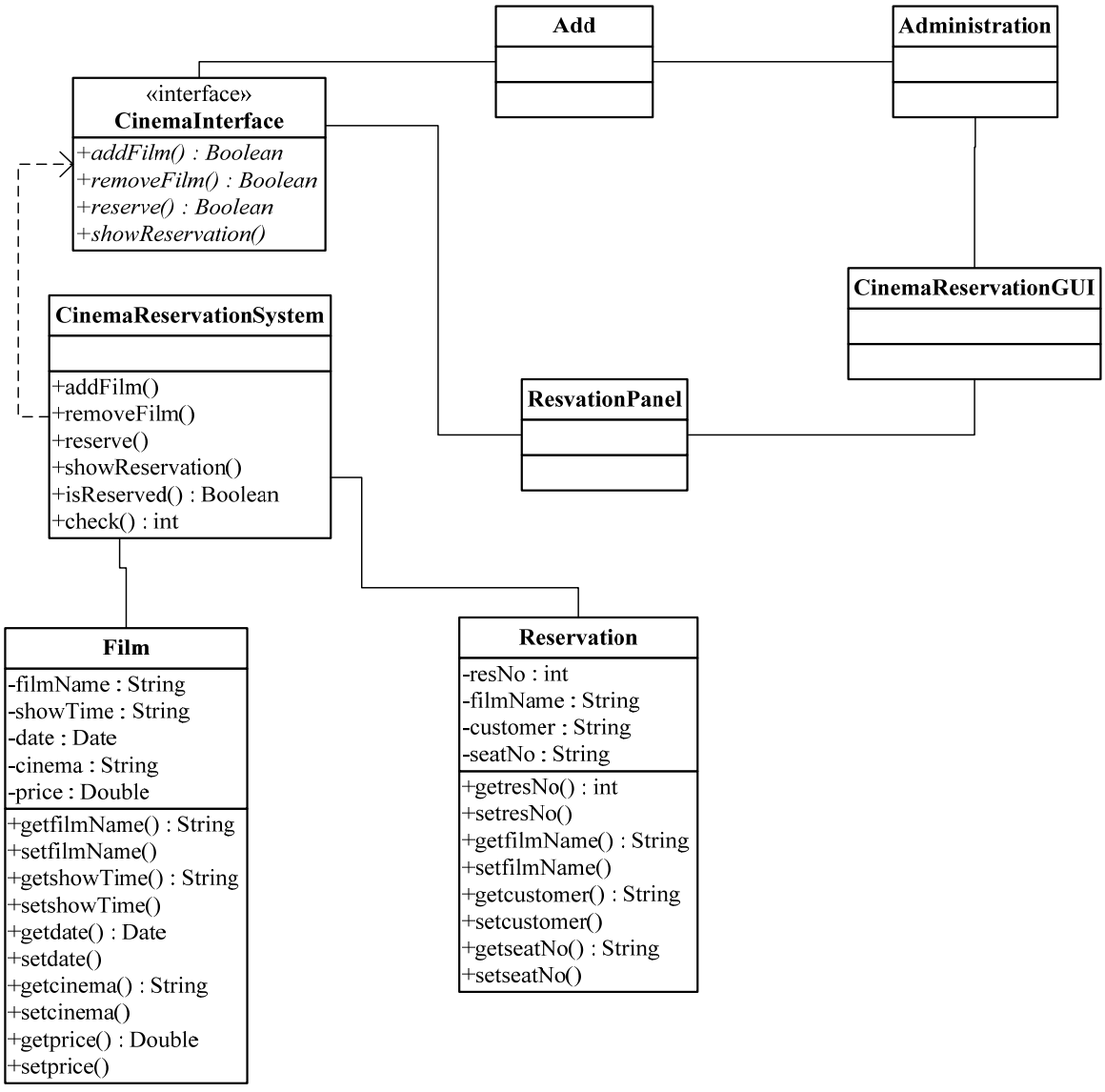

Figure 6.2 the class diagram for Cinema reservation system

Class description:
This section will introduce each class of the Cinema reservation system.

CinemaReservationGUI
This is the main Graphical User Interface of the cinema reservation system. It is a *JFrame* class, which contains a *JTabbedPane*， and the tab pane contains two panel are administration panel and reservation panel.

Add
This is a *JFrame* class, which contains the input text field of a new film's information.

ResvationPanel
This is a *JPanel* class, which contains the functions of *reserve* and *show*.

Administration
This is a *JPanel* class, which contains the functionalities like add, remove and refresh. The panel also contains a table to show all the film's information.

CineamaInterface
This class is an interface of the Cinema reservation system, which contains all the abstract functions of the system,

CineamaReservationSystem
This class implements the interface *CineamaInterface.*

Film
The Film class represent a film, which has 5 fields. A String *filmName* represent the film name. A Time *showTime* represents the show time of the film. A Date *date* represents the date of show. A String *cinema* represents the cinema's name that shows the film. And the last Double *price* represents the price of the film.

Reservation
The Reservation class represents a reservation, which has 4 fields. An Int *resNo* represents the unique ID for a reservation. A String *filmName* represents the film's name. A String *customer* represents the name of the customer. And the last String *seatNo* represents the reserved seat number.

Graphical User Interface
The figure 6.3 shows the GUI of the Cinema reservation system

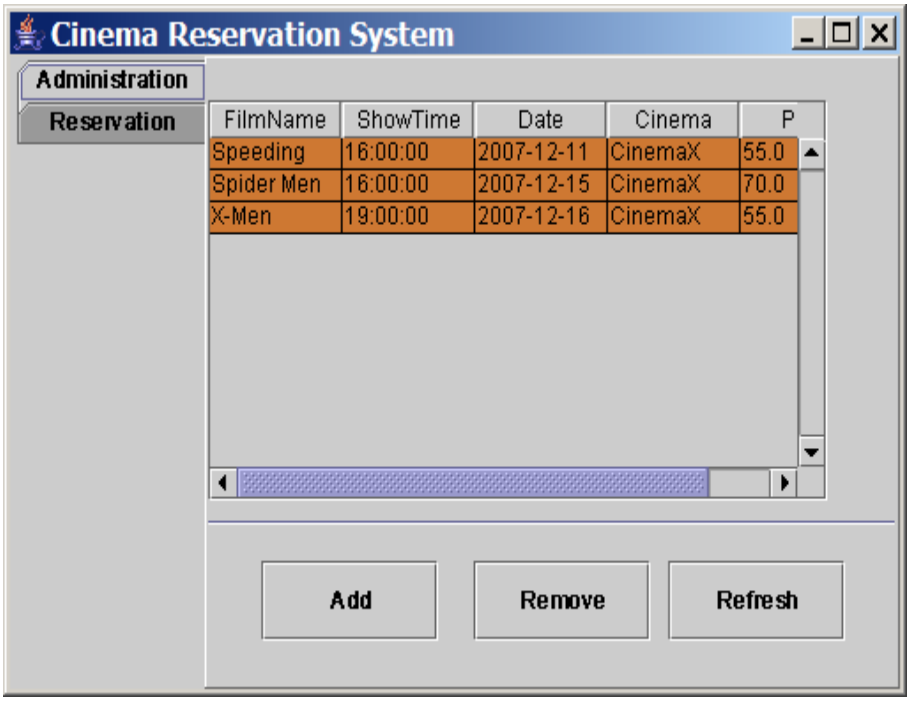

Figure 6.3 the Graphical User Interface for Cinema reservation system

Database Design

The database cinema has two tables, which are filminfo and reservation. The structure of two tables can be seen as follow:

The table 6.9 shows the structure of the table filminfo.

filmname	showtime	date	cinema	price
Varchar(100)	time	date	Varchar(100)	double

Table 6.9 structure of filminfo

The table 6.10 shows the structure of the table reservation.

ReservationID	filmName	costumerName	seatNo
int	Varchar(100)	Varchar(100)	Varchar(100)

Table 6.10 structure of reservation

d) Implementation Part

The implementation part shows the implementation of the system. This section will briefly introduce 3 aspects of the implementation. The whole source code can be found at the CD. The 3 aspects are the database connection part, the GUI and the *Reserve* function.

Database Connection

The system uses the mysql database and a JDBC driver. The database connection and the sql execution by the following code:

```
Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager.getConnection
        ("jdbc:mysql://localhost:3306/cinema",”root","");
    try {
        PreparedStatement stmt = con.prepareStatement(sql);
        ResultSet resultSet = stmt.executeQuery();
        ...
      } finally {
        con.close(); // release the connection
    }
  } catch (Exception e) {
    e.printStackTrace(); // "handle" errors
  }
```

Graphical User Interface

The system has a main JFrame class, which is CinemaReservationGUI.java. And this frame class will add other two panel class when the system starts.

```
JTabbedPane jtp = new JTabbedPane(SwingConstants.LEFT);
...
Administration ad = new Administration();
ResvationPanel rp = new ResvationPanel();
public CinemaReservationGUI(){
...
 jtp.addTab("Administration", null, ad, "Add, Remove film info");
 jtp.addTab("Reservation", null, rp, "Reserve a seat for a film");
 ...
}
```

Reserve function

The user can uses the system to reserve a seat by click the reserve button. To achieve this function, a button action listener is used in the class of *resvationPanel.java*.

```
resButton.addActionListener(new ActionListener()
        {
            public void actionPerformed( ActionEvent ae)
            {
                ...
                if(crs.isReserved(sqlstr)){
                jTextArea.setText("The seats already booked!");
                }else{
                crs.reserve(statement);
                jTextArea.setText("Reservation Made");
                }
            }
        }
```

In order to make successful reservation, two functions will be referenced from the class *CinemareservationSystem.java*. The two functions are *reserve(statement)* and *isReserved(sqlstr).*

e) Test Part

The test section contains two parts, one is the test strategy part and the other one is the test result part. The test is carried out manually.

