Test-driven development & Enterprise JavaBeans 3.0

B. Sc (Hons) in Internet Technologies 2008

Fabian Piau

Seamus Kelly

Supervisor

Acknowledgments

I would like to thank *Seamus Kelly* my supervisor for his guidance, suggestions and encouragement throughout this project. I would like to thank *my flatmates* and *my parents* for their constant support. And finally, thanks to the *MIAGe of Nantes* and the *Dundalk Institute of technology*. Without their collaboration, it would have been impossible for me to study this year in Ireland in the *Erasmus Program*.

Abstract

Testing is a critical part of good software development. Test-driven development (TDD) also known as Test-first development (TFD), is a technique, associated with Extreme Programming (XP) and Agile Programming methods, in which unit test cases are incrementally written prior to code implementation. Unit testing and selected aspects of Test-driven development can be used to improve learning and encourage emphasis on quality and correctness. Most users of Test-driven development use automated testing tools to facilitate code review and to encourage frequent and thorough regression testing throughout the development. These tools, such as JUnit, are very popular.

The Enterprise JavaBeans (EJB) specification is one of several Java APIs in the Java Platform, Enterprise Edition. EJB is a server-side component architecture that encapsulates the business logic of an application. This technology is relatively new and complex; one of the biggest issues is that testability has not been taken in consideration, at least not until EJB version 3.0.

The purpose of this dissertation is to present an analysis of the Test-driven development method but not in general; this report will focus on using TDD with the latest 3.0 Enterprise JavaBeans specification.

To achieve this goal, my reasoning falls into three parts. First, I need to carry out research into TDD in general and in particular as applied to EJB. Subsequently, I will put TDD into practice with the development of a prototype and sample EJB application. Lastly, I will have a good understanding of TDD and I will be able to give my point of view and a review of this method applied to the EJB 3.0 specification.

Table of contents

List of Figures

Listings

1. Introduction

1.1 Project Aims

The aims of the project are:

- ✓ To carry out an in-depth study of Test-driven development applied to a recent technology: EJB 3.0, a server-side component architecture. I will have to use a variety of resources, e.g. journals, books, the Internet, etc.
- ✓ To develop an EJB application prototype using the Test-driven development method, i.e. using test, code and refactor cycle.
- ✓ To provide a critical analysis of TDD as applied to development using EJB 3.0 specification.

1.2 Project Objectives

In order to realise the aims, I will have to:

- ✓ Research the literature on Test-driven development applied to EJB
- ✓ Write a literature review
- ✓ Make an interim presentation about Test-driven development to explain “where am I going?”
- ✓ Improve my skills in *JAVA Enterprise Edition* (*J5EE*), especially in the EJB 3.0 specification
- ✓ Learn new tools such as *JDeveloper* IDE, *JUnit* and *EasyMock* frameworks
- ✓ Test , write, and refactor part of an EJB application to investigate Test-driven development
- ✓ Present an analysis of Test-driven development
- ✓ Make a final presentation about TDD applied to EJB 3.0
- ✓ Write a report (this one!)

1.3 Report Content

The report will give a detailed explanation of the following topics and will provide answer to the following questions.

- ✓ A detailed presentation of TDD
  - › What are the fundamental basics of TDD?
  - › What is a unit test?
  - › Which main types of unit testing tools can I find?
  - › How to succeed with TDD?
- ✓ The EJB specification
  - › What is the EJB specification?
  - › Why are EJBs difficult to test-drive?
  - › What are the new features in the EJB 3.0 specification?
  - › Why is this last version more suitable for TDD?
- ✓ Put TDD into practice
  - › How to write a set of tests?
  - › What are the main steps to create an EJB 3.0 application using TDD method?
- ✓ Conclusions about TDD applied to EJB
  - › What are the advantages and issues of TDD in general?
  - › What are the problems I have encountered?
  - › What is my personal opinion?

  In this last part, I will try to give an objective point of view.

It is assumed that the readers of this report are quite familiar with *Java Enterprise Edition*.

2. Literature Review

2.1 Introduction

2.1.1 Extreme Programming

In 1999, Extreme Programming (XP) appeared on the horizon and, where it was adopted, everything changed! XP is a programming methodology that places developers at the centre of the process. (Corbett [6]))
In this agile model, there are currently (last revision) 13 primary practices and 11 corollary practices. As Figure 2.1.1 shows (Jeffries, 2001 [17]), Test-driven development plays a main part in the primary practices. Indeed, TDD is a core practice of XP which has become a widely adopted practice. Testing software from the beginning and throughout the entire development cycle is an essential software engineering practice. The Extreme Programming method takes this a step further and recommends an evolutionary approach to design that follows a test-code cycle of continually switching between coding and testing. (Olan, 2003 [8])

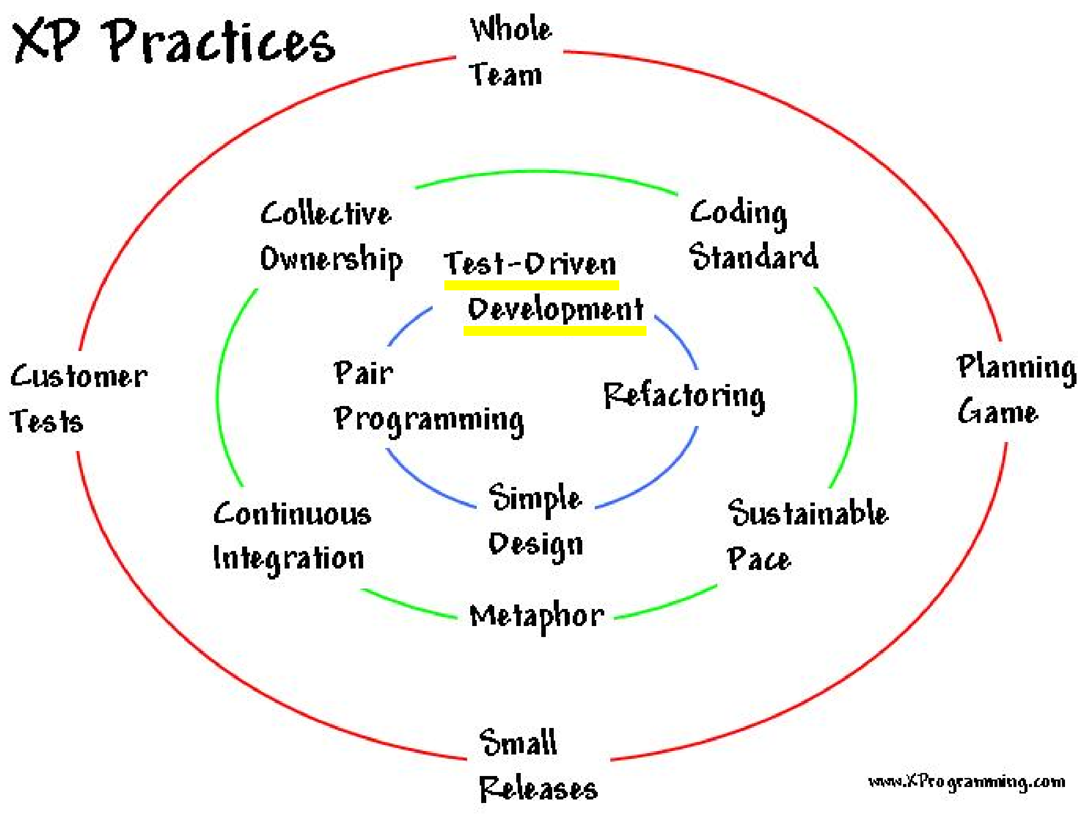


Figure 2.1.1: The 13 primary XP practices

2.1.2 An example to introduce TDD

Consider a convincing example. If you were building a car, you would probably construct each of its many complex components separately. Unless you tested each piece individually before assembly, you would have a lot of trouble to figure out why the car does not run correctly after you put all the components together. For example, is it the battery or the motor which is faulty? Without having evaluated each piece before, you have no way of knowing whether one or more of the pieces was built incorrectly, whether your integration was faulty, or both. You can imagine the amount of time you would waste trying to analyze what was wrong with the car.

If you tested each component by itself before assembly, you would be able to focus your debugging efforts on the integration, i.e. the way you assembled the car. And since you would have confidence in the individual pieces of the car, you would have more confidence in the car as a whole.
Exactly for the same reasons, it is important to unit test software.

2.1.3 Test-driven development

Although software testing has always been used by developers, it was typically performed after the code was designed and written. I think a great number of developers can attest that writing tests after coding is difficult to do and the tests themselves are often omitted when time runs out. Test-driven development attempts to resolve this problem and produce higher quality, well-tested code by “putting the cart before the horse” by writing the tests before developers write the code.

“*Only ever write code to fix a failing test.*”(Lasse, 2007 [1]). “*If you can’t write a test for what you are about to code, then you shouldn’t even be thinking about coding.*” (Chaplin, 2001 [2]). These two sentences are full of sense about principles and objectives of Test-driven development.
Test-driven development was born because of software developers looking for a way to develop software better and faster. (Lasse, 2007 [1])

Indeed, developers are familiar with the experience of making a change to an application, testing it, and releasing it, only to discover that the change they have made has broken something else in the application. In others words, unless you test your application fully, you cannot be sure the changes you make will not affect another part of the system and these tests can take a considerable amount of time.

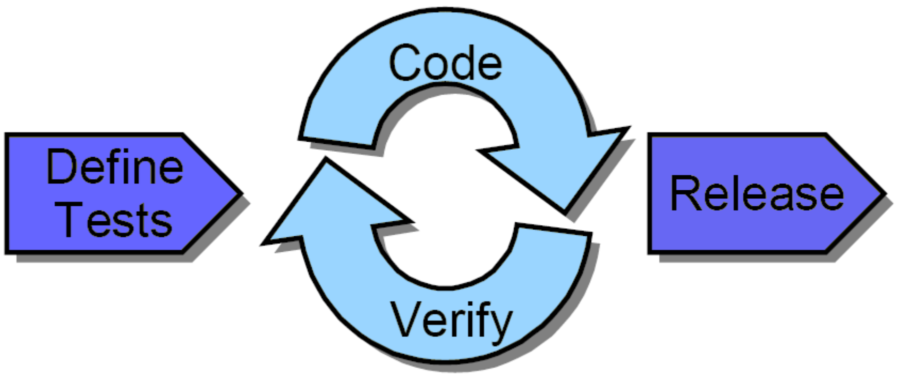


Figure 2.1.3: TDD lifecycle

Test-driven development alters the process of writing code so this change is not only possible, but desirable. It revolves around three basic activities: writing a test, writing code to pass the test, and refactoring the code to remove duplication, make it simpler, more flexible, and easier to understand. Thus, development occurs in rapid iteration cycles between developing, verifying and correcting code. (see on previous page – Figure 2.1.3 (Hohpe *et al.*, 2002 [16]))

Writing tests before writing the implementation class itself might seem curious (it was my first feeling) but it forces the developer to think about exactly what the client needs, the functionalities he requires. This helps the developer avoid writing extra code for functionalities that will be never used.

The cycle "testing, developing, verifying and correcting" is repeated frequently, each time running all the tests in order to ensure that the product is kept in a working state. The long gaps between the design, coding, and testing phases have disappeared. Therefore, the design (and code) actually improves rather than degrades as the project matures. (Stott and Newkirk, 2004 [32])

2.2 Tests are everywhere!

The fundamental idea of Test-driven development is to write tests before writing the code to be tested. As the code is written, the developer will have immediate confirmation of whether or not a new piece of code is completely functional, close to working or going to be a complete disaster.

Thus, first the developer writes a test, and then he writes code to make the test pass. After that, he finds the best possible design for what he has, relying on the existing tests to keep the code from breaking. This approach to building software encourages good design, produces testable code, and avoids over-engineering of the systems because of flawed hypotheses. And all of this is accomplished by the simple act of driving the design with executable tests that guide towards the final implementation. (Lasse, 2007 [1])

An essential aspect of unit testing is to test one feature at time; the developer needs to know exactly what he is testing and where any problems reside. Test code should communicate its intent as simply and clearly as possible.

With traditional testing a successful test finds one or more defects. It is the same with TDD; when a test fails you have made progress because you now know that you need to resolve the problem. More importantly, you have a clear measure of success when the test no longer fails. TDD increases your confidence that your system actually meets the requirements.

An interesting side effect of TDD is that you come close to achieve 100% **coverage** test, i.e. every single line of code is tested; something that traditional testing does not guarantee.