Test Strategy

A test strategy based on the functional test will be carried out in this project. Scenarios based on the use case will be made. The test will be carried out for each function as follows:

- *Add*: For the function of add, the test will be based on 3 aspects. The first is that inputting all correct format data to make the add film success. The second, input the wrong format data into the system, the system should give an error. The last, input film name that already exist in the database.
- *Remove*: For the function remove, the test will be based on 2 aspects. The first is that remove a film by click the row, which the film in. The second click the remove button without click the film.
- *Show all film info*: For the function show film info, the test will be based on 2 aspects. First, run the system and let the database listening. Second, stop the database and run the system.
- *Reserve Seat:* For the function reserve, the test will be based on 2 aspects. First, select a film, enter the customer name and select the seat number. Second, select the same film the same seat, then make the reservation.
- *Show Reservation:* For the function Show reservation, the test will be based on 2 aspects. First, select a film that has reservation, and click the button show. Second, select a film that has no reservation, and then click the button show.

Test Result

The follows shows the test result based on the test strategy mentioned above.

Test result for function of Add

Test Case 1: Enter Correct Format Film Data
Input: filmName: X-Men, showTime: 19:00:00, date: 2007-12-16, cinema: cinemaX, price: 55
Result: Film has been added

Test Case 2: Enter wrong format film data
Input: filmName: filmName: spider men, showTime: 19:00:00, date: 2007-12-16, cinema: cinemaX, price: fifty

Result: Wrong Data Format!

Test Case 3: Enter a exist film
Input: filmName: filmName: X-Men, showTime: 19:00:00, date: 2007-12-16, cinema: cinemaX, price: 70
Result: The film already exist

Test result for function of Remove

Test Case 4: Remove a exist film
Action: Select the film, and click the remove button
Result: The film removed

Test Case5: Remove a non-exist film
Action: Only click the remove button
Result: Select a film First!

Test result for function of Show Film info

Test Case 6: Start System together with Database
Action: Start the system and Database
Result: All the film information shows in the table

Test Case 7: Start System without Database
Action: Start the system without Database
Result: No data shows in the table

Test result for function of Reserve

Test Case 8: Reserve a seat
Input: filmName: X-Men, customer: Bai, row: A, column: 4
Result: Reservation Made

Test Case 9: Reserve a booked seat
Input: filmName: X-Men, customer: Yu, row: A, column: 4
Result: The seat has been booked

Test result for function of Show reservation

Test Case 10: Show reservation for a film has reservation
Action: Select Speeding, and click show
Result: [2, Speeding, wang, A2]
[1, Speeding, bai, A1]

Test Case 11: Show reservation for a film has no reservation
Action: Select Spider Men, and click show
Result: No reservation Info!

6.2.2 Shop stock management system

This chapter will introduce the entire process of developing the Shop stock management system. The Shop stock management system will be developed using XP with Test Driven Development. The project divides into 8 main phases, which are project plan, iteration of function Add, iteration of function Remove, iteration of function Modify, iteration of function Show OOS, iteration of function Show Stock, the iteration of GUI Design & Integration Test and maintenance. The project plan phase, which plans the project process and introduce the milestones of the project. The Iteration of Function Add phase will firstly gather the user requirement by using user story. And based on the *add* user story, the JUnit test case will be written. Run the test case to see it failed. Then make a little change of the production code. Run the test case to see it succeed. Then write the production code and remove duplicates. The same procedure will go through the phase of *remove, modify, show OOS, and show Stock.* This is also the reason of naming each phase as iteration. And the next phase will firstly design and implement a GUI for the system, and after, the system will be integrated, the integration test will be made based on the functional test at the end of this phase. The last phase is maintenance phases; a new function login will be added in this phase.

a) Project planning
The figure 6.4 shows the project plan of the Shop stock management system.

ID	*Task*	*Start*	*Finish*	*Last*
	Project Plan	2007-10-16	2007-10-17	2d
	Iteration of function Add	2007-10-17	2007-10-19	3d
	Iteration of function Remove	2007-10-22	2007-10-24	3d
	Iteration of function Modify	2007-10-25	2007-10-29	3d
	Iteration of function Show OOS	2007-10-30	2007-11-1	3d
	Iteration of function Show Stock	2007-11-2	2007-11-6	3d
	GUI Design & Integration and Integration Test	2007-11-7	2007-11-14	6d
	Maintenance	2007-11-15	2007-11-15	1d

Figure 6.4 project's plan for Shop stock management system

b) The Iteration of Function Add

This iteration of function *add* follows 5 steps, which are write a JUnit test case *testAdd()*, run the *testAdd()* to see it failed, make a little change for function of *add*, run the *testAdd()* and see it succeed, refactor and remove duplication. This section will introduce the user story of the function *add*, the test case *testAdd()* and the production code of *add* only.

User Story

Add a product
The user can use the system to add a product's information *Shop stock management system*. The system will add the product into database if the product's information doesn't exist in the database.

Table 6.11 user story for add a product

JUnit Test Case

The test case for the function *Add* can be seen as follow:

```
    public void testAdd() {
         ss= new StockSystem();
         sqlstr ="";
         sqlstr ="insert into stock (Barcode, name, Color, Price, Amount)"+
               "values ('213212','Prince','red','11','200')";
         String sqlstr="";
         sqlstr="SELECT * from stock where Barcode= '213212'";
         if(ss.check(sqlstr)==0){
             assertTrue(ss.check(sqlstr)==0);
         }else{
             assertTrue(ss.Add(sqlstr));
         }
    }
```

Application Code

The application code for the function *Add* can be seen as follow:

```
public boolean Add(String addproduct) {
        boolean result=false;
        try{
            Class.forName("com.mysql.jdbc.Driver");
       Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/shopstock","
root","");
        try {
         PreparedStatement stmt = con.prepareStatement(addproduct);
         stmt.execute(addproduct);
         result = true;
        } finally {
           con.close(); // release the connection
        }
    } catch (Exception e) {
         e.printStackTrace(); // "handle" errors
         result = false;
    }
    return result;
    }
```

c) The Iteration of Function Remove

This iteration of function *Remove* follows 5 steps, which are write a JUnit test case *testRemove()*, run all the test cases to see the new one failed, make a little change for function of *remove*, run all the test cases and see all succeed, refactor and remove duplication. This section will only introduce the user story of the function *remove*, the test case *testRemove()* and the production code of *remove*.

User story

Remove a Product
The user can use the system to remove a product's information. The system will remove the product's information from the database if the product exists.

Table 6.12 the user story for remove a product

JUnit Test Case

The test case for the function *Remove* can be seen as follow:

```
public void testRemove() {
       ss= new StockSystem();
       sqlstr = "delete from stock where Barcode = 213212";
       String sqlstr="";
      sqlstr="SELECT * from stock where Barcode= '213212'";
        if(ss.check(sqlstr)==0){
            assertTrue(ss.Remove(sqlstr));
            assertTrue(ss.check(sqlstr)==1);
        }else{
            assertTrue(ss.check(sqlstr)==1);
        }

   }
```

Application Code

The application code for the function *Remove* can be seen as follow:

```
public boolean Remove(String removeproduct) {
       boolean result=false;
       try{
           Class.forName("com.mysql.jdbc.Driver");
      Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/shopstock","
root","");
       try {
        PreparedStatement stmt = con.prepareStatement(removeproduct);
        stmt.execute(removeproduct);
        result = true;
       } finally {
          con.close(); // release the connection
       }
```

```
    } catch (Exception e) {
        e.printStackTrace(); // "handle" errors
        result= false;
    }
        return result;
    }
```

d) The Iteration of Function Modify

This iteration of function *Modify* follows 5 steps, which are write a JUnit test case *testModify()*, run all the test cases to see the new one failed, make a little change for function of *modify*, run all the test cases and see all succeed, refactor and remove duplication. This section will only introduce the user story of the function *remove*, the test case *testModify()* and the production code of *modify*.

User Story

Modify a product's information
The user can use the system to modify a product's information. The system will modify the product's information from database if the product exists.

Table 6.13

JUnit Test Case

The test case for the function *Modify* can be seen as follow:

```
    public void testModify() {
        ss= new StockSystem();
        sqlstr ="";
        sqlstr ="UPDATE shopstock.stock SET "+
               "Barcode='213212', "+
               "name='Prince', "+
               "Color='red', "+
               "Price='11', "+
               "Amount='100' "+
               "where Barcode = '213212'";
         String sqlstr="";
         sqlstr="SELECT * from stock where Barcode= '213212'";
         if(ss.check(sqlstr)==0){
            assertTrue(ss.Modify(sqlstr));
         }else{
            assertTrue(ss.check(sqlstr)==1);
         }
    }
```

Application Code

The application code for the function *Modify* can be seen as follow:

```
public boolean Modify(String modifyproduct) {
       boolean result=false;
       try{
           Class.forName("com.mysql.jdbc.Driver");
      Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/shopstock","
root","");
       try {
        PreparedStatement stmt = con.prepareStatement(modifyproduct);
        stmt.execute(modifyproduct);
        result = true;
       } finally {
          con.close(); // release the connection
       }
   } catch (Exception e) {
       e.printStackTrace(); // "handle" errors
       result = false;
   }
       return result;
   }
```

e) The Iteration of Function Show Stock and Show OOS

This section contains two parts. One part is for the function of *Show Stock*, and the other part is for the function *Show Out Of Stock*. The table 6.14 shows the user story of the function *Show Stock*.

Show Stock
The user can use the system to show the stock of the shop. The user clicks the Show Stock button. The system will show the stock information into a table.

Table 6.14 user story for show stock

The table 6.15 shows the user story of the function Show out Of Stock.

Show out Of Stock
The user can use the system to show the stock of the shop. The user clicks the Show Stock button. The system will show the stock information into a table.

Table 6.15 user story for show out of stock

Due to the validation for JUnit test case using on GUI, the test case for these two functions were hard to write. So these two functions will be implemented at the integration phase.

g) Graphical User Interface and Database Design

According the user requirement of the system, the Graphical User Interface is needed to be achieved. The figure 6.5 shows the GUI of the Shop stock management system.

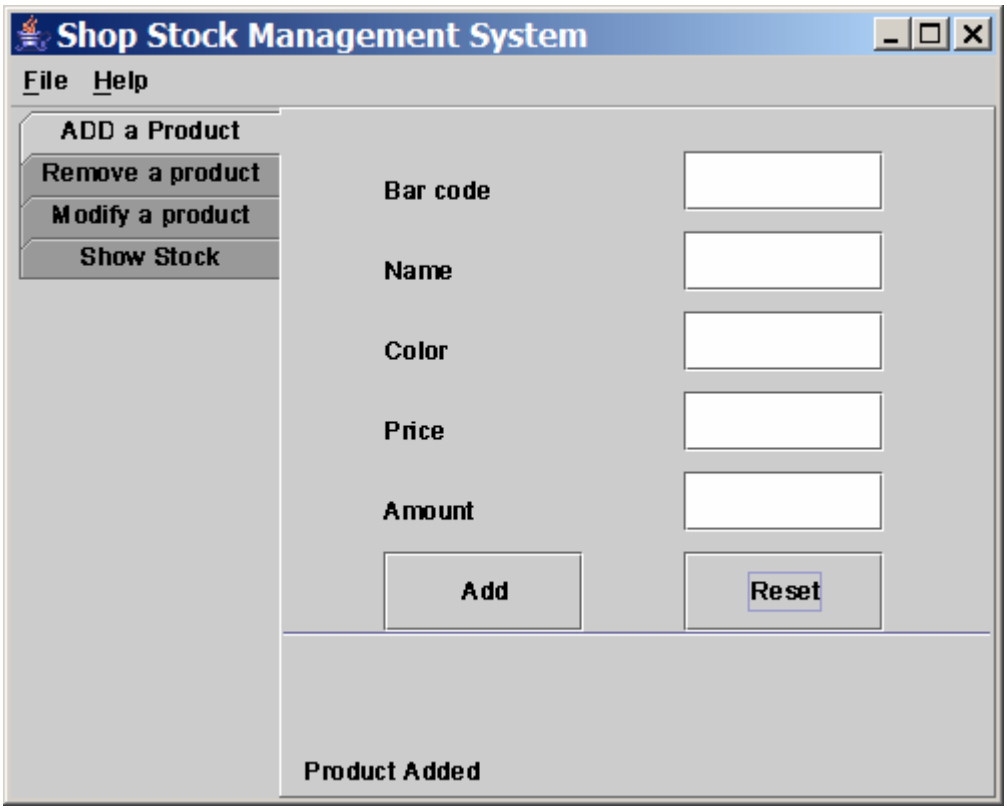

Figure 6.5 the Graphical User Interface for Shop stock management system

The database for Shop stock management system contains two tables, which are shown as following:

The table 6.16 shows the structure of table stock

Barcode	Name	Color	Price	Amount
Varchar(30)	Varchar(30)	Varchar(30)	double	int

Table 6.16 structure of table stock

The table 6.17 shows the structure of table member

username	password
Varchar (100)	Varchar(100)

Table 6.17 structure of table member

h) Integration system

This section contains two parts. The one part will introduce the class diagram of the whole integrated system. And the other part will introduce each class of the system.

Class Diagram

The figure 6.6 shows the class diagram of the Shop stock management system.

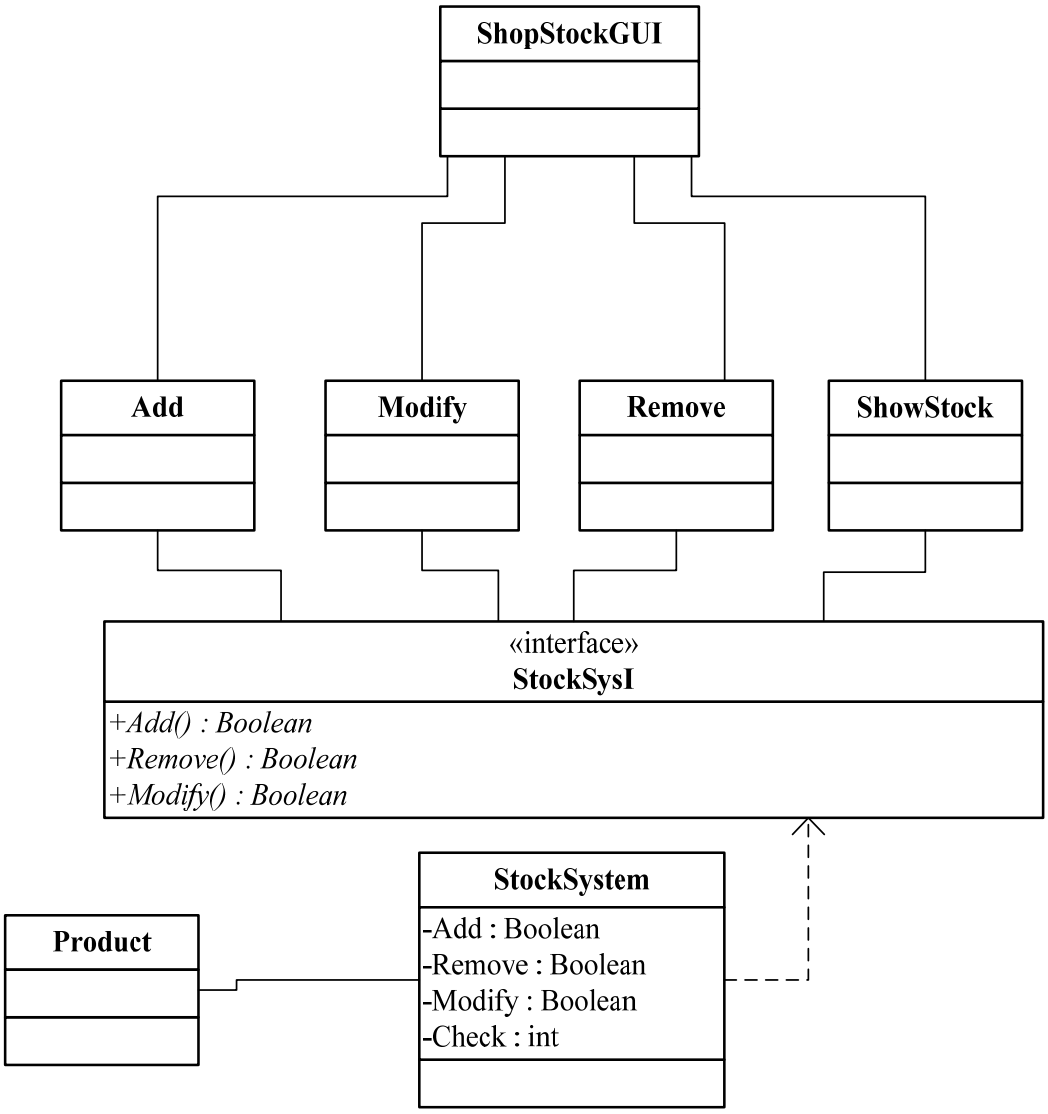

Figure 6.6 the class diagram for Shop stock management system

Class Description

The Shop stock management system contains 6 classes and 1 interface. This section will introduce each class and interface of the system.

ShopStockGUI

This is the main GUI of the Shop stock management system. It provides a

JTabbedPane that will add other panels on it. Those panels are *Add, Modify, Remove and ShowStock.*

Add
This *JPanel* based GUI class, which will be integrated into the main GUI. It provides the input and output of the function *Add*.

Modify
This *JPanel* based GUI class, which will be integrated into the main GUI. It provides the input and output of the function *Modify*.

Remove
This *JPanel* based GUI class, which will be integrated into the main GUI. It provides the input and output of the function *Remove*.

ShowStock
This *JPanel* based GUI class, which will be integrated into the main GUI. It provides the input and output of the function *Show Stock and Show OOS*.

StockSysI
This is the interface of the Shop stock management system and provides the all functions that the system has.

StockSystem
This class implements the interface StockSysI.

Product
The class product represents a product. And it contains 5 fields, which are barcode, name, color, price and amount. A String type barcode represents the bar code of the product. A String type name represents the name of the product. A String type color represents the colour of the product. A Double type price represents the price of the product. An Int type amount represents the amount of the product.

i) Integration Test
The Integration Test is based on the functional test and carried out manually. And this section contains two parts, one part is for the test strategy and the other part is for the test result.

Test Strategy
A test strategy based on the functional test will be carried out in the Integration Test phase. The test will be carried out for each function as follows:

- *Add*: For the function of add, the test will be based on 3 aspects. The first is that inputting all correct format data to make the *add* product success. The second,

input the wrong format data into the system, the system should give an error. The last, input product's barcode that already exist in the database.