The steps of TDD are shown below with a simple automate (Ambler, 2003 [10]):

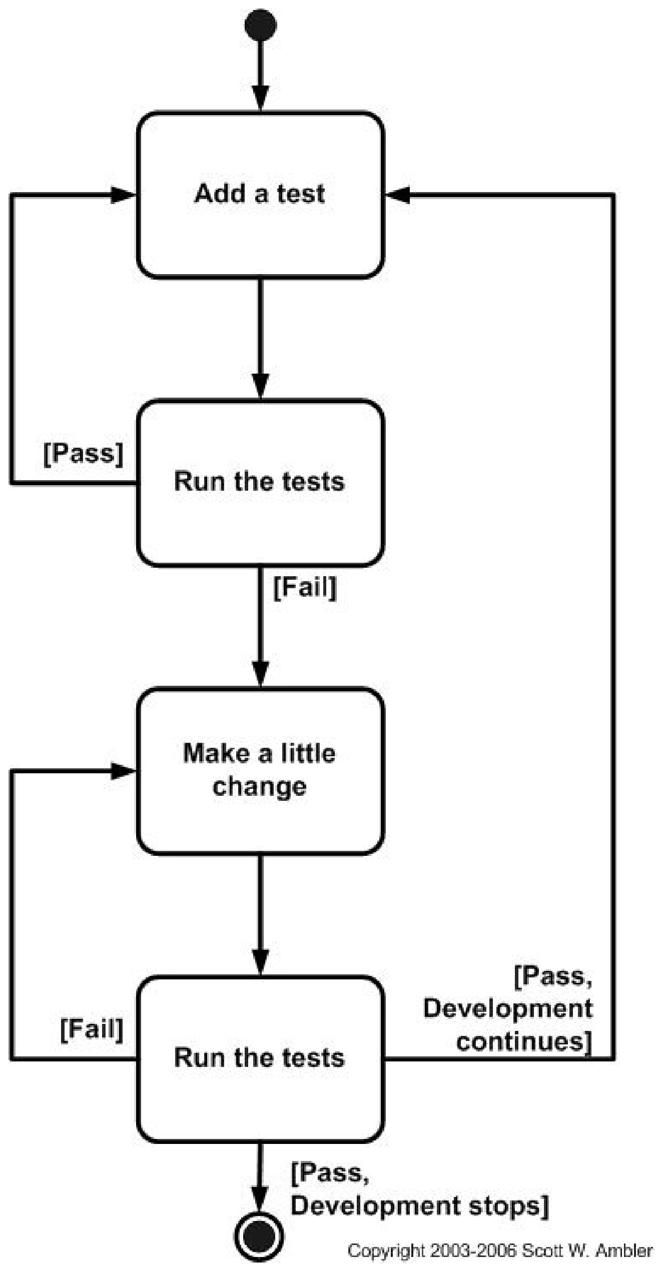

Add a test
Run the tests
[Pass]
[Fail]
Make a little change
Run the tests
[Fail]
[Pass, Development continues]
[Pass, Development stops]
Copyright 2003-2006 Scott W. Ambler

Figure 2.2: The steps of TDD

2.2.1 Add a test

In Test-driven development, each new feature begins with writing a test. The first time, the test must inevitably fail because it is written before the feature has been implemented; it is logical! In order to write a test, the developer must understand the specification and the requirements of the feature clearly.

2.2.2 Run all tests and see the new one fail

This validates that the test is "working correctly" and that the new test does not mistakenly pass without requiring any new code. At first sight, it does not make sense but, running the new test to see it fail the first time is a vital "sanity check".

2.2.3 Write some code

The next step is to write some code that will cause the new test to pass (Wikipedia [15]). 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 more no less.

2.2.4 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.
On the contrary, if not all the test cases pass, the developer has to keep working until they do and do not start on any new features. (Wikipedia [15])

2.2.5 Refactor code

Refactoring is a disciplined way of transforming code from one state or structure to another, removing duplication, and gradually moving the code toward the best possible design. By constantly refactoring, developers grow their code base, evolve their design incrementally, make the code more maintainable, making it simpler, more flexible, and easier to understand; for example, dividing a class or method into parts. (Lasse, 2007 [1])

Refactoring is an essential step in Test-driven development, providing the necessary feedback that allows your design and your code to improve as the product grows. The power of refactoring lies in its ability to reduce the danger inherent in making changes to working code. Any change will potentially introduce errors and thus tests must be repeated often. By re-running the test cases, the developer can be confident that refactoring is not damaging any existing functionality. (Olan, 2003 [8])
It is important to say that refactoring restructures existing code, alters its internal structure but does not change its external behaviour. You can find an example of code refactoring in the appendices. (p. 65)

2.2.6 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 he feels more confident. If the code written to satisfy a test does not work fairly quickly, then the step-size is certainly too big, and maybe the smaller testable steps should be used instead.

NB

Remember that the premise for unit testing is that you write tests to test your application's functionalities. You run your unit tests after you have made any changes to the system. If all tests pass, you can be confident the application still executes as you intended. But, you still have **no guarantee** it will, because it is absolutely possible the change might cause a problem a unit test does not cover.

Lastly, it is important to emphasize that the tests document the code. One can see exactly what the code does by looking at the tests. Of course, tests are not sufficient for documentation but they form an important part of it.

2.3 How to succeed with TDD

Here are some best practises I found on diverse websites which will be useful during the development of my application.

2.3.1 Best practices…

- ✓ Only work on one test at a time, do not write too many tests and then try to pass them all in one time.
- ✓ Do not be afraid of doing something trivial to make the test work.
- ✓ Never write a test that succeeds the first time.
- ✓ Make test assertions self-explanatory so other developers can easily see what the code is supposed to be doing just by reading the tests.
- ✓ Make the structure of your test projects follow the structure of the projects they are testing so developers can easily navigate from the tests to the code they are testing and vice-versa.
- ✓ Keep the tests small:
  - › More than 15 minutes on the same test is probably too long. Focus will be lost.
  - › If a test looks complicated, think of ways to break it down. If a test involves several methods or classes then either:
    - Write tests for those first and work your way up to the bigger test.
    - Fake them to pass this test and then work your way down to implement each of them one test at a time.
- ✓ If you are working in team, integrate your code changes every few unit tests to make sure there are not any nasty surprises waiting for you in someone else's changes.
- ✓ If you are working alone, end every day with one broken test to help you quickly refocus the next morning.
- ✓ If you are working in a team, never ever leave with broken tests!
- ✓ Take regular breaks and work reasonable hours. TDD is supposed to set a pace that can be sustained indefinitely.

2.3.2 …For continuous integration

- ✓ Before integrating your code changes, inform all team members. Until you have a successful build on the integration server, nobody else should consider integrating their changes. An integration server acts as a monitor to the repository. Every time a commit against the repository finishes, the server automatically checks out the sources onto the integration machine, initiates a build, and notifies the committer (Fowler, 2006 [11])
- ✓ Get any changed files that have been checked in to your team's repository since you last checked out the code and merge those with your changes.
- ✓ Run all of the tests.
- ✓ Do not integrate if any of the tests fail.
- ✓ If all tests pass then check in your code changes.
- ✓ Do a build with the latest code in the repository on the integration server and run all of the tests against that build.
- ✓ If all the tests pass then you have a successful build and other developers can start to integrate.
- ✓ If any tests fail then you have a broken build, which means nobody else can start integrating. Your team's priority is now to find out why the test(s) fail and fix the problem.

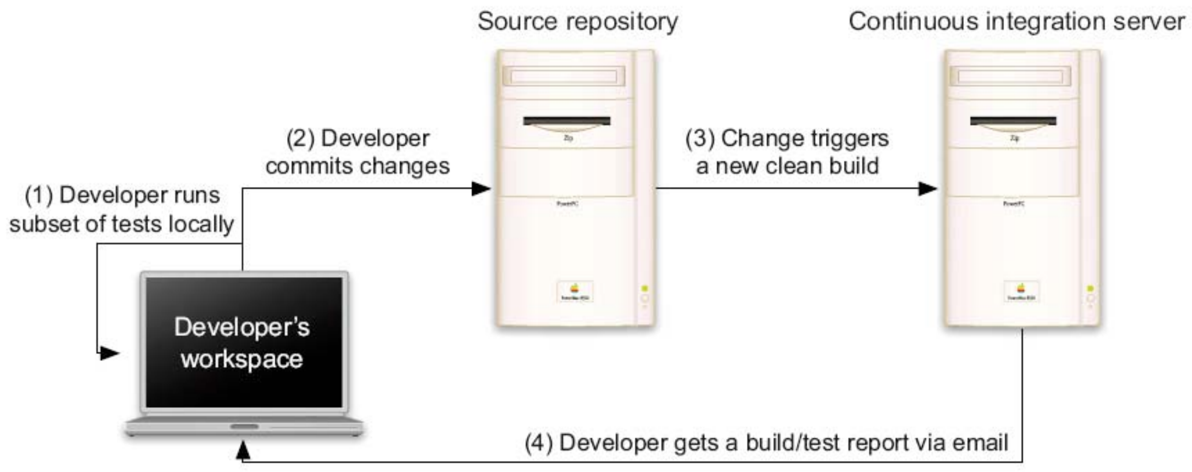

Source repository
Continuous integration server
(2) Developer commits changes
(3) Change triggers a new clean build
(1) Developer runs subset of tests locally
Developer's workspace
(4) Developer gets a build/test report via email

Figure 2.3.2: Typical enterprise scenario [1]

Most of this advice concerns work in a team. For my project, I will be "alone" to program my application, nevertheless, this advice is very interesting and it is what happens in reality. Indeed, when you work in a company, you rarely have the opportunity to code alone or it is only on small projects.

2.3.3 Characteristics of a good unit test

Following research, I have found the following characteristics to create a good unit test. A good unit test:

- ✓ Runs fast (i.e. short setups, run times) because if the tests are slow, they will not be run often, as time is very important for a company.
- ✓ Uses data that makes them easy to read and to understand and tries to use real data. (e.g. copies of production data)
- ✓ Separates or simulates environmental dependencies such as databases, file systems, networks, etc. Tests that depend on such dependencies will not run fast, and a failure does not give meaningful feedback about what the problem actually is.
- ✓ Is very limited in scope. If the test fails, it is obvious where to look for the problem. Use few *Assert* calls (see next part on the next page "The xUnit family") so that the offending code is obvious. It is important to only test one thing in a single test.
- ✓ Runs and passes in isolation. If the tests require special environmental setup or fail unexpectedly, then they are not good unit tests. Change them for simplicity and reliability. Tests should run and pass on any machine.
- ✓ Often uses stubs and mock objects. If the code being tested typically calls out to a database or file system, these dependencies must be simulated, or mocked. These dependencies will ordinarily be abstracted away by using interfaces. Mock Objects will be addressed in more details afterwards (see p. 19).
- ✓ Clearly reveals its intention. Another developer can look at the test and understand what is expected of the production code.

2.4 The xUnit "family"

Putting TDD into practice does not have to be daunting. Many resources exist to help you. The tests used in TDD are automated unit tests, designed to validate that a unit or module of code is working properly (Corbett [6]). To support the creation of these automated tests, unit testing tools exist for almost every programming language imaginable. The most famous software is the xUnit family. By family, I mean set of versions for most programming languages (*CppUnit* for *C++*, *JUnit* for *Java*, *NUnit* for *.NET*, *PyUnit* for *Python*, *VBUnit* for *Visual Basic*, etc.). In this part, I will talk about *JUnit* because it is the tool I will use to develop my application. You should be aware that other tools exist outside the xUnit family, such as *TestNG*, *Cactus*, etc.

2.4.1 Red or Green bar?

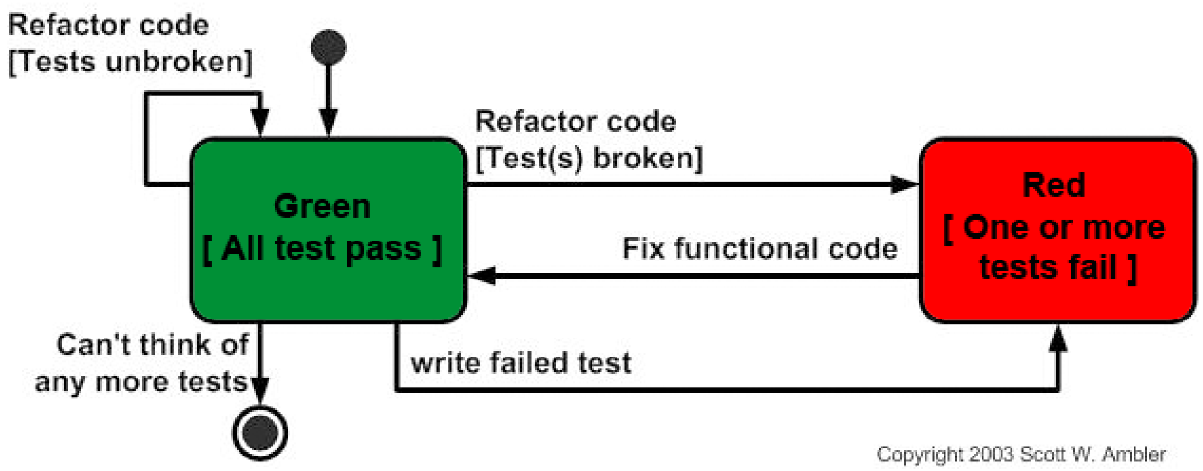

Refactor code [Tests unbroken]
Green [All test pass]
Refactor code [Test(s) broken]
Red [One or more tests fail]
Fix functional code
Can't think of any more tests
write failed test
Copyright 2003 Scott W. Ambler

Figure 2.4.1: Red / Green bar [10]

Each time it runs a test suite, *JUnit* displays a bar green to indicate passed tests and displays the bar red to indicate a failed test. The slogan on the *JUnit* official website is "*Keep the bar green to keep the code clean.*"

This software is freely available for download on the web, so the cost is not a problem. However, downloading, installing, and learning to use these tools can be non-trivial for developers, especially if they are relatively new to computing.

2.4.2 Tools provided by JUnit

JUnit is the primary *Java* test framework for developers. Written by Erich Gamma and Kent Beck, it is distributed as an open source project and defines a common language for writing and running repeatable tests. *JUnit* uses reflection to examine the tests and code under tests. Reflection is the process by which a computer program can observe and modify its own structure and behaviour (Wikipedia [15]). *JUnit* uses reflection to automatically locate all *testXXX()* methods in your test case, adding them to a suite of tests. It then runs all tests in this suite. This allows *JUnit* to execute any method of any class and examine the results. With this framework, all developers on the project know how to write and execute tests, and interpret test results using the following tools:

- ✓ `TestCase`: Abstract class for implementing a basic unit test
- ✓ `TestSuite`: Composite class for organizing and running groups of tests
- ✓ `Assertions`: For testing expected results (`assertNotNull(..)`, `assertEquals(..)`, `assertSame(..)`, etc.) , assertions are expressions that describe what must be true when some action has been executed
- ✓ `TestRunner`: Application - Graphical (red/green bar) and text-based test runners
- ✓ `Failure`: Indicates a checked test assertion failed (i.e., `assertNotNull(..)` returned false)
- ✓ `Error`: Indicates an unexpected exception or setup failure that stopped the test
- ✓ `Setup`: Code that is run before every test method is executed (e.g., logging in as a particular user)
- ✓ `Teardown`: Code that is run after every test method has finished (e.g., deleting rows from a table that were inserted during the test)

In this part, I have presented *JUnit* 3.8 which actually is not the latest version of *JUnit* (the current version is 4.0) but the version I will use to create my sample application.