- *Remove*: For the function remove, the test will be based on 2 aspects. The first is that remove a product by inputting the product's barcode and click the *Remove* button, the product information was stored in the database. The second aspect is *removing* a product that is not stored in the database.
- *Modify*: For the function *Modify*, the test will be based on 3 aspects. The first is that inputting all correct format data to make the *Modify* product success. The second, input the wrong format data into the system, the system should give an error. The last, input product's barcode that doesn't exist in the database.
- *Show OOS:* For the function Show OOS, the test will be based on 2 aspects. First, for the database contains product's information, clicking the button *Show OOS*. Second, for the database contains no product's information, clicking the button *Show OOS*.
- *Show Stock:* For the function *Show Stock*, the test will be based on 2 aspects. First, for the database contains product's information, clicking the button *Show Stock*. Second, for the database contains no product's information, clicking the button *Show Stock*.

Test Result

The follows shows the test result based on the test strategy mentioned above.

Test result for function of Add

Test Case 1: Enter Correct Format product Data
Input: Barcode: 222334 Name: Juice, Color: yellow, price: 12, amount: 100
Result: Product has been added

Test Case 2: Enter wrong format product data
Input: Barcode: 123400 Name: Coffee, Color: yellow, price: 12, amount: a hundred
Result: Wrong Data Format!

Test Case 3: Enter a exist product
Input: Barcode: 222334 Name: Juice, Color: yellow, price: 12, amount: 100
Result: The product already exist

Test result for function of Remove

Test Case 4: Remove a exist product
Action: Enter the product's barcode, and click the remove button
Result: The product removed

Test Case5: Remove a non-exist product

Action: Enter the product's barcode, and click the remove button
Result: the product doesn't exist

Test result for function of Modify

Test Case 6: Enter Correct Format product Data
Input: Barcode: 222334 Name: Juice, Color: yellow, price: 12, amount: 90
Result: Product's information has been modified

Test Case 7: Enter wrong format product data
Input: Barcode: 123400 Name: Coffee, Color: yellow, price: 12, amount: five
Result: Wrong Data Format!

Test Case 8: Enter a non-exist product
Input: Barcode: hhh111 Name: Juice, Color: yellow, price: 12, amount: 100
Result: The product doesn't exist

Test result for function of Show Stock

Test Case 10: Show Stock with full database
Action: Click the *Show stock* button
Result: All products' information show in the table

Test Case 11: Show Stock with empty database
Action: Click the *Show stock* button
Result: No products' information show in the table

Test result for function of Show OOS

Test Case 12: Show OOS with full database
Action: Click the *Show OOS* button
Result: All products' information with amount less than 50 show in the table

Test Case 13: Show OOS with empty database
Action: Click the *Show OOS* button
Result: No products' information show in the table

The Iteration of Function Login

The login function is added into system after the system developed. The aim of adding this login function is to record a time of changing a requirement. This section contains 5 parts, which are user story, JUnit test case, the application code, the test strategy and

the test result.

User Story

login
The user can only use the system after successfully login.

Table 6.18 user story for login

JUnit Test Case

The test case for the function login can be seen as follow:

```
public void testLogin() {
       ss= new StockSystem();
       sqlstr = "select * from members where username = 'baijohn427'" +
              " and password = '123456'";
       assertTrue(ss.login(sqlstr));
   }
```

Application Code

The application code for the function *login* can be seen as follow:

```
public boolean login(String login) {
       boolean result = false;
       try{
           Class.forName("com.mysql.jdbc.Driver");
      Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/shopstock","
root","");
       try {
        PreparedStatement stmt = con.prepareStatement(login);
        stmt.execute(login);
        ResultSet resultSet = stmt.executeQuery();
        if(resultSet.next()){
        return result=true;
        }else{
        return result = false;
        }
       } finally {
          con.close(); // release the connection
       }
   } catch (Exception e) {
       e.printStackTrace(); // "handle" errors
```

```
    }
        return result;
}
```

Test Strategy

For the function Login, the test will be based on 2 aspects. First enter the correct username and password, press login. Second, enter the wrong username of password, press login.

Test result for function of Login

Test Case 14: login success
Input: username: baijohn427, password: 123456
Result: Login successfully

Test Case 15: Enter a non-exist user
Input: username: yuliana, password: 1111111
Result: Wrong username or password, try again!

6.3 Evaluation of the claims by experiment result

This section will introduce the evaluation for those claims by using the experiment's result and perception.

1) **XP with TDD has better productivity than waterfall model**
The programmer's productivity in this experiment is measured by lines of code per hour. The higher productivity is, the faster programmer producing code.

73.25 hours is spending to develop the Cinema reservation system (waterfall). And 78.75 hours is spending to develop the Shop stock management system (TDD). A free tool Metric 1.3.6 was used to calculate the Source Line of Code. Total lines of code were counted as non-blank and non-comment lines in a compilation unit. The total lines of code for Cinema reservation system are 710 lines. The total lines of code for Shop stock management system are 818 lines.

So, the waterfall model programmer's productivity is 710/73.25, which are approximately 9.69 lines per hour. And the XP with TDD programmer's productivity is 818/78.25, which are approximately 10.45 lines per hour.

So, the XP with TDD programmer's productivity is greater than the waterfall model programmer from the experiment's result. So, the experiment's result supports this claim.

2) **TDD has advantage in defect reduction**

The defect reduction is measured by defect rate in this experiment. The defect rate is the total number of defect divides the total lines of code. The lower defect rate is, the better defect reduction is.

There are 6 defects were found at the implementation and testing phase from the Cinema reservation system (waterfall). The first defect is about Database connection error. When I run the system after me finishing the function of addFilm, the system prints out a database connection error. This is because I write wrong database name. The second defect concerns the function of add. I add a film (Speeding) into database twice. And the system gives an error of key duplicated. The third defect concerns function of remove. I remove a film that does not exist in the database. The system can not found the object. The fourth defect concerns the function of remove. I didn't initialize the variable of vector index. When I refer the filmname from vector, the system give an error of array out of bound. The fifth defect concerns sql syntax error for function reserve. I wrote wrong sql statement for the function of reserve. The sixth defect concerns the JavaNullPointer exception for function show. I didn't call a new object of DefaultTableModel, so the data can't load into JTable.

There are 2 defects were found from the Shop stock management system during the developing process. The first defect concerns the function of remove. I wrote wrong sql statement to remove a product. The system replies a sql syntax error. The second defect concerns the function of modify. I wrote wrong sql statement to modify a product's information. The system replies a sql syntax error.

The total lines of code for Cinema reservation system are 710 lines, and the total lines of code for Shop stock management system are 818 lines.

So the defect rate for Cinema reservation system is 6/710 (approximately 0.0085 defect per LOC), the defect rate for Shop stock management system is 2/818 (approximately 0.0025 defect per LOC). Obviously, the defect rate for Cinema reservation system is greater than Shop stock management system. So the experiment's result support that TDD is advantage in defect reduction.

3) **XP with TDD has better Flexibility than waterfall mode**l

The flexibility of system is measured by the time used to adapt the requirement variations per modified LOC. The number of modified LOC in this experiment is calculated by the number of SLOC after modify minus the number of LOC before modify. The shorter time spending on per modified LOC, the better flexibility is.

The experiment result shows that the Shop stock management system uses one hour to add a new function of login. The initial of the system is supposed to any user can use the system. And the requirement is changed to only the registered user can use the system. A new GUI for login is added into this system. And the login function will fetch the username and password from database. If both username and password are

matched, then the user login successfully. And the Cinema reservation system uses half hour to modify the function of reserve. The initial of this function is supposed to the user can reserve the seat by typing the film name into the film text field. And the function is required to change to the user can select a film from a JComboBox list and make the reservation. A new java swing component JComboBox is added into this system. And the JComboBox will fetch the film names from the database.

The total number of LOC before modify for Cinema reservation system is 685, and the total number of LOC after modify is 710. So the number of modified LOC is 25. The flexibility rate for modifying function reserve of Cinema reservation system can be indicated as approximately 0.02 (0.5/25) hour per LOC. The total number of LOC before modify for Shop stock management system is 693, and the total number of LOC after modify is 818. So the number of modified LOC is 125. The flexibility rate for adding function login of Shop stock management system can be indicated as approximately 0.008 (1/125) hour per LOC.

The experiment's result shows the Cinema reservation system spends more time on per modified LOC than the Shop stock management system. So, this claim is supported from this experiment.

4) **TDD has a nearly 100% code coverage for test cases**

The code coverage for test cases is calculated by a free tool EclEmma. The test is carried out manually in this experiment. The code coverage of test cases for each function can be seen as follow:

Test for function of Add

Test Case 1: Enter Correct Format product Data
Input: Barcode: 222334 Name: Juice, Color: yellow, price: 12, amount: 100
Test Case 2: Enter wrong format product data
Input: Barcode: 123400 Name: Coffee, Color: yellow, price: 12, amount: a hundred
Test Case 3: Enter a exist product
Input: Barcode: 222334 Name: Juice, Color: yellow, price: 12, amount: 100

The table 6.19 shows the code coverage for test of function add

name	Class,%	Method,%	Block,%	Line,%
Add.java	100%(3/3)	100% (5/5)	100% (490/490)	100 %(85/85)

Table 6.19 code coverage for function add

Test result for function of Remove

Test Case 4: Remove a exist product
Action: Enter the product's barcode, and click the remove button
Test Case5: Remove a non-exist product

Action: Enter the product's barcode, and click the remove button

The table 6.20 shows the code coverage for test of function remove

name	Class,%	Method,%	Block,%	Line,%
Remove.java	100%(3/3)	100% (5/5)	100% (216/216)	100% (43/43)

Table 6.20 code coverage for function remove

Test for function of Modify

Test Case 6: Enter Correct Format product Data
Input: Barcode: 222334 Name: Juice, Color: yellow, price: 12, amount: 90
Test Case 7: Enter wrong format product data
Input: Barcode: 123400 Name: Coffee, Color: yellow, price: 12, amount: five
Test Case 8: Enter a non-exist product
Input: Barcode: hhh111 Name: Juice, Color: yellow, price: 12, amount: 100
Result: The product doesn't exist

The table 6.21 shows the code coverage for test of function modify

name	Class,%	Method,%	Block,%	Line,%
Modify.java	100%(3/3)	100% (5/5)	100% (486/486)	100% (81/81)

Table 6.21 code coverage for function modify

Test for function of Show Stock and Show OOS

Test Case 10: Show Stock with full database
Action: Click the *Show stock* button
Test Case 11: Show Stock with empty database
Action: Click the *Show stock* button
Test Case 12: Show OOS with full database
Action: Click the *Show OOS* button
Test Case 13: Show OOS with empty database
Action: Click the *Show OOS* button

The table 6.22 shows the code coverage for test of function show stock and show oos

name	Class,%	Method,%	Block,%	Line,%
ShowStock.java	100%(3/3)	100% (5/5)	97% (477/493)	90% (81.7/91)

Table 6.22 code coverage for function show stock and show oos

The experiment's result shows that the code coverage for testing the function add, remove and modify are all 100%. The median code coverage for testing the function show stock and show oos are 96.75%. The lines are not covered in the ShowStoc.java are the throw exception lines.