JUnit is widely accepted as a standard for unit testing in *Java*. Many of the available testing products on the market are either based on, or extend, *JUnit*. Also, many IDEs currently have built-in support for *JUnit*. To make my application, I will use *JDeveloper*, an IDE from *Oracle* which perfectly integrates the *JUnit* framework.

2.4.3 Summary

A TDD project might generate thousands, or even hundreds of thousands of tests. Given that everyone on the team will be going through the "write a test, fix the code, refactor" cycle frequently, it is essential that the programmer both writes and runs tests efficiently (Stott and Newkirk, 2004 [32]). A test framework like *JUnit* is designed to help him to do this. It allows arranging the test cases into individual projects that can be loaded into *JUnit* in the same way as creating projects in an IDE. It is also possible to display all the project's test cases in a hierarchy, run tests individually or as a suite, and see the result of their execution as pass (green) or fail (red) with detailed information about each failure (see Figure 3.3-3 p. 30)

2.5 Mock Objects

One of the biggest challenges developers have to face when they write units tests is to make sure that each test is only testing one thing. It is very common to have one object dependent on other objects to do its work. In this case, if the developer writes a test for a method, he will test not only the code in that method, but also the code in the other classes. This is a problem. To avoid this situation, "Mock Objects" are used.

A Mock Object emulates a real class and helps test expectations about how that class is used. In other words, it is an object that pretends to be a particular type, but is really just a "sink", a fake object that replaces a real one, implementing the methods that have been called on it. (Miller, 2002 [34])

Using such an object is necessary because normal unit testing is difficult from outside. "Outside" means that the code under development relies on a database, a web service or any other external process or service. Thus, domain code is replaced with dummy implementations that emulate real code. These Mock Objects are passed to the target domain code which they test from inside.

Why use them?

- ✓ Avoid complex setup
- ✓ Avoid external dependencies and having to write the code in a certain order
- ✓ Reduce coupling
- ✓ Keep tests fast (e.g. simulating a database)
- ✓ Test object interactions
- ✓ Promote interface based design
- ✓ Ensure tests are durable

Frameworks such as *JMock* or *EasyMock* exist to make the process of creating and using mock objects easier.

I will use Mock Objects to write my sample application. EJB is a managed object–based technology; i.e. all the objects are managed by the container. The EJB specification will be addressed in more detail in the following part "Enterprise JavaBeans".

This is just a short introduction to Mock Objects because I have not tried to use them yet. I will talk about these Mock Objects in more detail when I describe my application.

2.6 Enterprise JavaBeans

Enterprise JavaBeans (EJB) technology is the server-side component architecture for *Java Platform, Enterprise Edition (Java EE)*. EJB technology enables rapid and simplified development of distributed, transactional, secure and portable applications based on *Java* technology (Sun Microsystems [35]). But before introducing EJB in more detail, I briefly present *Java EE* and its multitiered architecture.

2.6.1 Java Enterprise Edition 5

The aim of the *Java EE 5* platform is to provide developers with a powerful set of APIs while reducing development time, reducing application complexity, and improving application performance (Sun Microsystems [35]). *Java EE* provides services to developers to allow them to focus and concentrate on the core area of developing software.

To develop my sample application using *Java Standard Edition (Java SE)* many hours of development time would be consumed creating classes to handle security, transactions and concurrency control. I avoid all that by simply implementing the *Java EE* model.

The *Java EE platform* uses a distributed multitiered application model for enterprise applications. Application logic is divided into components according to function, and the various application components that make up a *Java EE* application are installed on different machines depending on the tier in the multitiered *Java EE* environment to which the application component belongs. (see Figure 2.6.1 below, an example of three-tiered application (Sun Microsystems [35]))

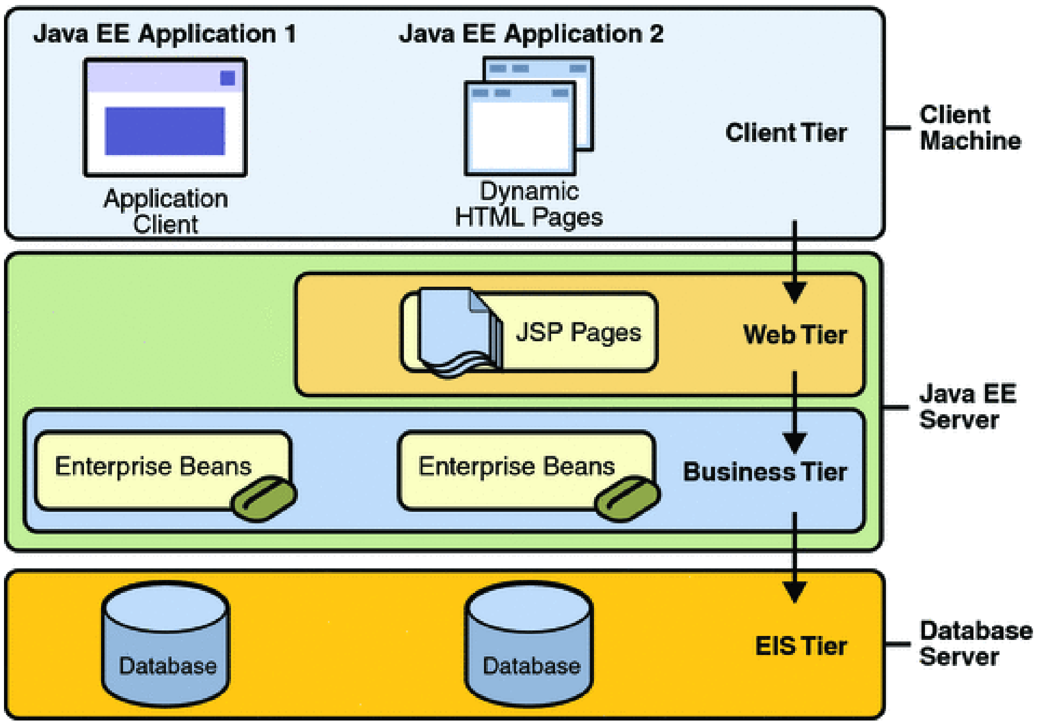


Figure 2.6.1: JEE multitiered architecture

Java EE multitiered applications are generally considered to be three-tiered applications because they are distributed over three locations: client machines, the *Java EE* server machine, and the database or legacy machines at the back end. (Sun Microsystems [35])

The EJB 3.0 specification defines the new simplified EJB API (set of software calls and routines) targeted at ease of development. It also includes the new *Java* Persistence API for the management of persistence and object/relational mapping with *Java EE* and *Java SE*. The purpose of this latest version is to improve the EJB architecture by reducing its complexity from the developer's point of view.

In this subsection, I will first outline some of the limitations of the old version EJB 2.1. Next I will briefly discuss testing EJBs, and finally, I will describe how EJB 3.0 addresses these difficulties by describing some of the significant changes.

2.6.2 The limitations of EJB 2.1

Developing EJBs with EJB 2.1 has not been the easiest thing to do. The reasons are easy to find (Ranganathan and Pareek, 2006 [18]):

- ✓ To create a single EJB you need to create a multitude of XML deployment descriptors.
- ✓ A set of three source files must be created.
- ✓ Multiple callback methods must be implemented that usually are not used.
- ✓ You have to throw and catch several types of unnecessary exceptions.
- ✓ Yet another complaint is that the EJBs are very difficult to test outside the context of the container since components like container-managed entity beans are abstract classes (read below for more details).
- ✓ Finally, EJB-QL (Enterprise JavaBeans Query Language) in its current form is limited in functionality and difficult to use. These limitations force developers to use straight JDBC and SQL, or to use other persistence frameworks (*Spring, Hibernate*).

2.6.3 The problem with unit testing EJBs 2.1

Recall that Unit tests run best when they run individually, in isolation, and quickly (see “Characteristics of a good unit test” p. 16). A test case typically constructs the objects it is testing. However, sometimes the object being tested is dependent on the behaviour of other objects. This is the case with EJBs which are distributed components that are deployed and executed within an EJB container. Because EJBs require several resources provided by the J2EE server to perform their desired tasks, they cannot be easily tested outside the container. Here are a few more reasons why such testing is difficult (Panda, 2004 [20]):

- ✓ EJBs are not pure *Java* classes and they have lifecycle methods.
- ✓ EJB methods cannot be invoked directly and require JNDI (Java Naming and Directory Interface) to look up and create instances.
- ✓ EJB requires services such as transactions and datasources for execution.

2.6.4 Main useful improvements in EJB 3.0 for TDD

Metadata annotations

The configuration of XML deployment descriptors was a major difficulty in the path to simplifying development of EJBs. Therefore one of the primary goals of the EJB 3.0 specification was to shield the developer from having to work with XML files. This is accomplished by the use of metadata annotations. Note that annotations are a feature introduced in *Java EE 1.5* specification and later. From the developer's point of view, annotations are modifiers like public/private and can be used in classes, fields, or methods. Developers may still use XML if desired or can combine XML and annotations. Indeed, some developers prefer XML or are more comfortable with it. In the next chapters, I will give you some examples using annotations.

POJO programming model

Another big improvement is the use of POJO (Plain Old Java Object or Regular Java objects) programming model. To write tests, programmers typically start by instantiating the object under test. Then, they populate the object with any dependencies it requires, often using mock implementations rather than the real thing. Finally, they invoke the functionality they want to test, and perform assertions on the object's and the collaborators' states to verify correct behaviour.

In the case of managed objects used in EJB 2.1, objects are, by definition, managed by someone else, the container. Programmers typically do not instantiate the object themselves directly; rather, they ask the container to provide a reference to their managed object. Figure 2.6.4 (Lasse, 2007 [1]) illustrates this difference between managed and regular Java objects.

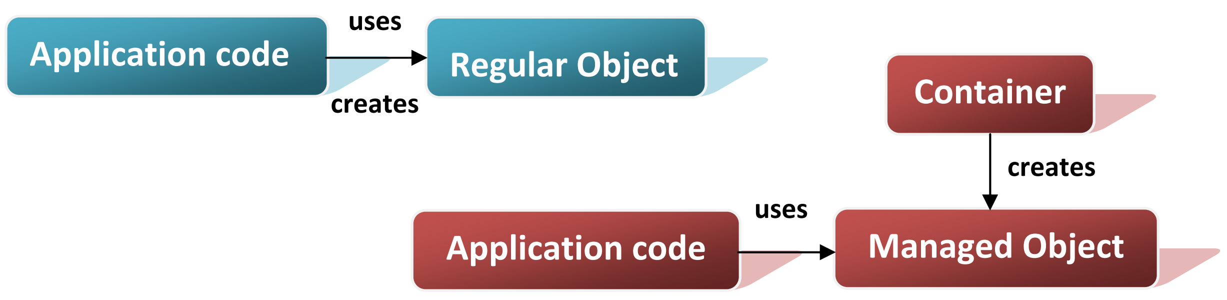


Figure 2.6.4: Managed Object/Regular Object distinction

For instance, with EJB 3.0, a stateless session bean is complete in itself. Interfaces are optional for entity beans and required for session beans and message-driven beans. However, that does not mean that you have to define an interface for your session bean or message-driven bean. If you do not implement an interface, a bean interface will be generated for you. The type of generated interface, either local or remote, is dependent on the annotation you used in the bean class. (Ranganathan and Pareek, 2006 [18])

2.6.5 The Model View Controller architecture

2.6.5.1 General description

The Model View Controller (MVC) is a commonly used and powerful architecture for multitiered applications. It is a way of breaking an application into three parts: the model, the view, and the controller. It separates the code that handles business logic from the code that controls presentation and user interfaces.

Model

The model is the information (data) that the application is manipulating and the rules used to manipulate it.

View

The view implements the visual display of the model. It corresponds to elements of the user interface such as text, checkbox, lists, buttons, etc.

Controller

The controller receives all the input events and manipulations from the user (e.g. mouse and keyboard inputs) and translates them into possible changes on the model.

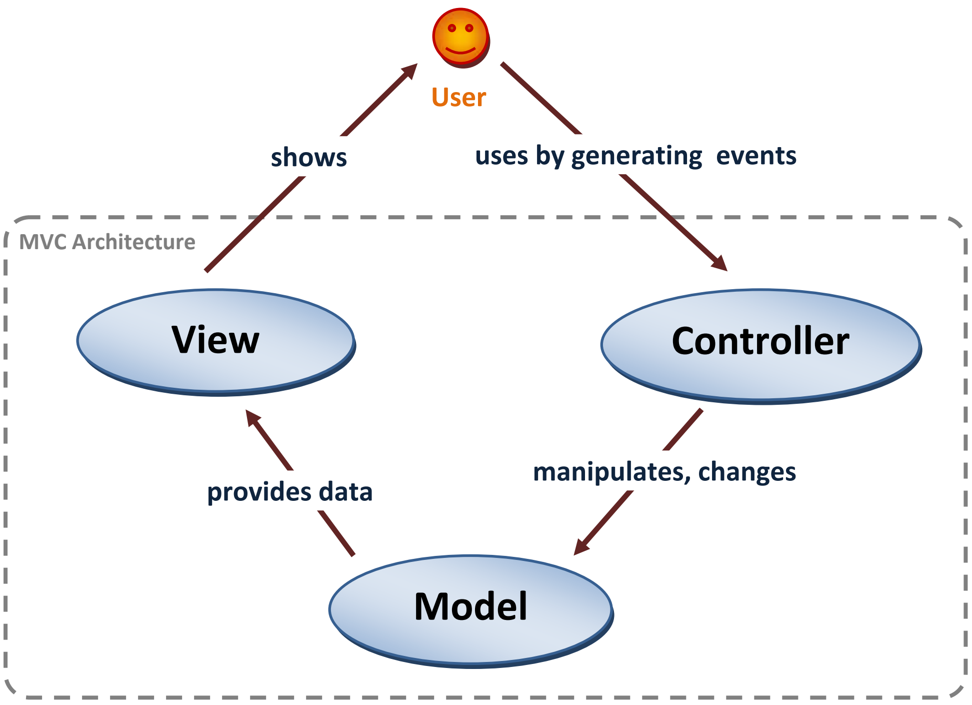


Figure 2.6.5.1: Model View Controller architecture

MVC separates the maintenance of the domain model (the Model), the presentation of the model (the View), and the interpretation of user input (the Controller). As a result it is easier to modify the visual appearance of the application or the underlying business rules without affecting the other.