So, the experiment's result is supportive for this claim.

5) **Test Driven Development drives the design**
During the process of the TDD project, 2 design decisions were changed. The design changes occurred during the iteration of function Add. The user story told me that the system will give a correct message if the system successfully adds a product's information. And the system will also give a failed message if the system adds a product's information. I initially design the function add as a void type function. But I soon discovered that it is very hard to make an assertion for a void type function. Then I change the function type from void to Boolean. Then I run the test case and see the test failed. Again, I write few lines of code to pass the test. At that moment, I discovered that it is not just sufficient to add a product's information into the database. It is a need to have a function that can check if the product already in the database before run the add function. So, I change my design again. And I design a check function that will check the product whether has been in the database. These are the two design decision change. There is no design decision changed from the waterfall model project.

Also, when I write the test cases for the function add, I also image how the user will interact with this function. So, I am concerned to design the interface too.

So, these perceptions are supportive that TDD drives the design.

6) **XP with TDD detects defects earlier**
The first defect from the Cinema reservation system (waterfall) was detected at 16^{th} day after the project started. And it can be converting to the 50.25^{th} hour of the whole project lifetime (73.25 hours total). This defect is about Database connection error. When I run the system after me finishing the function of addFilm, the system prints out a database connection error. This is because I write wrong database name.

The first defect from Shop stock management system (TDD) was detected at fifth day after the project started. And it can be converting to the 13.25^{th} hour of the whole project lifetime (78.75 hours total). This defect concerns the function of remove. I wrote wrong sql statement to remove a product. The system replies a sql syntax error.

So, the 13.25/78.75 is less than 50.25/73.25. The experiment's result is supportive for using XP with TDD detects defect earlier.

7) **TDD is limited on applicability of practice**
This claim will be evaluated by perception during the experiment. During the developing of the Shop stock management system, the two plans of two iteration of the function S**how Stock** and **Show OOS** is changed. Because these two function require event and a GUI to output data. Due to the validation of automated test framework for GUI, it is very difficult to write JUnit test case for them. So, the TDD has some limitations for Shop stock management system. The experiment's result supports this claim.

7. Conclusion

This chapter contains three sub sections, which are the achievements of this these, the evaluation of claims and the thoughts & further work. The first section will summarize the achievement of this thesis, which is what the thesis has done. The second section will finally evaluate those claims by the thesis finding combine with the result of literature research and experiment. The last section will introduce the thoughts and further work.

The Achievement of this Thesis

Test Driven Development (TDD) is a practice of eXtreme Programming (XP) where unit- and functional tests drive the development of the code. This thesis has collected some of claims about Test Driven Development, and evaluations were done from both literature research and experiment.

The literature research gives a literary evaluation of claims. This thesis has studied some of papers concerning those collected claims. The evaluation based on those papers was done. The experiment gives a hand-on perception of evaluating claims. This thesis has developed two systems, which are Cinema reservation system using waterfall model and Shop stock management system using XP with TDD. All the relevant data from the experiment has been recorded into PSP table. The evaluation was done by the experiment's result.

Evaluation of claims

This section will finally evaluate those claims, which were collected in this thesis. The evaluation will be made based on the literature study and the experiment result. And the thesis's finding will also be concluded in this section.

- XP with TDD has better productivity than waterfall model

The productivity in this thesis is measured by the lines of LOC produced per hour by the programmer. The literature research and experiment's result are both supportive for this claim. This thesis also found that using XP with TDD will save time on writing formal analysis and design document, e.g. use case specification. Since TDD focusing on creating test cases first, programmer design and implement the function faster. So, the thesis supports this claim.

- TDD has advantage in defect reduction

The defect reduction in this thesis is measured by defect rate, which the number of defects during the developing process per LOC. The literature research and experiment's result are both supportive for this claim. This thesis also found that by continuously running those test cases, one can find out whether a change breaks the existing system. This will avoid bugs. So, the thesis confirms this claim.

- XP with TDD has better Flexibility than waterfall model

The flexibility in this thesis is measured by the spending time per modified/added LOC. The literature research and experiment's result are both supportive for this claim. This thesis also found that using XP with TDD has strength in the unstable requirement project. Due to iterative based process, possible change request can be identified earlier.

- TDD has a nearly 100% code coverage for test

The literature research and experiment's result are both supportive for this claim. This thesis also found that due to writing test cases firstly, the production code written to pass the test. This way of proceeding ensures code coverage,

- Test Driven Development drives the design

The literature research and experiment's result are both supportive for this claim. This thesis also found that writing the test cases firstly will lead programmer image the interaction between the user and system. And this also drive programmer to design the interface of system.

- XP with TDD detects defect earlier

The literature research and experiment's result are both supportive for this claim. This thesis also found that using XP with TDD, the test goes alone with the whole development process, the defect will be discovered immediately when running test failed.

- TDD is limited on applicability of practice

The literature research and experiment's result are both supportive for this claim. This thesis also found that the TDD relying on the automated unit test framework. If the developer has lack knowledge of automated unit test, then the TDD has limitation of applicability.

The Thoughts and Further work

This section will cover some thoughts based on a personal vision which is developed during the execution of experiment. These thoughts can be a topic for further research and are not covered by hard evidence in this study.

The main thought that I possess is the influence of unit test framework on TDD. Since the TDD replying on the unit test, possession of knowledge for unit test framework is required to use TDD. Developer may not play TDD well if the developer has lack knowledge of unit test framework. Another thought is that TDD shows advantages with XP. What will be if a system is developed using waterfall model with TDD? Are advantages of TDD kept also? Furthermore, it is controversial for using TDD with GUI, is this really challenge of TDD?

Due to the lack of time, this thesis just collected some of the claims about TDD. In further, I would like to collect some more claims and evaluate them. These can be some claims like TDD is lack of design; the test cases are test asset; TDD enhances programmer’s code comprehension and etc.

Reference

[1] K. Beck, Extreme Programming Explained: Embrace Change. Addison-Wesley, 1999. first edition

[2] Gerardo canfora, Aniello Cimitile, Felix Garcia, Mario Piattini and Corrado Aaron Visaggio, Productivity of Test Driven Development: A controlled Experiment with Professionals, Research Center on Software Technology, University of Sannio, Italy

[3] Robert C. Matin, Professionalism and Test Driven Development

[4] Raghvinder S. Sangwan, Phillip A. LaPlante LaPlante, "Test-Driven Development in Large Projects," IT Professional, vol. 8, no. 5, pp. 25-29, Sept/Oct, 2006

[5] V. R. Basili, F. Shull, and F. Lanubile, "Building Knowledge Through Families of Experiments," IEEE Transactions on Software Engineering, vol. 25, pp.456 - 473, 1999.

[6] David S. Janzen Hossein Saiedian, On the Influence of Test-Driven Development on Software Design. Electrical Engineering and Computer Science University of Kansas, Lawrence, KS USA

[7] http://en.wikipedia.org/wiki/Agile_software_development#_note-2

[8] http://www.agilemanifesto.org/principles.html

[9] Boehm, B.; R. Turner (2004). *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA: Addison-Wesley. ISBN 0-321-18612-5. Appendix A, pages 165-194

[10] http://www.mariosalexandrou.com/methodologies/agile-software-development.asp

[11] http://en.wikipedia.org/wiki/Iterative_and_incremental_development

[12] http://en.wikipedia.org/wiki/Waterfall_model

[13] Laplante, P.A.; C.J. Neill (February 2004). ""The Demise of the Waterfall Model Is Imminent" and Other Urban Myths". *ACM Queue* **1** (10). Retrieved on 2006-05-13.

[14] Sommerville, Ian [1982] (2007). "4.1.1. The waterfall model", *Software engineering*, 8th edition, Harlow: Addison Wesley, pp 66f.

[15] http://heavylogic.com/agile.php

[16] Beck, K.: Extreme Programming Explained: Embrace Change. 2 edn. Addison-Wesley (2004)

[17] Ming Huo; Verner, J.; Liming Zhu; Babar, M.A., "Software quality and agile methods," Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International, vol., no.pp. 520- 525 vol.1, 28-30 Sept. 2004

[18] http://en.wikipedia.org/wiki/Extreme_Programming#_note-0

[19] B. W. Boehm, Software Engineering Economics, Prentice-Hall, 1981.

[20] B. Hambling, “Realistic and Cost-effective Software Testing”, in Kelly, M.: Management and Measurement of Software Quality, UNICOM SEMINARS, Middlesex, UK, 1993, pp. 95-112.

[21] M. J. Harrold, “Testing: a roadmap”, International Conference on Software Engineering, ACM, 2000, pp. 61-72.

[22] W.S. Humphrey, Winning with Software, Addison-Wesley, 2002.

[23] D. J. Mosley and B. A. Posey, Just Enough Software Test Automation, Prentice Hall, 2002.

[24] Kent Beck. Test Driven Development: By Example. Addison-Wesley, 2003.

[25] Boby George and Laurie Williams. A structured experiment of test-driven development. Information and Software Technology, 46(5):337–342, 2004.

[26] http://en.wikipedia.org/wiki/Test-driven_development

[27] R. Martin, “The Bowling Game Kata,” June 2005

[28] Newkirk, JW and Vorontsov, AA. Test-Driven Development in Microsoft .NET, Microsoft Press, 2004

[29] T. A. Corbi, Program understanding challenge for the 1990s, IBM Systems Journal 28 (1989) 294–306.

[30] D. Hamlet, J. Maybee, the Engineering of Software, Addison-Wesley, Boston, 2001.

[31] W.S. Humphrey, Managing the Software Process, Addison-Wesley, Reading, MA, 1989.

[32] K. Beck, Aim, fire, IEEE Software 18 (2001) 87–89.

[33] A. van Deursen, Program comprehension risks and opportunities in Extreme Programming, CWI, Amsterdam, SEN-R0110, ISSN 1386-369X, 2001

[34] A. van Deursen, L. Moonen, A. vandenBergh, G. Kok, Refactoring test code, presented at XP 2001, 2001

[35] F.P. Brooks, The Mythical Man-Month, Addison-Wesley, Reading, MA, 1995.

[36] D. Gelperin, W. Hetzel, Software quality engineering, presented at Fourth International Conference on Software Testing, Washington, C, June 1987

[37] C. Larman, V. Basili, A history of iterative and incremental development, IEEE Computer 36 (2003) 47–56.

[38] D. Chaplin, Test first programming, TechZone (2001).

[39] Hans. Wasmus, the evaluation of the Test Driven Development, Software Engineering Research Group, Department of Software Technology, Faculty EEMCS, Delft University of Technology Delft, the Netherlands

[40] Lars-Ola Damm, Lars Lundberg, David Olsson, Introducing Test Automation and Test-Driven Development: An Experience Report

[41] M. M. Müller, O. Hagner, Experiment about test-first programming, presented at Empirical Assessment In Software Engineering EASE ’02, Keele, April 2002

[42] Craig Larman and Victor R. Basili. Iterative and incremental development: a brief history. IEEE Computer, 36(6):47–56, June 2003.

[43] D. Janzen and H. Saiedian. Test-driven development: concepts, taxonomy and future directions. IEEE Computer, 38(9):43–50, Sept 2005

[44] L. Williams, E. M. Maximilien, M. Vouk, Test-driven development as a defect-reduction practice, presented at IEEE International Symposium on Software Reliability Engineering, Denver, CO, 2003

[45] Lei Zhang1, Shunsuke Akifuji1, Katsumi Kawai2, and Tsuyoshi Morioka2, Comparison Between Test Driven Development and Waterfall Development in a Small-Scale Project

[46] S. Cornett, Code Coverage Analysis, Bullseye Testing Technology, 2002

[47] http://en.wikipedia.org/wiki/Software_testing, visit at 20-09-2007

[48] http://en.wikipedia.org/wiki/Test_case visit at 21-09-2007

[49] Marvin V. Zelkowitz, Dolores Wallace, EXPERIMENTAL VALIDATION IN SOFTWAREENGINEERING, Keele University, Staffordshire, U.K., 24-26 March 1997.

[50] N. E. Fenton and S. L. Pfleeger, Software Metrics: A Rigorous and Practical Approach: Brooks/Cole Pub Co., 1998.

[51] D. T. Campbell and J. C. Stanley, Experimental and Quasi-Experimental Design for Research. Boston: Houghton Mifflin Co., 1963.

[52] Beck, K. Test-Driven Development by Example, Addison Wesley, 2003

[53] http://homepage.mac.com/keithray/blog/2005/01/16/

[54] http://adaptionsoft.com/tdd.html

[55] David Astels, Test Driven Development: A Practical Guide, Prentice Hall PTR July 02, 2003

[56] http://qualitycode.com/html/Essay10.html

Appendices A

The Appendices A contains all the PSP tables from the experiments.

A-1 Time recording log for Cinema reservation system

Date	**Start**	**Stop**	**Interruption (hours)**	**Delta Time(hour)**	**Task**	**Comments**
2007-09-17	10:00	12:00	0.25	1.75	Start Project plan	Interruption for break
2007-09-17	13:00	14:00		1.00	Make Project plan	
2007-09-18	10:00	11:00		1.00	Start Requirement analysis	
2007-09-18	12:00	15:00	0.5	2.5	Write Use case for *Add*	Interruption for break
2007-09-19	12:00	15:00	1.00	2.00	Write Use case *remove*	Interruption for answering call
2007-09-20	10:00	15:00	1.00	4.00	Write use case for *show film info* and *refresh*	Interruption for break and lunch
2007-09-21	10:00	15:00	1.00	4.00	Write use case for *Reserve a seat* and *Show reservation*	Interruption for break and lunch
2007-09-24	10: 00	15:00	1.00	4.00	GraphicaL User Interface Design	Interruption for break and lunch
2007-09-25	10:00	15:00	1.00	4.00	Class diagram Design	Interruption for break and lunch

Date	Start	Stop	Interruption (hours)	Delta Time(hour)	Task	Comments
2007-09-26	10:00	15:00	1.00	4.00	Class Diagram design	Interruption for break and lunch
2007-09-27	10:00	12:00	0.5	1.50	Database Design	Interruption for break
2007-09-28	10:00	12:00	0.5	1.5	Create Database and table	Interruption for break
2007-10-01	10:00	15:00	1.00	4.00	Coding for Main GUI and panel Administration.	Interruption for break and lunch
2007-10-02	12:00	15:00	1.00	2.00	Coding for panel Reservation	Interruption for break
2007-10-03	10:00	15:00	1.00	4.00	Coding for function of *Add* and *Remove*	Interruption for break and lunch
2007-10-04	10:00	15:00	1.00	4.00	Coding for function *refresh* and *show all film info*	Interruption for break and lunch
2007-10-05	10:00	15:00	1.00	4.00	Coding for function *Reserve* And *Show Reservation*	Interruption for break and lunch
2007-10-08	10:00	15:00	1.00	4.00	Debugging	Interruption for break and lunch

Date	Start	Stop	Interruption (hours)	Delta Time(hour)	Task	Comments
2007-10-09	10:00	15:00	1.00	4.00	Debugging	Interruption for break and lunch
2007-10-10	10:00	15:00	1.00	4.00	Debugging	Interruption for break and lunch
2007-10-12	10:00	15:00	1.00	4.00	Test the function *add* and *remove*	Interruption for break and lunch
2007-10-15	10:00	15:00	1.00	4.00	Test the function *refresh* and *show all film info*	Interruption for break and lunch
2007-10-16	10:00	15:00	1.00	4.00	Test the function *reserve* and *show reservation*, modify the function *reserve*	Interruption for break and lunch

Table A-1 the time recording log for Cinema reservation system

A-2 Bug recording log for Cinema reservation system

Bug No.	**Date**	**Bug detected time**	**Stop**	**Delta Time (minites)**	**comment**
1	2007-10-08	11:00	11:20	20	Database connection error
2	2007-10-08	14:00	14:35	35	key duplicated for function add
3	2007-10-09	10:20	10:50	30	The system can not found the object (remove a nonsexist film)
4	2007-10-09	13:00	13:15	15	Array out of bound for function remove
5	2007-10-10	11:00	11:20	20	Sql syntax error for function *reserve*
6	2007-10-10	14:12	15:00	48	JavaNullPointer exception for function show

Table A-2 Bug recording log for Cinema reservation system

A-3 Modify recording log for Cinema reservation system

Date	**Modify start**	**Interruption**	**stop**	**Delta Time(hour)**	**Comment**
2007-10-16	14:00		14:30	0.5 hour	Modify the function of *reserve*. The function initially gets the film name from test field. The function change to get the film name from a combo box list.