2.6.5.2 JavaServer Faces

This part will help you to understand the basic concepts of *JavaServer Faces* (JSF). JSF is based on the MVC architecture mentioned previously and is part of the *Java EE standard.*

Model

The model is commonly represented by entity beans, and the client accesses these entities through session beans.

View

The view in a *Java EE* application may consist of a large number of *JavaServer Pages*.

Controller

The controller in a *Java EE* application may be implemented by the *JavaServer Faces* servlet (or called *Faces Servlet*), which acts as the front controller to the application. A front controller servlet is an object that receives all requests and forwards the request for processing before sending the response to the appropriate JSP. (see Figure 2.6.5.2 below)

From Wikipedia, here is a definition of *JavaServer Faces*:

"JSF is a Java-based Web application framework intended to simplify development of user interfaces for *Java EE* applications [...] JSF uses a component-based approach. The state of user interface components is saved when the client requests a new page and then is restored when the request is returned. JSF uses *JavaServer Pages* (JSP) for its display technology."

The following figure gives an overview of the different components in the Web Tier (also called Presentation Tier). The Client Tier does not appear and I do not describe all components in each tier because Figure 2.6.1 already mentioned it.

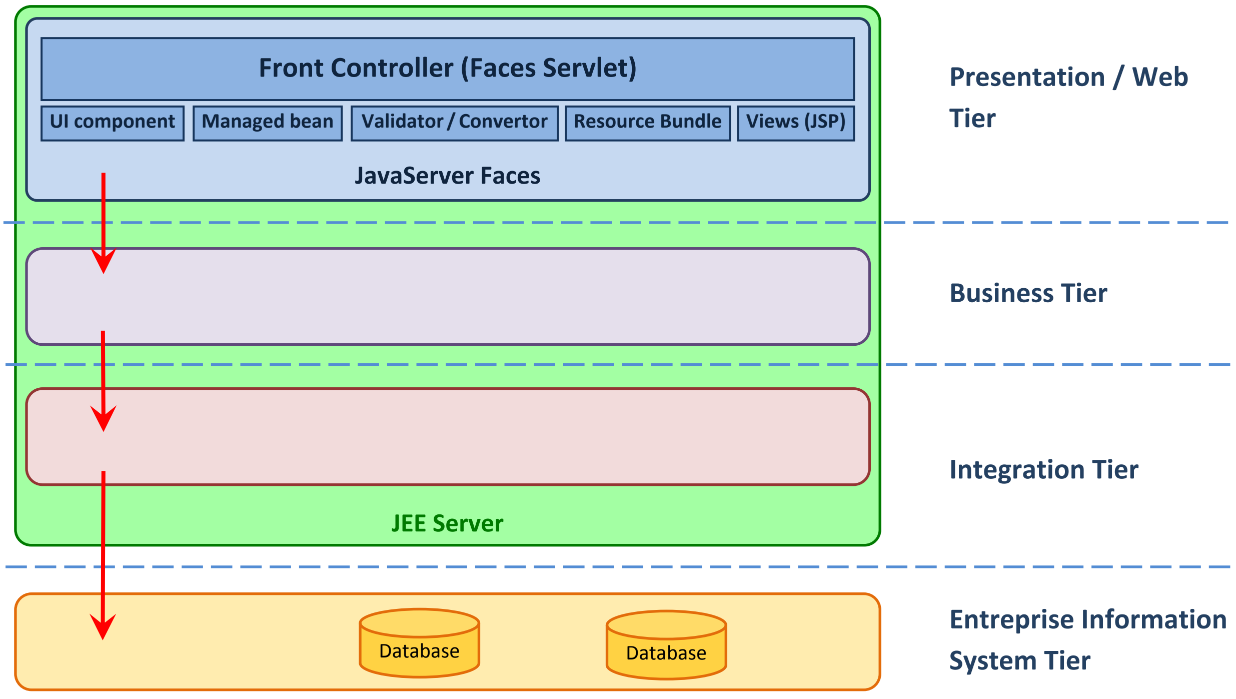


Figure 2.6.5.2: JSF in the Presentation Tier

I will describe JSF and its components in more detail in chapter 6 when describing the implementation of my sample application (see p. 48).

2.7 Conclusion

TDD is a way of programming that encourages good design and is a disciplined process that helps developers to avoid programming errors. TDD does this by writing small and automated tests, which eventually build up a very effective alarm system for protecting the code from regression.

Starting a program by thinking about how to test it might seem unnatural at first, but it has a certain satisfying quality to it. As soon as a new feature is implemented, a test is marked as a new success. It is good for motivation and reminds you when you forget to implement a feature.

I have considered the fundamental issues of EJB testing:

- The problem of objects which are too tightly coupled
- The problem of managed objects which require the services provided by the container (the container is too heavy to be included as part of the unit tests. Indeed, the fact that the unit tests run quickly is very important).

Until version 3 of EJB, testability was very difficult for test-driven developers.

Agile programming and Test-driven development are emerging development techniques that are generating enthusiasm in the software community. The old EJB specification was not suitable for these techniques. Fortunately, the EJB 3.0 specification greatly changes the programming model. Testing of components outside the application server or container is now possible, particularly with the new dependency injection feature. (see p. 42)

3. Proposed Solution

The aim of this chapter is the practical application of TDD. Indeed, the only way to really understand Test-driven development is to put it into practice. To reach this objective, I will create a sample application using TTD method and unit testing tools.

3.1 Prototype application

The following application will be the basis for my unit tests. The objective is not to develop a complete application ready for use but to learn and present a new development technique.

Thus I will only design and implement some of the main features with a simple interface. Thus, I describe my application as a "prototype" or "sample" in this report.

Test-driven development completely changes the habits of programmers and it will be disconcerting and difficult initially.

My application will enable a user to manage rental accommodations, i.e. it will enable a manager to manage the accommodation operation of his customers. This application will assist managers to rent out their apartments in particular residences, manage customers, provide useful business information, etc.

The manager will have access to several functions (after successful login) such as:

- ✓ Rent out a room to a single person, a couple or a family
- ✓ Manage prices of the different kind of residences (depending on the number of rooms, bathrooms, location, etc.)
- ✓ Amend details of a tenant
- ✓ List of tenants (all tenants, by residence, by floor, by apartment, etc.)
- ✓ List of tenants who have not paid for the current month
- ✓ List of regular tenants
- ✓ Number of empty rooms in a residence
- ✓ Money management (rent, employees wages)
- ✓ Modify date of renovation of a residence
- ✓ Calculate profits for the month, year…
- ✓ etc.

For the moment, this is just a starting point... I will certainly add, remove or modify some functions in the future. For instance, if I have time, I will implement a home page where tenants can logon and consult their personal accounts.

3.2 Required tools

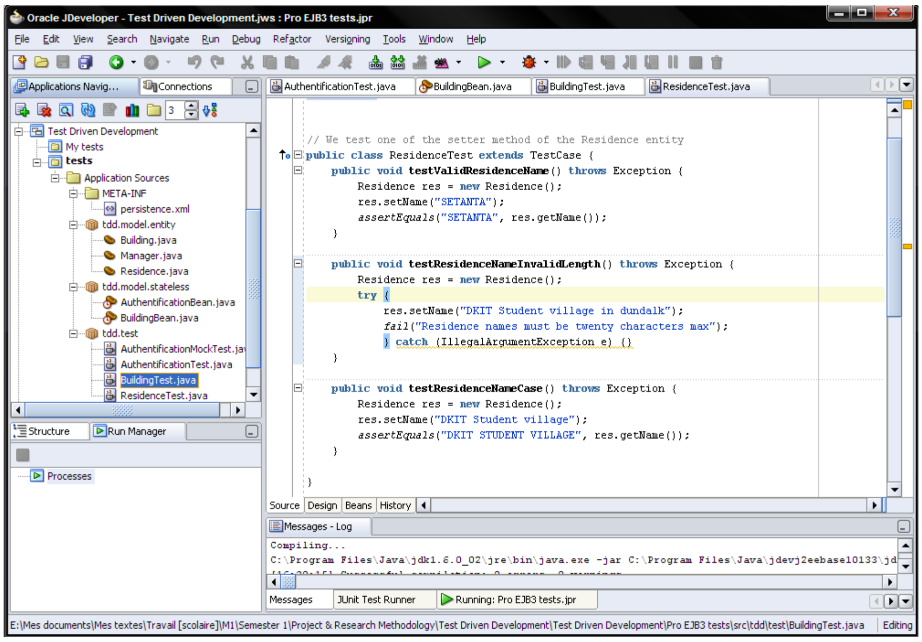
This application will be developed with *Enterprise JavaBeans* 3.0 (EJB 3.0) with the IDE from *Oracle*: *JDeveloper* version 10.1.3.3.0. I will use *JavaServer Faces* (JSF) as the server side user interface component of my application. Recall that EJB is one of the several *Java* APIs in the *Java Platform, Enterprise Edition*. EJB is a server-side component that encapsulates the business logic of an application.

Accordingly, the EJB specification details how an application server provides (Wikipedia [15]):

- ✓ Persistence
- ✓ Transaction processing
- ✓ Concurrency control
- ✓ Events using Java Message Service
- ✓ Naming and directory services (JNDI)
- ✓ Security
- ✓ Deployment of software components in an application server
- ✓ Remote procedure calls using RMI-IIOP
- ✓ Exposing business methods as Web Services

I have chosen *Java* as the programming language and EJB 3.0 as the server-side technology for my prototype application because this version is more suitable for TDD than EJB2.x as mentioned previously. TDD may be used with others languages and technologies. Moreover, as mentioned in the Literature Review, I need more tools to practise TDD. Thus, to finalize this application, I will use a variety of tools integrated with *JDeveloper*: *JUnit*, *EJBUnit*, *EasyMock*, and *JMock*. At this point in time, I do not know yet which tools I will use. I will investigate various tools and choose the most appropriate.

Lastly, I will need a database to save my data about the managers, customers and accommodations. The database I will use is *Oracle Database 10g Express Edition* (Oracle Database XE). It is the free database from *Oracle*. The free version has some limitations as expected from a free product but it will be suitable for my application.



```
// We test one of the setter method of the Residence entity
public class ResidenceTest extends TestCase {
    public void testValidResidenceName() throws Exception {
        Residence res = new Residence();
        res.setName("SETANTA");
        assertEquals("SETANTA", res.getName());
    }

    public void testResidenceNameInvalidLength() throws Exception {
        Residence res = new Residence();
        try {
            res.setName("DKIT Student village in dundalk");
            fail("Residence names must be twenty characters max");
        } catch (IllegalArgumentException e) {}
    }

    public void testResidenceNameCase() throws Exception {
        Residence res = new Residence();
        res.setName("DKIT Student village");
        assertEquals("DKIT STUDENT VILLAGE", res.getName());
    }

}
```

```
Compiling...
C:\Program Files\Java\jdk1.6.0_02\jre\bin\java.exe -jar C:\Program Files\Java\jdevj2eebase10133\jd
```

```
E:\Mes documents\Mes textes\Travail [scolaire]\M1\Semester 1\Project & Research Methodology\Test Driven Development\Test Driven Development\Pro EJB3 tests\src\tdd\test\BuildingTest.java
```

Figure 3.2: JDeveloper IDE running under Windows

3.3 Some preliminary tests

Initially, I created two entities: Residence and Manager using *JDeveloper* and wrote some basic tests in relation to my application.