Table A-3 Modify recording log for Cinema reservation system

A-4 PSP Project Summery Form for Cinema reservation system

	Time in hours	**Amount**	**Time in whole project lifetime**	**Code coverage %**
The sum time spend on	73.25			
The number of the bugs	---	6		
The time when detect the bug			69%	
The time of modify function reserve	0.5	1		
The number of changing design by		0		

Table A-4 the summery form for Cinema reservation system

A-5 Time recording log for Shop stock management system

Date	**Start**	**Stop**	**Interruption (hours)**	**Delta Time(hour)**	**Task**	**Comments**
2007-10-16	10:00	12:00	0.25	1.75	Start Project plan	Interruption for break
2007-10-16	13:00	14:00		1.00	Make Project plan	
2007-10-17	10:00	11:00		1.00	Start Requirement analysis of function *Add*	
2007-10-17	12:00	15:00	0.5	2.5	Write User story for *Add*	Interruption for break
2007-10-18	12:00	15:00	1.00	2.00	Write the test case of *Add* and the application code	
2007-10-22	10:00	15:00	1.00	4.00	Start requirement analysis of function *remove* and write user story of *remove*	Interruption for break and lunch
2007-10-23	10:00	15:00	1.00	4.00	Write the test case of *remove* and the application code	Interruption for break and lunch
2007-10-25	10: 00	15:00	1.00	4.00	Start requirement analysis of function *Modify* and write the user story	Interruption for break and lunch

Date	Start	Stop	Interruption (hours)	Delta Time(hour)	Task	Comments
2007-10-26	10:00	15:00	1.00	4.00	Write the test case for function *Modify* and the application code	Interruption for break and lunch
2007-10-30	10:00	12:00		2.00	Start requirement analysis and write the use story for function *show OOS*	
2007-10-31	10:00	12:00		2.00	Start requirement analysis and write the use story for function *show Stock*	
2007-11-01	10:00	16:00	1.00	5.00	Design and implement the main application GUI	Interruption for break and lunch
2007-11-02	10:00	16:00	1.00	5.00	Design and implement the panel *Add*, *Remove*, *Modify* and the *Show Stock*	Interruption for break and lunch
2007-11-03	10:00	16:00	1.00	5.00	Design each class and draw class diagram	Interruption for break and lunch
2007-11-05	10:00	16:00	1.00	5.00	Design and implement the function *Show OOS* and *show Stock*	Interruption for break and lunch
2007-11-06	10:00	16:00	1.00	5.00	Integrate the system	Interruption for break and lunch
2007-11-07	10:00	16:00	1.00	5.00	Debugging	Interruption for break and lunch

Date	Start	Stop	Interruption (hours)	Delta Time(hour)	Task	Comments
2007-11-08	10:00	16:00	1.00	5.00	Debugging	Interruption for break and lunch
2007-11-09	10:00	16:00	1.00	5.00	Integration Test	Interruption for break and lunch
2007-11-12	10:00	16:00	1.00	5.00	Integration Test	Interruption for break and lunch
2007-11-13	10:00	16:00	1.00	5.00	Integration Test	Interruption for break and lunch
2007-11-14	10:00	11:00		1.00	Write the new function *login*	

Table A-5 time recording log for Shop stock management system

A-6 Modify recording log for Shop stock management system

Date	Modify start	Interruption	stop	Delta Time(hour)	Comment
2007-11-14	10:00		11:00	1 hour	Add a new function login into the system. The system will be only used by authorized user.

Table A-6 modify recording log for Shop stock management system

A-7 Bug recording log for Shop stock management system

Bug No.	**Date**	**Bug detectted time**	**Stop**	**Delta Time (minites)**	**comment**
1	2007-10-23	11:00	11:10	20	Sql syntax error for function *remove*
2	2007-10-26	14:00	14:30	30	*Modify* function can't modify the data from database

Table A-7 bug recording log for Shop stock management system

A-8 Design changing injection for Shop stock management system

Injection No.	**Date**	**Time**	**Comments**
1	2007-10-17	12:00	The function *add* initially designed to be a void type. And it changes to a Boolean function.
2	2007-10-30	10:00	The function *check* was designed besides *add*

Table A-8 Design changing injection for Shop stock management system

A-9 PSP Project Summery Form for Shop stock management system

	Time in hours	**Amount**	**Time in whole project lifetime**	**Code coverage %**
The sum time spend on	78.75			
The number of the bugs		2		
The time when detect the bug			17%	
The time of add function login	1 hour	1		
The number of changing design		2		

Table A-9 the summery form for Shop stock management system

Appendices B

The Appendices B contains the screenshots of the code coverage report from tool EclEmma.