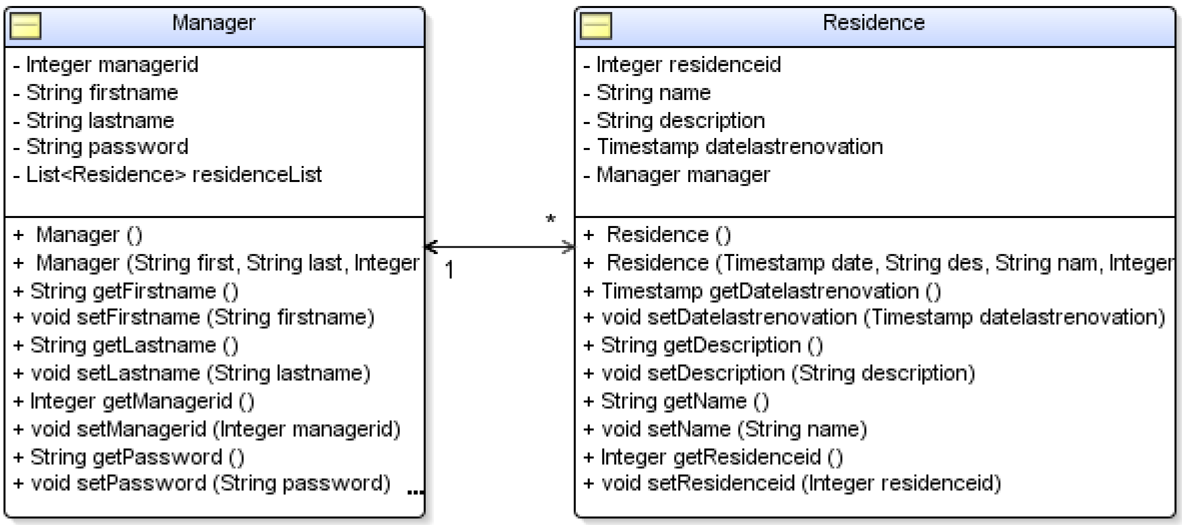

Manager
- Integer managerid
- String firstname
- String lastname
- String password
- List<Residence> residenceList
+ Manager ()
+ Manager (String first, String last, Integer
+ String getFirstname ()
+ void setFirstname (String firstname)
+ String getLastname ()
+ void setLastname (String lastname)
+ Integer getManagerid ()
+ void setManagerid (Integer managerid)
+ String getPassword ()
+ void setPassword (String password) ...
1
*
Residence
- Integer residenceid
- String name
- String description
- Timestamp datelastrenovation
- Manager manager
+ Residence ()
+ Residence (Timestamp date, String des, String nam, Integer
+ Timestamp getDatelastrenovation ()
+ void setDatelastrenovation (Timestamp datelastrenovation)
+ String getDescription ()
+ void setDescription (String description)
+ String getName ()
+ void setName (String name)
+ Integer getResidenceid ()
+ void setResidenceid (Integer residenceid)

Figure 3.3-1: UML diagram sample

- ✓ I test one of the setter and getter methods of the Residence entity: "*setName*" and "*getName*". setName sets the name of the residence in upper case if the name is no more than twenty characters. Normally, setter/getter methods are too trivial to require a test but it is be familiar with *JUnit* 3.8.

LISTING 3.3-1: TRIVIAL UNIT TEST

```
package tdd.test;

import junit.framework.TestCase; // JUnit library
import tdd.model.entity.Residence;  // Residence entity


public class TestResidenceTest extends TestCase {
                               // The class extends JUnit’s TestCase
    // Test 1 - simple
    public void testValidResidenceName() throws Exception {
        Residence res = new Residence();
        res.setName("OCEANIC PLACE");
        assertEquals("OCEATNIC PLACE", res.getName());
    }
    // Test 2 – length over twenty characters
    public void testResidenceNameInvalidLength() throws Exception {
        Residence res = new Residence();
        try {
            res.setName("DKIT Student village in Dundalk");
            fail("Residence names must be twenty characters max");
            } catch (IllegalArgumentException e) {}
    }
```

```
        // Test 3 – upper case correctly done
        public void testResidenceNameCase() throws Exception {
            Residence res = new Residence();
            res.setName("Holidays village");
            assertEquals("HOLIDAYS VILLAGE", res.getName());
        }
    }
```

- ✓ Then, I test the function "*DurationRenovation*" which return: how long the Residence has been renovated?

LISTING 3.3-2: ANOTHER UNIT TEST

```
    package tdd.test;

    import java.sql.Timestamp;
    import junit.framework.TestCase;
    import tdd.model.entity.Residence;
    import tdd.model.stateless.ResidenceBean;


    // MILLIS_PER_YEAR is a constant define in ResidenceBean
    public class TestResidenceBeanTest extends TestCase {

        public void testDurationRenovation() throws Exception {
            ResidenceBean bean = new ResidenceBean();
            Residence res = new Residence();
            res.setResidenceid(1);
            res.setDatelastrenovation(
                        new Timestamp(System.currentTimeMillis()
                        - ResidenceBean.MILLIS_PER_YEAR * 5) );
            int DurationRenovation =
                      bean.getDurationRenovation(res.getResidenceid());
            assertEquals(5, DurationRenovation);
        }
    }
```

Comments

The two tests above have performed some processing and returned a measurable result. Use an automated unit testing tool, in this case *JUnit,* to write effective unit tests is essential for a TDD programmer. Extending *TestCase* gives, among other things, a set of assertion methods such as *assertEquals()*.

There are some naming conventions, for example, all tests must be prefixed with the word "test", e.g. ***test****DurationRenovation()*. Indeed, all public void methods starting with test are considered as test cases by the *JUnit* TestRunner. Note that this naming convention is used for *JUnit* 3.8 only. In *JUnit* 4.0, an annotation is used instead. Test methods must not take parameters or return values.

JUnit has both text and GUI interfaces. This example uses the GUI interface to *JUnit*, with the results shown on the next page.

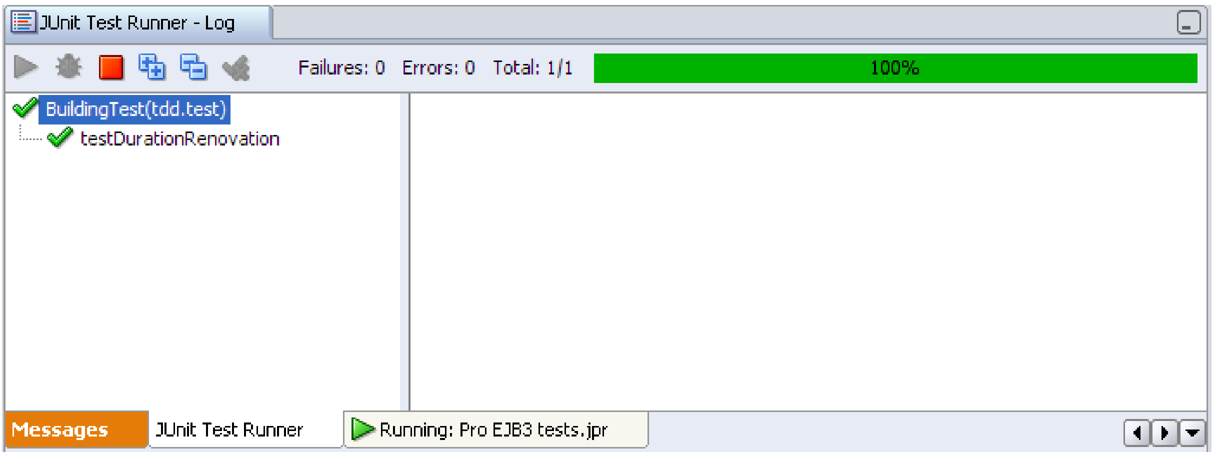
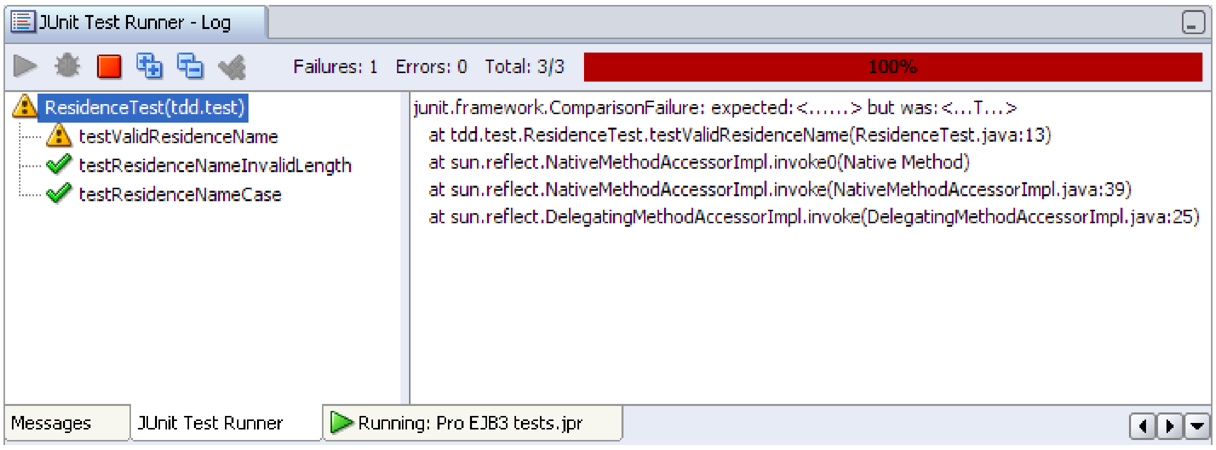


Figure 3.3-2: Success is displayed by a "green bar"



```
junit.framework.ComparisonFailure: expected:<......> but was:<...T...>
  at tdd.test.ResidenceTest.testValidResidenceName(ResidenceTest.java:13)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
```

Figure 3.3-3: A failure in one of the tests is displayed by a "red bar"

When the red light comes up, the first thing to do is looking at the exception raised by *JUnit.*
`Junit.framework.ComparisonFailure:`
`expected <......> but was: <...T...>.`
I take a glance at the code and immediately see that I have compared `"OCEANIC PLACE"` with `"OCEATNIC PLACE"`. I correct this test by simply removing the "T" and the tests now pass.

Running the tests is very fast because the *Java* embedded server and container is not needed. There is no delay starting and stopping the server.

I used *JUnit* 3.8 as this is supported in *JDeveloper* 10.1.3.3.0. As I mentioned, the latest version 4.0 uses annotations but is not supported in this version of *JDeveloper*.

4. Plan for Progression

Actually, this part is not essential to the report but it can be interesting to see the progression of such project. This plan for progression was written at the end of the first semester and shows the main steps I have to investigate during the second semester.

Starting date	Description	Allotted Time (weeks)
08/01/2008	- To compare different unit testing tools and take the most suitable (*JUnit*, *EJB3Unit* or *Cactus*). - To compare two libraries for Mock Object (*EasyMock* and *JMock*) and take the most suitable. To do that, I have to install these plugins, read associated documentations and write tests. - To continue with reading the "T*est Driven TDD and Acceptance TDD for Java Developers*" book from Manning.com and the "Black book" at the DkIT library "Test first development with JUnit" , ref SP1105 Student Project	3
29/01/2008	Write some tests in relation to my application, more complex than in this first part and using Mock Object, etc.	2
19/02/2008	Think about the future functions of my application, write use cases and UML diagrams	1
26/02/2008	Develop my prototype application using the test previously made and new ones. Using "test, code and refactor" cycle…	10
29/04/2008	Finish writing this report	1
	Total:	**17 weeks**

→ During these 17 weeks, I will continuously write my report.

5. Design

In this chapter, I will describe the main steps to create my prototype application. As part of the code design, I will present the concept of annotations which I briefly introduced in the literature review. Annotations are fundamental when you use the latest version of *Java Enterprise* (1.5).

5.1 Functionalities of my sample application

Before writing the list of tests, I have to think about the future functions of my application. This is the starting point of my application. Once the functionalities are defined, I will be able to start to test, code and refactor the different parts (i.e. functionalities) of my application.

Login: the application's access is limited to the managers

Manager management: control the manager's access to the application
- Add a manager
- Modify a manager (his password)
- Delete a manager

Customer management
- Add a new customer
- View a customer: view customer details
- Delete a customer

Rent out an apartment: let an apartment located in a particular residence to a customer

Release an apartment: the released apartment is available for other customers

Statistics: some numbers which can help the manager
- Number of customer registered
- Number of tenants
- Tenant Rate
- Total number of apartments
- Total number of apartments available
- Occupation Rate
- Number of apartments available for each residence

Logout: facility to log out

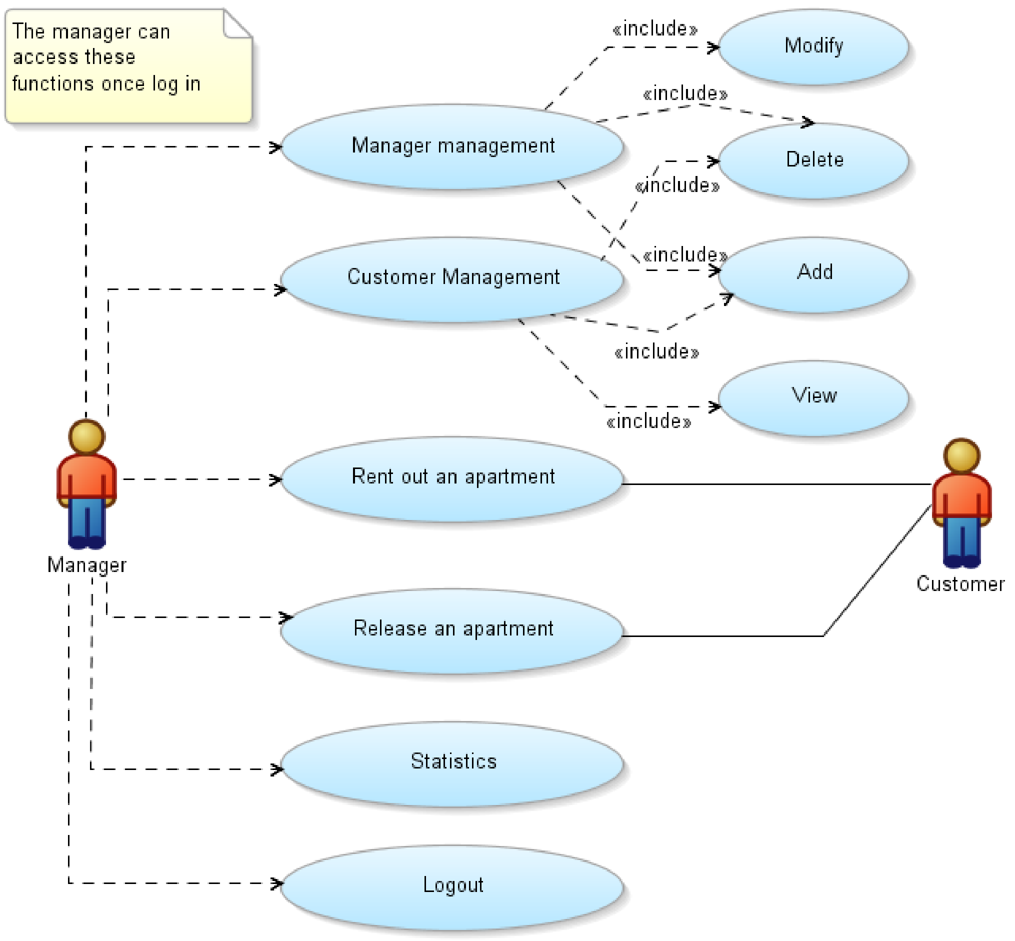


Figure 5.1: Use case Diagram

As can be seen, I have seven main functionalities (including Login) which will require manipulation of Managers, Customers, Apartments and Residences. Actually, each of these terms will "become" a table in my database and an entity in the code.