The figure B-1 shows the code coverage for testing function Add

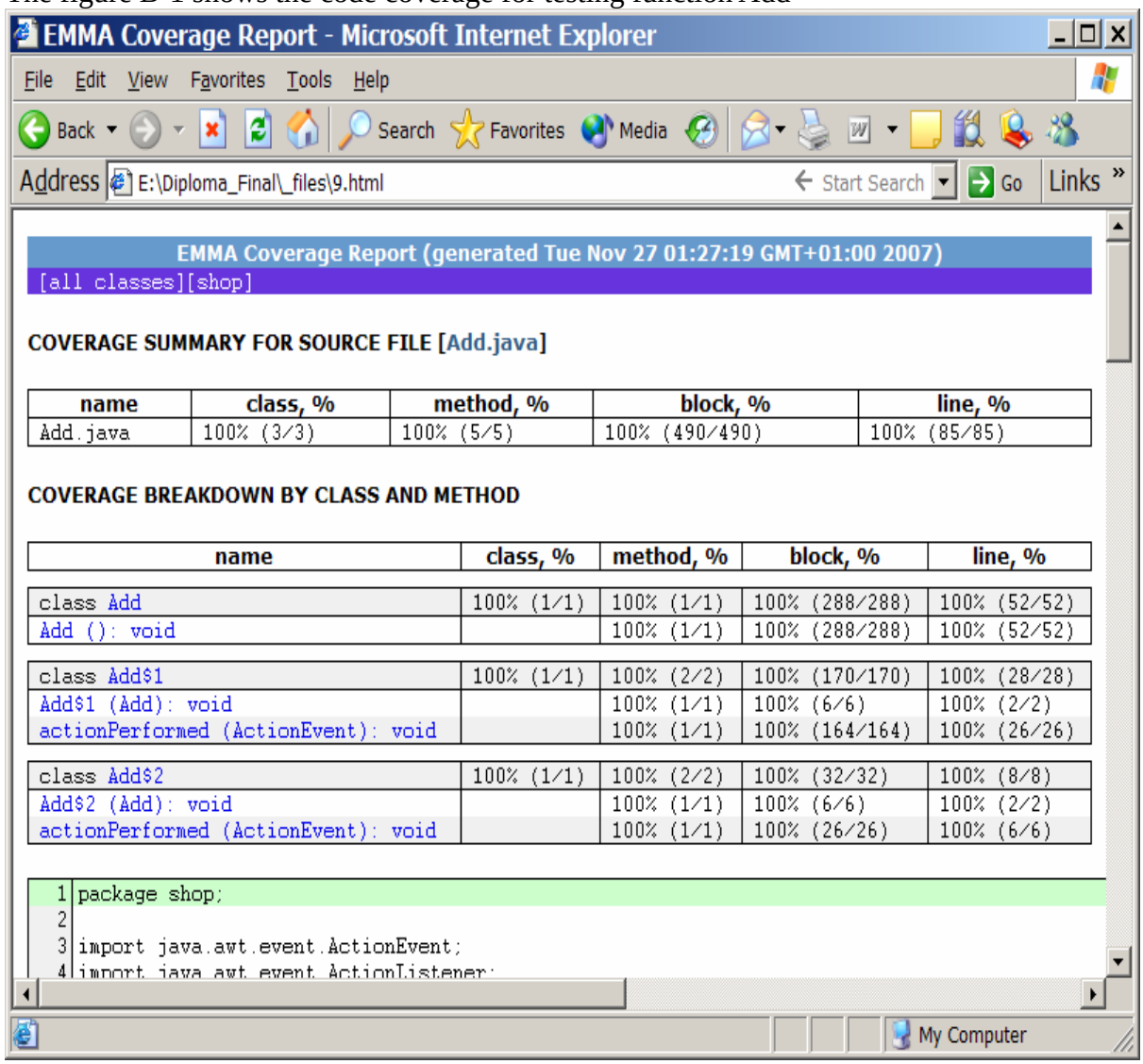

Figure B-1 the code coverage for function Add

The figure B-2 shows the code coverage for testing function remove

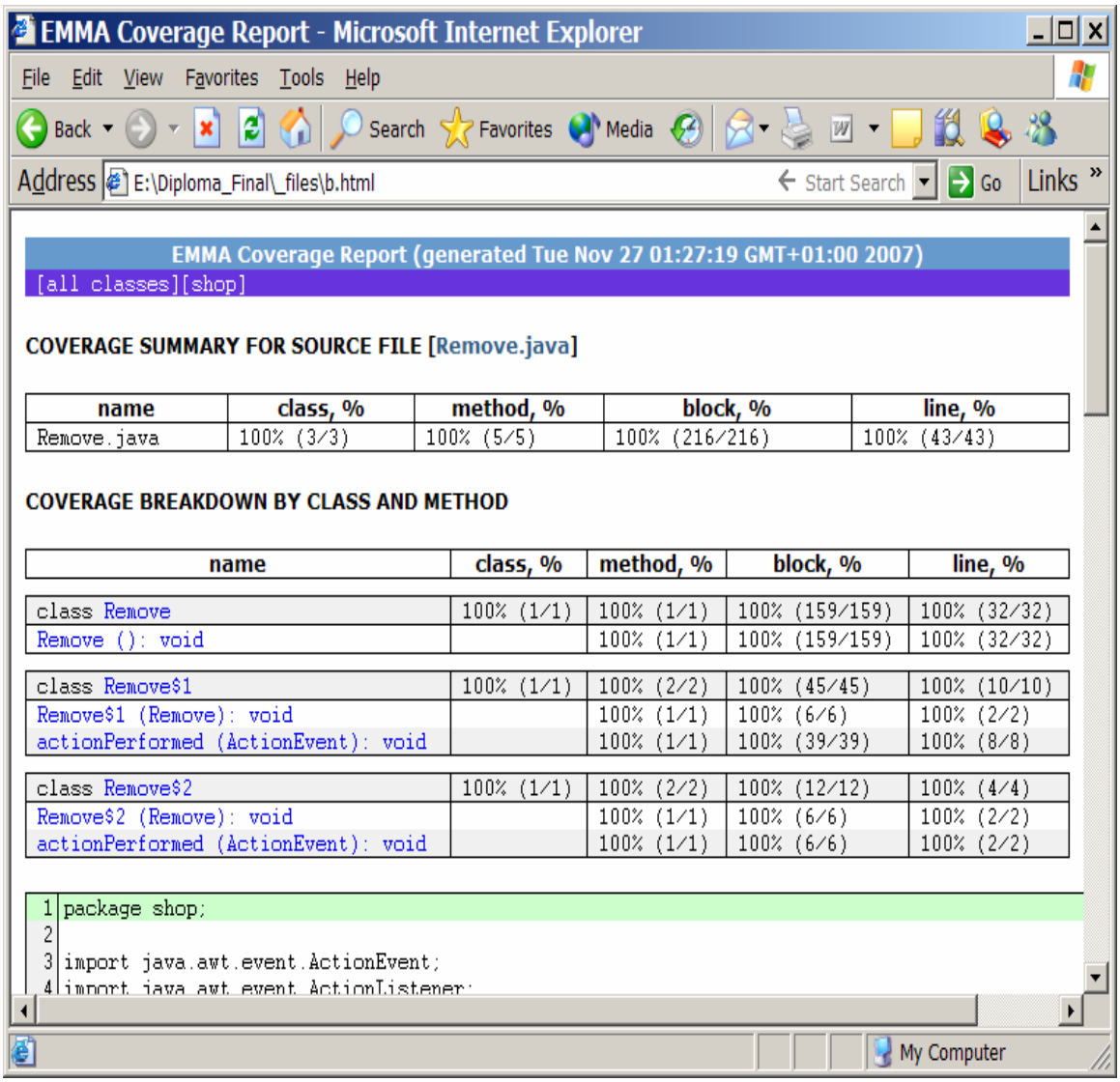

Figure B-2 code coverage for testing function remove

The figure B-3 shows the code coverage for test function of Modify

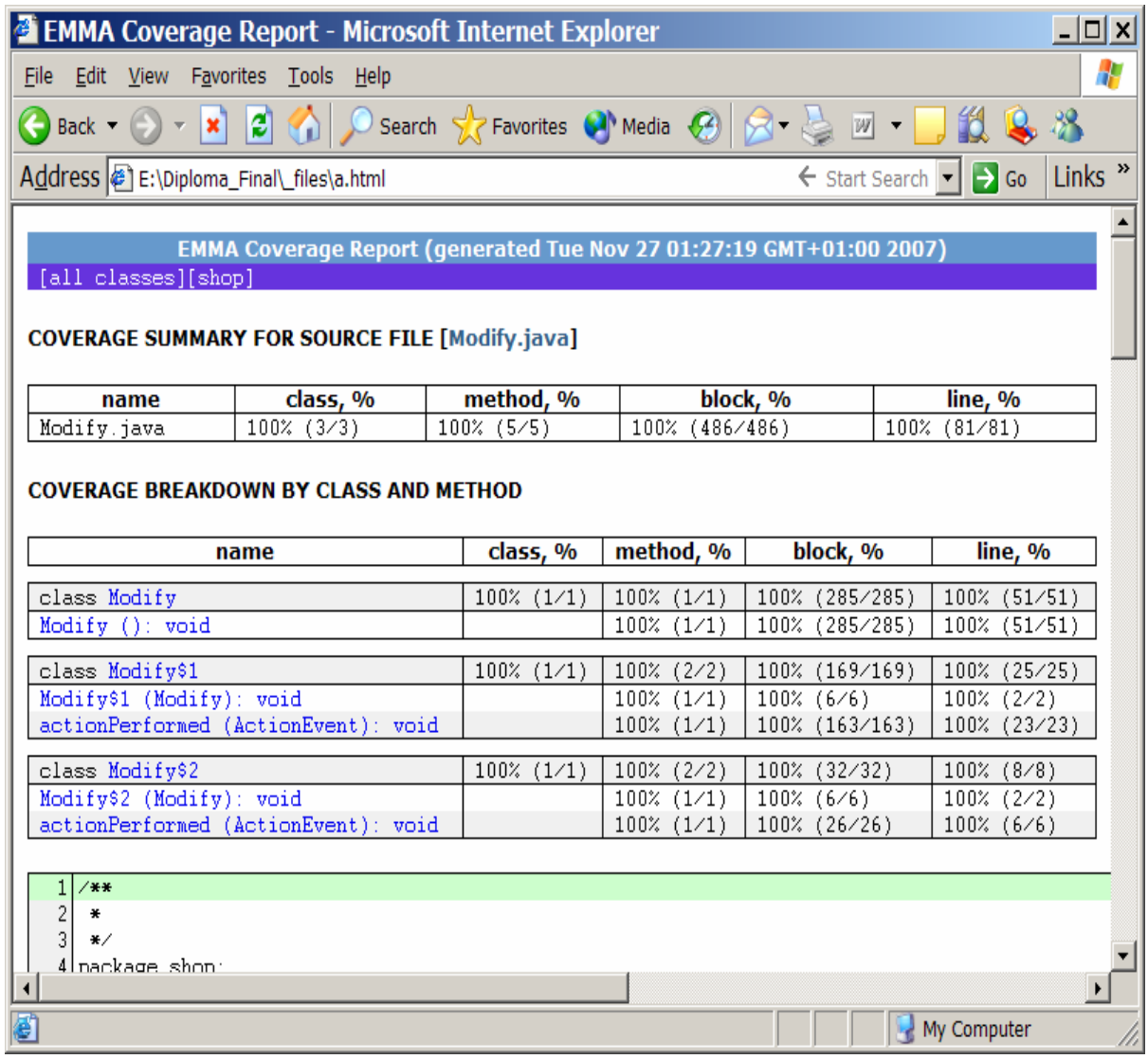

Figure B-3 code coverage for testing function modify

The figure B-4 shows the code coverage for testing the function Show stock and show oos

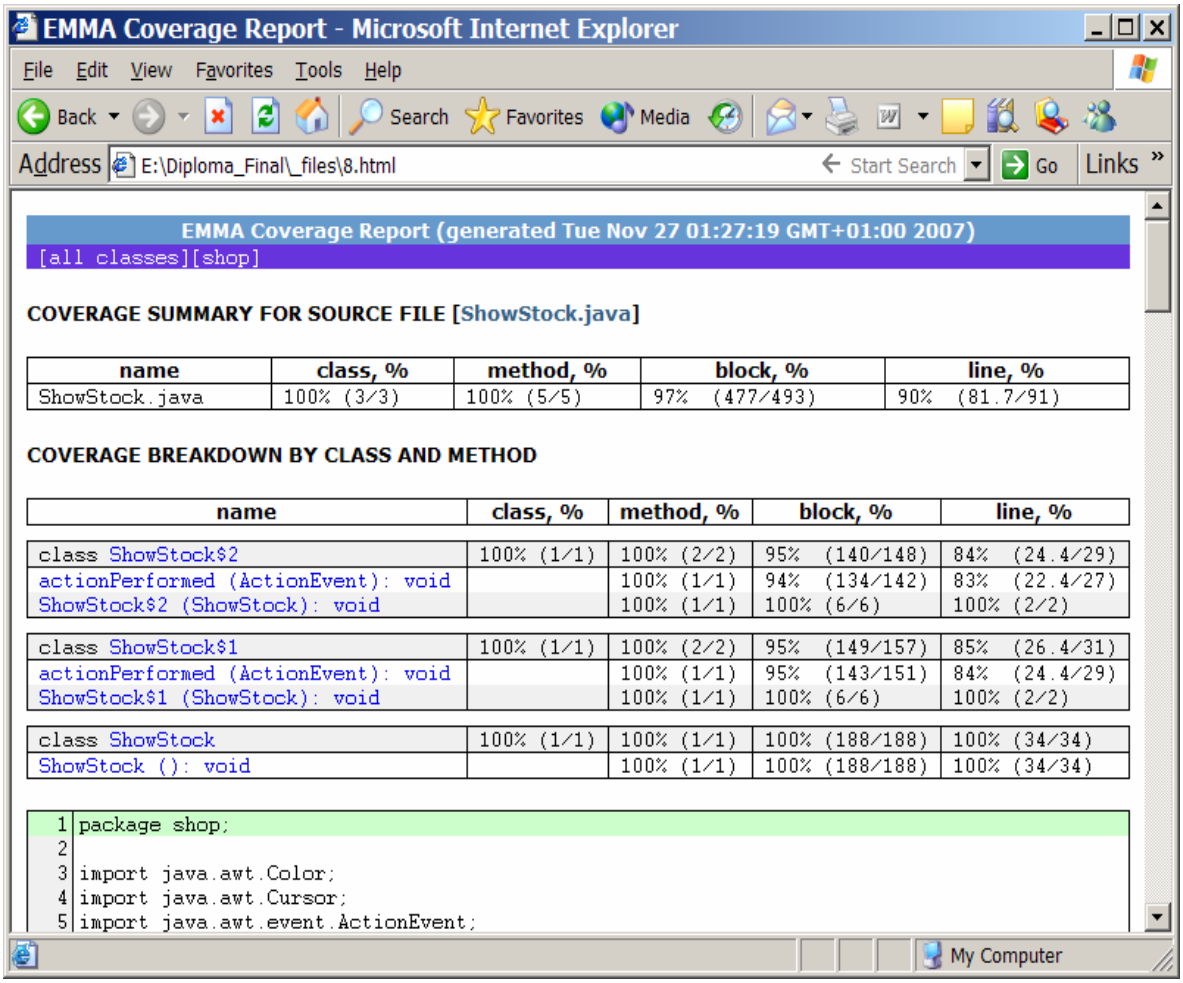

Figure B-4 code coverage for the function show function and show oos