5.2 Creating the list of tests

5.2.1 List of requirements

Before writing the initial list of tests, I need to think about some requirements. For my residences management sample application, I have identified the following set of requirements:

<u>Login</u>

- The system recognizes a manager when he is logged in to the system. To log in, the manager has to give his username and his password and this information has to be valid (i.e. exists in the database – Manager table)
- If a managers tries to connect but leaves out some piece of information (username, password or both), the system attempt to ask the missing parameter.
- If the manager try to connect but the information filled out is false, the system has to inform the manager that the login was unsuccessful.

<u>Manager administration</u>

Add

- Once logged in, a manager can add a new manager. The manager must enter some information about the new manager (firstname, lastname, username and password). Username and password are used for the log in.
- If a manager tries to add a manager who already exists in the database (i.e. same username), the system will not create the new manager.
- If a manager tries to add a manager but does not give all the required information, the system will not create the new manager.

Delete

- A manager can delete a manager to remove his access to the application. The manager simply selects the manager in a list.
- If a manager tries to delete a previously deleted manager or who does not exist in the database, the system will inform that the manager does not exist.

Modify

- A manager can only modify his password or the password of another manager. The manager selects the manager he wants to modify (or himself), enters the old password and enters the new password.
- If the manager tries to modify the password of a manager who does not exist, the system will inform the manager.
- If the manager gives a wrong old password, the system will not consider the new password.

<u>Customer administration</u>

Add

- Once logged in, a manager can add a new customer. Thus, this customer will be able to rent an apartment in one of the residences. The manager has to give some information about the new customer (firstname, lastname, date of birth, average salary, email address and phone number). Only the phone number is not required. When a customer is registered in the database, a date of registration is automatically calculated and recorded.
- If a manager tries to add a customer but does not give all the required information, the system will not create the new customer.

Delete

- A manager can delete a customer even if this customer is currently renting out an apartment. Indeed, the apartment will be automatically released and be available for another customer. The manager simply selects the customer in a list.
- If a manager tries to delete a customer he had deleted before or who does not exist in the database, the system will inform that the customer does not exist.

View

- The manager selects a customer in a list to display the customer's personal details (and also his date of registration)

<u>Let an apartment</u>

- To let an apartment to a customer, the manager needs to provide three pieces of information (the customer, the residence and the apartment). First, the manager selects a customer in the list. Then, he chooses the residence the customer wants to live in and finally chooses one of the available apartments.
- If one of the three steps fails (e.g. there is no apartment available in one residence), the system will inform the manager and no rent will be registered (e.g. the system will ask the manager to select another residence).

<u>Release an apartment</u>

- To release an apartment is very simple; the manager just needs to select one of the customers.
- If the customer does not currently occupy an apartment, nothing will happen. No release is needed but the system will inform the manager that the current customer is not renting.

Statistics

- The manager can access to a list of figures which can help him in his business (number of customers registered, number of tenants, tenant rate, total number of apartments, total number of apartments available, occupation rate, number of apartments available per residence)

Logout

- A manager who is logged in to the application can quit the program simply by clicking on the logout button. The logout function is not accessible by logged off users.

Because these requirements are very detailed, I can now write the tests. Tests are typically more explicit and describe the behaviour of specific scenario rather than giving the description of that behaviour.

5.2.2 List of tests

Here is one attempt at turning the residences management requirements into proper set of tests:

Login [Name of the method used: verifyLogin]

I insert the following record directly in the mock object and in the database – Manager table [*"Fabian", "Piau", "piauf", "mypswd"]* to have one basic registered manager.

- I log in with this manager with the valid information ("piauf", "mypaswrd"). It results in a successful login.
- I log in with this manager but I omit the username (*null*, "mypaswrd"). It results in login failed. Omitting the username is represented by null in my test. It is not the empty string but NULL, the special value in programming.
- I log in with this manager but I omit the password ("piauf", *null*). It results in login failed.
- I log in with this manager but I do not fill out any field (*null*, *null*). It results in login failed.
- I log in with a manager who does not exist, with the invalid information ("unknown", "nopaswrd"). It results in login failed.

Manager administration

Add [addManager]

- I test to add the following manager [*"Lucky", "Luke", "lukel", "jumper"]*. It results in a successful insertion. To verify that this new manager is correctly inserted, I can try to log in to the application with this new manager.

- I test to add the following manager who already exists [*"Fabian", "Piau", "piauf", "mypswd"]*. It results in a non-insertion of the manager.

Delete [deleteManager]

- I test to delete the following manager who exists in the mock object/database [*"Fabian", "Piau", "piauf", "mypswd"]*. The deletion is successful.

- I test to delete the following manager who does not exist in the mock object/database [*"unknow", "unknow", "unknow", "nopswd"]*. The deletion is unsuccessful.

Modify [changePasswordManager]

- I test to modify the following manager who exists in the mock object/database [*"Fabian", "Piau", "piauf", "mypswd"]* and give the new password "newpswd" The modification is successful.

- I test to delete the following manager who does not exist in the mock object/database [*"unknow", "unknow", "unknow", "nopswd"]*. The modification is unsuccessful.

Customer administration

I insert the following record directly in the mock object and in the database – Customer table *["Sarah", "Connor", "08/03/1970", "sarah.connor@skynet.com", "1500 - 2000", "0567891256"]* to have one basic registered customer.

Add [addCustomer]

- I test to add the following customer [*"The", "Daltons", "01/05/1950", "daltons@farwest.fr", "> 3000", ""]*. It results in a successful insertion.

- I test to add the following manager who already exists *["Sarah", "Connor", "18/11/1981", "Sarah.connor@gmail.com", "500 - 1000", ""]*. It results in a non-insertion of the customer. The combination lastname/firstname has to be unique.

Delete [deleteCustomer]

- I test to delete the following customer *["Sarah", "Connor", "08/03/1970", "sarah.connor@skynet.com", "1500 - 2000", "0567891256"]*. It results in a successful deletion.

- I test to delete the following manager who does not exists [*"unknown", "unknown", "11/02/1980", "unknown@unknown.fr", "> 3000", ""*]. It results in a non-deletion of the customer.

I do not turn all the requirements into tests (e.g. logout, statistics, see a customer, etc.) because they are too simple and too basic to test.

Now, the list of tests is become something which is a degree more concrete, more executable than the requirements. The next step is to make these tests pass one by one. At the end, these tests will document my application, especially the code.

The entire list of tests is available in the appendices. (p. 60, 61 – UnitTest.java file)

5.3 Examples of annotations in EJB 3.0

Enterprise Beans in EJB 3.0 are easier to test than EJBs written with prior versions of the specification. This is largely because in the 3.0 specification EJBs are simply POJOs annotated with specific EJB 3.0 annotations. (Panda, 2004 [20])

The goal of EJB 3.0 is to simplify development by supporting the use of metadata annotations to "generate several things" (such as interfaces), as well as to replace deployment descriptors.

From a developer's point of view, annotations are modifiers - just like public - and can be used in classes, fields, methods, parameters, local variables, constructors, enumerations and packages.
One major goal of J5EE is to simplify development using annotations, so as you might expect, it includes its own set of annotations. Annotations are marked with @ as follows:

LISTING 5.3-1: A BASIC EXAMPLE OF ANNOTATION

```
@Author("Fabian Piau")

@Bean
public class MySessionBean
```

As I mentioned previously, in EJB 3.0, annotations replace deployment descriptors. Each attribute in the deployment descriptor has default values so you do not have to specify these attributes unless you want a value other than the default value. These values can be specified using annotations in the bean class itself.

Before moving to the next example, let take a glance at *Local* and *Remote* interfaces. A Local interface means that it is accessible only to clients within the same application server. It is not possible to use a session bean with a local interface from a remote client, for example (Keith and Schincariol, 2006 [31]). Thus, a second type of business interface for remote clients exists and is called Remote interface.

LISTING 5.3-2: THE LOCAL ANNOTATION

```
package model.stateless;

import java.util.List;
import javax.ejb.Local;


@Local
public interface ApartmentServiceLocalEJB {
    String getApartmentInformation(Long apartmentid);
    Boolean giveCustomerApartment(Long customerid, Long apartmentid);

   // etc.
}
```

I indicate, in this example, that the ApartmentServiceLocalEJB interface is the local interface for the ApartmentServiceBean EJB only with the "@Local" annotation. This annotation uses the "ja - vax.ejb.Local" class, so it needs to be imported.
Instead of using "@Local" it is possible to use the "@Remote" annotation to declare that the interface should be used remotely.

The following example illustrates the specification of some value on annotations' properties.

When an entity is associated with a Collection of other entities, it is most often in the form of a one-to-many mapping. For example, a residence would normally have a number of apartments. Figure 5.3 shows the Residence and Apartment relationship.

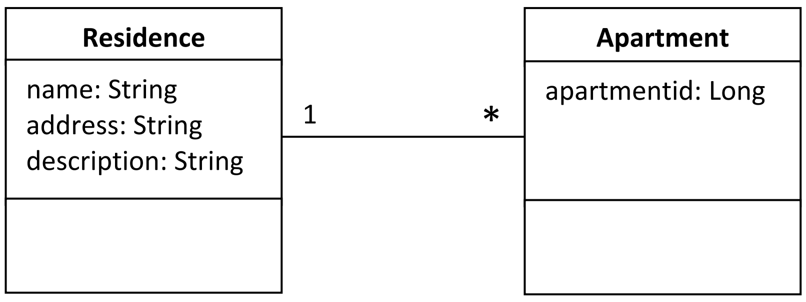


Figure 5.3: Residence and Apartment relationship

LISTING 5.3-3: ANNOTATIONS IN RELATION WITH DATABASES

```
package model.entity;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.OneToMany;

@Entity
@NamedQueries({
  @NamedQuery(name = "Residence.findAll", `
              query = "select o from Residence o order by o.name"),

  @NamedQuery(name = "Residence.findNbTotalApartmentResidence",
              query = "select count(a.apartmentid) from Apartment a
                       where a.residence.name = :residencename"),
  @NamedQuery(name = "Residence.findNbUnavailableApartmentResidence",
              query = "select count(a.apartmentid) from Apartment a,
             Customer c where a.apartmentid = c.apartment.apartmentid
                  and a.residence.name = :residencename"),

  @NamedQuery(name = "Residence.findAllApartmentResidence",
              query = "select a from Apartment a where
                 a.residence.name = :residencename order by a.numb"),
  @NamedQuery(name = "Residence.findUnavailableApartmentResidence",
              query = "select a from Apartment a, Customer c where
                       a.apartmentid= c.apartment.apartmentid and
                       a.residence.name = :residencename")
})
public class Residence implements Serializable {
    private String address;
    private String description;
    @Id
    @Column(nullable = false)
    private String name;
    @OneToMany(mappedBy = "residence")
    private List<Apartment> apartmentList;

    // etc.

}
```

To turn the *Residence* class into an entity I first need to annotate the class with *@Entity*. This is primarily just a marker annotation to indicate to the persistence engine that the class is an entity.

The second annotation that I need to add is *@Id*. This annotates the particular field or property that holds the persistent identity of the entity (i.e. the primary key).

To indicate specific characteristics of the physical database column that the object model is less concerned about, I can specify the *@Column* on the attribute. In my case, I want that the name of the residence is never null.

Named queries are a powerful tool for organizing query definitions and improving application performance. A named query is defined using the *@NamedQuery* annotation, which may be placed on the class definition for any entity. The annotation defines the name of the query, as well as the query text. If more than one named query is to be defined for a class, they must be placed inside of a *@NamedQueries* annotation, which accepts an array of one or more *@NamedQuery* annotations. (Keith and Schincariol, 2006 [31])

On the residence side of the relationship, I need to map the *apartmentList* list of *Apartment* entities as a one-to-many association using the *@OneToMany* annotation.
As part of the annotation I must add a *mappedBy* element to indicate that the owning side is the *Apartment* and not the *Residence*. Because *Residence* is the inverse side of the relationship, it does not have to supply the join column information. (Keith and Schincariol, 2006 [31])

6. Implementation

This chapter presents the basics to implement TDD techniques and Unit testing. *Java Enterprise 1.5* provides some features such as dependency injection which are useful to implement Unit testing in your *JavaBeans* Application. As I mentioned previously, this chapter also presents in more detail *JavaServer Faces* (JSF), the server side user interface component I have used to implement my application.

6.1 Unit testing vs. Integration testing

6.1.1 Two scenarios

There are two common scenarios where EJB unit testing may be performed: during development and during integration.

By “development”, I mean at the time of coding. The developer may want to run the unit test at any time after some code changes. This testing is normally performed on one or a few beans at a time.

By “integration”, I mean at the time beans from different developers are integrated together and integration test suites are run, since beans may behave differently when assembled together. This is normally adopted by XP teams from the idea of continuous integration. Integration tests may be run every day or even every hour.

Generally, there is confusion with unit testing and integration testing.

6.1.2 Unit testing

Unit tests are written by developers and focus on isolated components of an application. Unit testing is used to validate that individual units of source code are working properly. Depending on your approach, this may be a single method or a class. In other words, Unit testing is testing code in isolation. The only key defining element is that the unit test is not coupled to any server resources (these are typically mocked) and execute very quickly. It must be possible to execute an entire suite of unit tests from within an IDE and get the results in a matter of seconds.

6.1.3 Integration testing

Individual software modules are combined and tested as a group. Integration tests are also written by developers and focus on use cases within an application. They are still decoupled from the application server, but the difference between a unit test and an integration test is that the integration test makes full use of external resources such as a database. In effect, an integration test takes a component from an application and runs in isolation as if it were still inside the application server. (Keith and Schincariol, 2006 [31])

Running the test locally makes it much faster than a test hosted in an application server but still slower than a unit test. Integration tests are also automated and often run at least daily to ensure that there are no regressions introduced by developers.
Integration testing is an extension of unit testing that takes components of a *Java EE* application and executes them outside of an application server. (Keith and Schincariol, 2006 [31])

6.1.4 Summary

The big difference between the two types of testing is that unit testing focuses on the internal logic of one procedure, and integration testing tries to identify problems that might happen when one procedure (the parent or "outer" procedure) calls another procedure (the child or "inner" procedure). Another important difference is that integration testing usually follows unit testing. After ridding two or more procedures of any internal defects, you can integration-test the procedures by checking for defects in the outer procedure's call statements and in any data or error messages the inner procedure returns to the outer procedure. (Sawyer, 2004 [28])

6.2 Dependency Injection and Unit Testing

6.2.1 What is Dependency Injection?

The next major change associated with EJB 3.0 is actually a change in the J5EE specification: dependency injection. The process of automatically looking up a resource and setting it into the class is called dependency injection because the server is said to inject the resolved dependency into the class. The J5EE specification requires that J2EE application clients, enterprise beans, and web components have access to a JNDI (Java Naming and Directory Interface) naming environment. The naming environment can be accessed either through explicit JNDI lookup calls or through annotations which specify that the container injects the dependency automatically. What this means is that you can now declare a dependency on an EJB or other container-managed resources through a simple annotation such as:

LISTING 6.2.1: A DEPENDENCY INJECTION USING FIELD INJECTION

```
@Stateless
public class ApartmentServiceBean implements ApartmentServiceEJB {
      @EJB
      BedroomService bedroom;
      // etc.
}
```

The container will inject an implementation of the Bedroom interface into the Apartment and the Apartment can use the object as if were a simple POJO.

6.2.2 Two kinds of Dependency Injection

Field Injection

The first form of dependency injection is called field injection. Injecting a dependency into a field means that after the server looks up the dependency in the environment naming context, it assigns the result directly into the annotated field of the class. The piece of code above is an example of field injection.

Field injection is certainly the easiest to implement. The only thing to consider with field injection is that if you are planning on unit testing, then you need either to add a setter method or to make the field accessible to your unit tests in order to manually satisfy the dependency.

Setter Injection

The second form of dependency injection is called setter injection and involves annotating a setter method instead of a class field. When the server resolves the reference, it will invoke the annotated setter method with the result of the lookup.

LISTING 6.2.2: A DEPENDENCY INJECTION USING SETTER INJECTION

```
@Stateless
public class ApartmentServiceBean implements ApartmentServiceEJB {
private BedroomService bedroom;
      @EJB
      public void setBedroomService(BedroomService bedroom) {
      this.bedroom = bedroom;
      }
      // etc.
}
```

This style of injection allows for private fields yet also works well with unit testing. Each test can simply instantiate the bean class and manually perform the dependency injection by invoking the setter method, usually by providing an implementation of the required resource that is tailored to the test.

6.2.3 Why Dependency Injection can ease unit tests

The dependency injection approach offers more flexibility because it becomes easier to create alternative implementations of a given service type. This is especially useful in unit testing, because it is easy to test particular units of code by injecting a mock implementation of a service (e.g. use a dummy mock class to act like the real one) into the object being tested; thus, removing the need to test the dependent code. (Wikipedia [15])

Without any JNDI API code in the class that has dependencies on the application server runtime environment, the bean class may be instantiated directly in a unit test. The developer can then manually supply the required dependencies and test the functionality of the class in question instead of worrying about how to work around the JNDI APIs.
To summarize, dependency injection aims to solve a particular problem in designing and constructing data structures dependent on other pieces of code, in a way that minimizes the coupling between them.

6.3 Testing the functionality of the application

6.3.1 EJBs overview

In the following sections, I provide a brief description of two types of EJBs.

<u>Entity bean</u>

An entity bean (or entity) represents an object in a persistent storage mechanism. In my application, these objects are customers, managers, apartments, and residences. Typically, each entity bean has an underlying table in a relational database, and each instance of the bean corresponds to a row in that table. As database tables have relationships (foreign keys), entities also have relationships to other entities. I have four entities representing four tables in my database. I have used the same names for my entities and my tables, although it is not necessary to do so. The relationships between the four entities are shown on page 63 ("Full UML diagram" in the appendices).
In the most general sense, entities are business domain objects that have specific meaning to the application that accesses them (Keith and Schincariol, 2006 [31]).

<u>Session bean</u>

Session beans are a component technology designed to encapsulate business services. The operations supported by the service are defined using a regular Java interface, referred to as the business interface of the session bean, that clients use to interact with the bean (Keith and Schincariol, 2006 [31]). There are two types of session bean, stateless and stateful.

Stateless
A stateless session bean is a distributed object that does not have an associated conversational state, thus allowing concurrent access to the bean. The contents of instance variables are not guaranteed to be preserved across method calls (Wikipedia [15]). I have created stateless bean for my application.

Stateful
Stateful session beans are distributed objects having a conversational state. The state could be persisted, but access to the bean is limited to only one client. (Wikipedia [15])

Actually, a third type of EJBs exists: Message-driven beans. I mention them here but I did not use them.

6.3.2 Functionalities in the session beans

What I need to test are my stateless session beans. Indeed, all functionalities are in the session beans. To organize my session beans, I have brought together the functionalities by entity. In other words, one session bean contains all the functionalities to manage a customer, another all the functionalities to manage an apartment, etc. For instance, all methods and code referring to the login are in the manager session bean because only a manager can log into the system.

Following this session beans "organization" and having four entities, I have created four stateless session beans: ResidenceServiceBean, ApartmentServiceBean, CustomerServiceBean and ManagerServiceBean.

6.3.3 Testing session beans

My session beans need access to the entities in order to function properly. However, when I want to test a session bean, I do not want to test the entities. Indeed, entities require the services of the database server and involve some dependencies. Recall that a good test in TDD has to run fast and has to run in isolation.

The solution is to use Mock Objects. Instead of using a library such as *EasyMock*, I created my own mock objects. I created one mock object for each session bean: ResidenceServiceBeanMock, ApartmentServiceBeanMock, CustomerServiceBeanMock and ManagerServiceBeanMock. In concrete terms, these mock objects will simulate the entities behaviour in order to avoid using the entities.

LISTING 6.3.3-1: REPLACEMENT OF ENTITY BY A SIMPLE LIST IN THE MOCK OBJECT

```
public class ManagerServiceBeanMock implements ManagerServiceRemoteEJB {
    private List<Manager> listManag = new ArrayList<Manager>();

    public ManagerServiceBeanMock() {
       listManag.add(new Manager("Piau", "Fabian", "piauf", "181186"));
       listManag.add(new Manager("Bond", "James", "bondj", "007"));
    }
// etc.
}
```

As can be seen, my mock object simulates the *Manager* entity with a simple list of *Manager* Objects. I can add, remove, and modify managers in my test without using any entities.
Each time I create this mock object, I insert two managers in the list (constructor method). This simulates the two same rows I normally have in the “Manager” table in my database.

Once the tests run successfully against the mock object (green light with *JUnit*), I run the tests again but against the database. To switch between Mock Object and database, I simply use the *setUp()* method of *JUnit*. This particular method is always called in first by the *JUnit* test runner when the test case is executed.

LISTING 6.3.3-2: RUN THE TESTS AGAINST THE MOCK OBJECTS

```
    @Override protected void setUp() {
        ManBean = new ManagerServiceBeanMock();
        CustBean = new CustomerServiceBeanMock();
        AptBean = new ApartmentServiceBeanMock();
        //ManBean = createManagerBean();
        //CustBean = createCustomerBean();
        //AptBean = createApartmentBean();
    }
```

LISTING 6.3.3-3: RUN THE TESTS AGAINST THE DATABASE

```
    @Override protected void setUp() {
        //ManBean = new ManagerServiceBeanMock();
        //CustBean = new CustomerServiceBeanMock();
        //AptBean = new ApartmentServiceBeanMock();
        ManBean = createManagerBean();
        CustBean = createCustomerBean();
        AptBean = createApartmentBean();
    }
```

The entire code for the *ManagerServiceBeanMock* mock object is available in the appendices (p. 61-62)

It is possible to test entity beans and I did so as an introduction to *JUnit* (see p. 28) but they are too trivial to be tested. Moreover, entity beans are automatically created by *JDeveloper* from the tables within the database. Thus, an error in your entity beans certainly means an error in the definition of your tables.

6.4 Order of tests with JUnit

The following is an extract from my set of tests:

LISTING 6.4-1: MANAGER UNIT TEST SAMPLE

```
    public void testAddManager() {
        assertNull(ManBean.verifyLogin("lukel", "jumper"));
        // test Add Manager
        assertTrue(ManBean.addManager("Lucky", "Luke", "lukel", "jumper"));
        Manager m = ManBean.verifyLogin("lukel", "jumper");
        assertEquals (m.getUsername(), "lukel");
        assertEquals (m.getPassword(), "jumper");
        // test Add Same Manager
        assertNotNull(ManBean.verifyLogin("lukel", "jumper"));
        assertFalse(ManBean.addManager("Lucky", "Luke", "lukel", "jumper"));
    }

    public void testDeleteManager() {
        // test Delete Manager
        assertTrue(ManBean.deleteManager("Lucky", "Luke", "lukel"));
        assertNull(ManBean.verifyLogin("lukel", "jumper"));
        // test Delete Same Manager
        assertFalse(ManBean.deleteManager("Lucky", "Luke", "lukel"));
        // test Delete non-exist Manager
        assertFalse(ManBean.deleteManager("Lucky", "Luc", "lukel"));
    }
```

At first sight, there is no problem but an issue came to light when I ran these tests with *JUnit*. The second test (deletion) fails without obvious explanation.

I have a situation where I would like to use the results of one test in another test. The first test “testAddManager” checks if I can create a new instance of Manager in the database; the second test “testDeleteManager” attempts to delete the instance created in the first test.

My first attempt was to rearrange the two tests and merge them into one unique test. This is the new test I obtained after merging:

LISTING 6.4-2: MERGING INTO A UNIQUE UNIT TEST

```
public void testAdministrationManager() {
    assertNull(ManBean.verifyLogin("lukel", "jumper"));
    // test Add Manager
    assertTrue(ManBean.addManager("Lucky", "Luke", "lukel", "jumper"));
    Manager m = ManBean.verifyLogin("lukel", "jumper");
    assertEquals (m.getUsername(), "lukel");
    assertEquals (m.getPassword(), "jumper");
    // test Add Same Manager
    assertNotNull(ManBean.verifyLogin("lukel", "jumper"));
    assertFalse(ManBean.addManager("Lucky", "Luke", "lukel", "jumper"));

    // test Delete Manager
    assertTrue(ManBean.deleteManager("Lucky", "Luke", "lukel"));
    assertNull(ManBean.verifyLogin("lukel", "jumper"));
    // test Delete Same Manager
    assertFalse(ManBean.deleteManager("Lucky", "Luke", "lukel"));
    // test Delete non-exist Manager
    assertFalse(ManBean.deleteManager("Lucky", "Luc", "lukel"));
}
```

Without explanation, all the tests now pass! I can see the green bar instead of the red one. After some research, I found a good explanation.

There is no fixed order in which the *JUnit* framework may execute the test methods during a run of the test runner. Indeed, *JUnit* does not specify the order in which the tests are run. The reason is that tests are supposed to be independent of each other.

What I have done (merging tests) is one solution but there is an alternative which control the order of the tests: use a *static suite()* method like this one to ensure the ordering:

LISTING 6.4-3: A SUITE TO CONTROL THE ORDER OF TEST

```
public static Test suite() {
   suite.addTest(new SomeTestCase ("testDoThisFirst";));
   suite.addTest(new SomeTestCase ("testDoThisSecond";));
   return suite;
}
```

There is no guarantee in the *JUnit* API documentation as to the order in which your tests will be called, because *JUnit* employs a *Vector* to store tests. However, you can expect the above tests to be executed in the order they were added to the test suite. (Schneider, 2001 [29])
What I tried to do is very common among programmers. There are many things I can only test after I have done a lot of things to set up an environment where things make sense. But I have to do it all in one test case, making assertions along the way to make sure that everything is as it should be before I go to the next step. It is preferable that each test cover as little functionality as possible, but that is not always possible. It is also preferable to avoid modifying the database in the tests, but sometimes it is unavoidable.

6.5 JavaServer Faces

The JSF user interface (UI) component model includes (Goyal and Varma, 2004 [38]):

- ✓ A set of UI components
- ✓ Managed Beans
- ✓ Page navigation support

To explain how it works, I will base most of my examples on the login page.

6.5.1 UI Components

JSF lets you create user interfaces from a set of standard, reusable server-side components and provides a set of JSP tags to access those components. (Command, Form, Output, Input, Panel, SelectBoolean, SelectMany, SelectOne, etc.)

LISTING 6.5.1: JSP TAGS TO ACCESS YOUR UI COMPONENTS

```
<h:outputLabel value="Welcome #{loginBean.firstname}"
#{loginBean.lastname}" />

<h:inputText required="true" maxlength="10"
value="#{loginBean.username}" id="username"/>

<h:inputSecret maxlength="10" required="true"
value="#{loginBean.password}" id="password"/>

<h:commandButton  value="Connect" action="#{loginBean.verifyLogin}"
styleClass="bouton_submit" />

<h:selectOneMenu value="#{modifyManBean.managerid}">
         <f:selectItems value="#{modifyManBean.list_manager}"/>
</h:selectOneMenu>
```

The `h:inputText`, `h:inputSecret` and `h:commandButton` correspond to the text field, password field and the submit button of the login page.

The input fields are linked to object properties. For example, the attribute `value="#{loginBean.username}"` tells the JSF implementation to link the text field with the *username* property of a `loginBean` object (Geary and Horstmann, 2007 [39]). In the *loginBean* class, the *setUsername(), getUsername(), setPassword(), getPassword()* methods must be implemented.

6.5.2 Page navigation

faces-config.xml is a key configuration file type within a *JavaServer Faces* software implementation. It lists the navigation rules. A navigation rule tells the JSF implementation which page to send back to the browser after a form has been submitted.

LISTING 6.5.2: NAVIGATION MODEL

```
<navigation-rule>
    <from-view-id>/login.jsp</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/welcome.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>failure</from-outcome>
      <to-view-id>/login_failure.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```

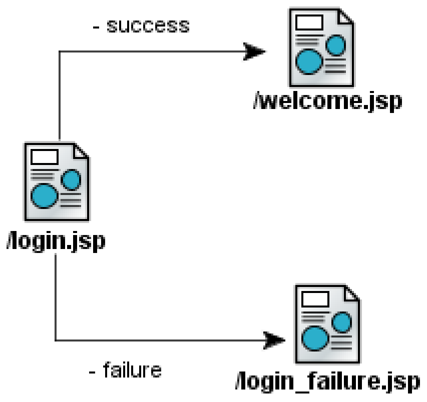


Figure 6.5.2: Navigation diagram generated using JDeveloper

If the login fails, the user is redirected on the *login_failure* web page, if the login is successful, the user is redirected on the *welcome* web page.

6.5.3 Managed bean

A bean is needed to manage the user data (username and password). It is this bean (in fact, *verifyLogin()* method of this bean) which will return “success” or “failure” depending on the username and password correctness. The beans resources are also listed in *faces-config.xml,* generally at the end (i.e. after the navigation rules).

LISTING 6.5.3: THE LOGIN MANAGED BEAN REFERENCED IN THE FACE-CONFIG.XML FILE

```
<managed-bean>
    <managed-bean-name>loginBean</managed-bean-name>
    <managed-bean-class>view.beans.LoginBean</managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
</managed-bean>
```

Note that *JDeveloper* automatically modifies the *faces-config.xml* file. You simply need to draw a diagram as Figure 6.4.2 and turn your bean into a managed bean with some clicks with a wizard. Personally, I did not use the diagram drawing function, as I preferred to write my navigation rules by hand but I used the managed bean wizard which is very practical.

7. Conclusion

This final chapter reviews the aims and objectives set out at the beginning of this report. In addition, the main problems encountered and the learning outcomes will be outlined. You will also find my opinion, the advantages, issues, etc. which came up when investigating TDD area and using it for the development of my application.

7.1 Review of aims and objectives

The aims of the project were to carry out an in-depth study of TDD applied to EJB 3.0, to develop a sample application using TDD, and to provide an analysis of this method. They were generally met by this report and my application prototype.

I consider that the application was too "data-centric" without sufficient complex logic to fully test TDD. Indeed, I have mentioned the automatic libraries for creating mock objects (*EJBUnit*, *EasyMock* and so on) in this report but never use them during the development of my application. Moreover, my application uses some SQL queries, and it is very simple to verify a query just by typing it in the *Oracle* SQL "executer" and see if the result is expected.

But do not conclude that the development of my application was useless and not suitable for Unit Testing. On the contrary, this application was a very good introduction to TDD. With it, I have created my own mock objects to mock entities, I have tested stateless session beans, etc. Now, I know exactly what TDD is and can use it in my next projects.

7.2 Problems encountered

During the development, I had some difficulties with the "rent out an apartment" function because of the non-destruction of the stateless session bean.

When a manager selects the "rent out an apartment" function with the main menu, the application loads in one field of the bean "*RentApartbean*", i.e. the list of customers who have yet to rent an apartment. When the manager has selected a customer and gives him an apartment, one expects that this customer does not appear in the list anymore. But this can happen only if the application deletes and rebuilds the list, in other words, destroys the bean to create a new one with a new list. This is a problem as the "rent out an apartment" is performed in three steps and the manager can leave by clicking on another function in the menu.

I solved this problem by adding a "cancel" button on each page of each step (see below screenshot of step 2). This button calls the *cancel()* method which will destroy the session. Thus the *java* manager will be able to destroy the session bean. If the manager decides to give up before confirming the rental, the only choice he has is to use this button. Indeed, all the functions in the menu have been deactivated.

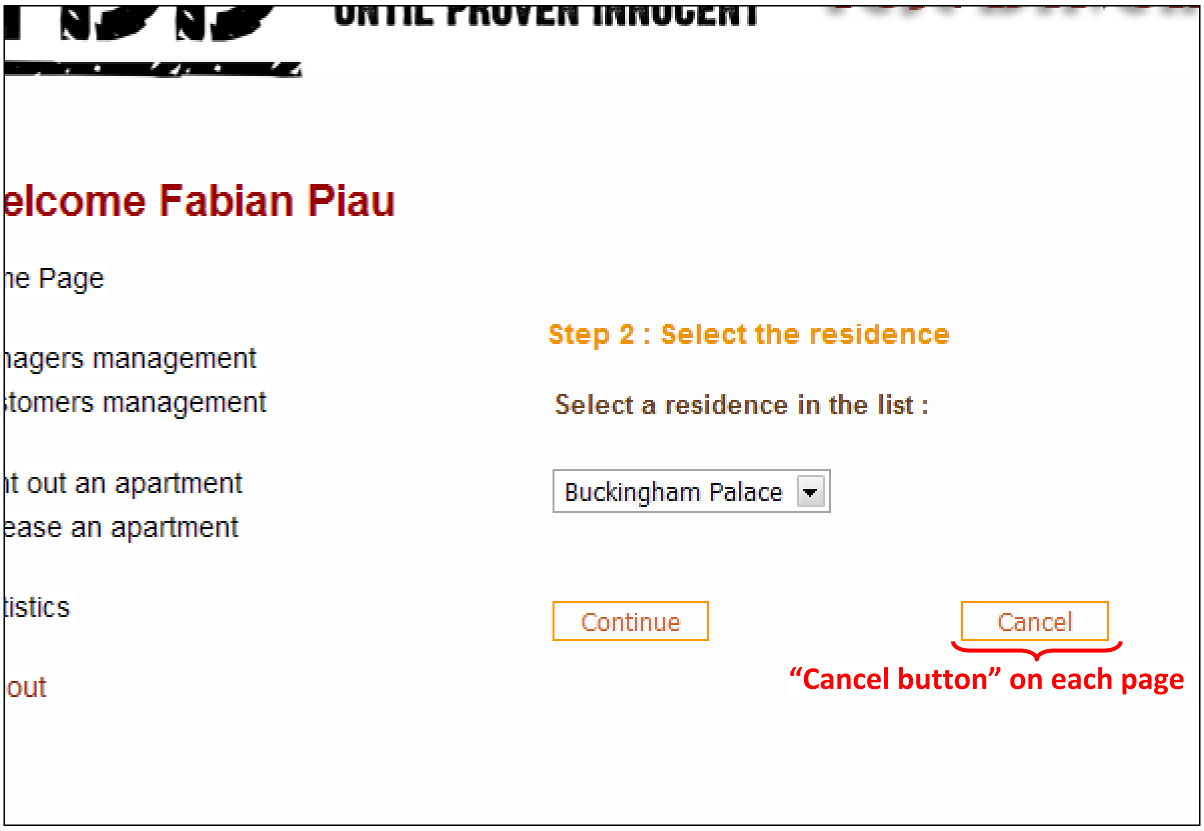


Figure 7.2: The "cancel" button

Another problem was the systems resources needed by *Oracle JDeveloper* together with my database management system (*Oracle Database 10g Express Edition*). This problem was not a real nuisance but a waste of time. Every time I want to launch the sample application or run my test against the database, I have to wait for *JDeveloper* and the embedded server to start. In addition, I need to restart the embedded server every time I modify the code of a bean (even for a little modification).
On my laptop, I have 1 gigabyte of ram with a simple core processor. At least 90% of its resources were used even with a large virtual memory on hard drive.
I would recommend using a strong computer with at least 2 gigabytes of ram in order to work comfortably on this kind of development.

Actually, this resources problem is a good argument for using TDD because you do not need to start your database on top of *JDeveloper* when you want to test your code.

7.3 Learning outcomes

Learning a new method of development was the main part of this project but throughout it, I also gained knowledge and experience in the latest *Java Enterprise Edition platform*, *JavaBeans*, and also *JavaServer Faces* which are technologies I never saw before and are currently trendy in IT companies. From a personal perspective, acquiring this experience will considerably help me to build my professional plan and open up wider career horizons to me.

By writing the literature review, I also improved my skills in information retrieval (processing a large amount of information and differentiating between the important and the trivial, etc.) and obviously my English writing skills.

I learned to use new tools such as *JUnit*, *Oracle XE*, and *JDeveloper*. Before this project, I was more familiar with *Eclipse*. Learning a new IDE is not so difficult, as the functionality and user interface are similar.

The idea of keeping an up to date blog (once a week) as my project progressed was valuable. In the beginning I thought it was restrictive and a waste of time but in the end, it was the best and easiest way to follow my progress. I will use it again for my future projects.

7.4 Project conclusion

7.4.1 Some TDD technicalities

The following are some subtleties about TDD which I have encountered myself and which often crop up on forums about TDD.

Tests that depend on some data pre-conditions (such as certain records existing in the database) are brittle and will break when the data changes. Tests should ideally set up all the pre-conditions for a test (Adzic, 2008 [37]). If I delete some rows from my “Manager” table while testing my management functionality, some of my tests will inevitably fail.

If tests depend on other tests to set up the context, then the order of test execution becomes important and you can no longer run individual tests in isolation. This can lead to big problems, especially if the test runner does not guarantee the order of tests (this is the case with *JUnit*). Again, the tests should ideally set up all the pre-conditions for a test and individual tests should be independent (Adzic, 2008 [37]). I have myself come up against this problem. It is hard to write good tests which correctly run in isolation.

Some programmers have a tendency to “over-mock” their application in order to test it. This result in very complex code and it is easy to become confused, resulting in reduced productivity. I do not suggest that Mock Objects are useless, just that they are a tool which has to be used where appropriate, not a pattern to base the foundations of your entire implementation on.

Consider a convincing quotation from a TDD user. *“Also, I have gone in the trap of thinking I have to test everything. And because I want every test to be isolated, that means a lot of mocking. What I discovered was that my tests dig too deep into the implementation and no longer tests just external behaviour. That means my tests are coupled to the implementation, which is not how it should be.”*

7.4.2 What others think about TDD

The table below (Ritzkopf, 2006 [36]) shows the results of different empirical studies analyzing the code quality and the programmer productivity using TDD.

Müller & *Hagner* (2002)	TDD does neither improve quality nor productivity, but improves program understanding.
Maximilien & *Williams* (2003)	TDD improves quality, but there is hardly any difference in productivity.
George & *Williams* (2003)	While TDD improves quality, it does not produce better productivity.
Geras, *Smith* & *Miller* (2004)	There is no difference in productivity, but TDD encourages developers to write more test cases and to execute them more frequently.
Erdogmus, *Morisio* & *Torchiano* (2005)	TDD encourages programmers to write more tests and a higher number of tests leads to higher productivity. TDD does not improve quality on average, but with the higher number of tests it increases the minimum achievable quality.

Experimental weaknesses and contextual limitations cannot be ignored when viewing the outcome of these studies. The results are quite contradictory. But, generally these studies agree that TDD improves the understanding of the code but does not improve the productivity. At this point, these empirical studies do not deliver a complete evaluation of TDD. Moreover, what remains unclear is how the effectiveness of TDD depends on individual skill level and how long it takes to master this technique. These studies should encourage researchers to extend experiments for further investigations.

7.4.3 What I think about TDD

In my view, TDD requires at least the following skills: discipline, mastery of a unit testing framework, an ability to recognize duplication and to know when and where to refactor. All these skills take time to develop.

It takes time also to get used to this development method. Initially, this form of development is slower, and complex (introducing new concepts, new tools, mock objects, etc.) and changes the habits of programmers. Having used TDD for six months and with an understanding of its underlying concepts, I will still have difficulty fully integrating TDD into my future developments. Indeed, when you are used to traditional programming techniques for years (writing code followed by testing it), it is easy to fall back on your old habits.

But, the use of TDD for developing applications has many advantages. In my opinion despite the studies’ results above, it has a positive impact on programmer productivity, code quality and documentation. It is much more interesting than writing tests after writing the code. TDD forces you to define your interface before you start coding, thus you do not spend time on unimportant features (it is very common among developers to add some functions “just in case”). When you write your code using TDD, each line of your code has been tested. Thus, code coverage is higher and the code quality too.

As far as I am concerned, it not possible to say to another developer “TDD works very well, I advise you to use it!” after working on this project for only six months. I think I need to use TDD for years on a variety of projects.

Anyway, I think it is important to understand Test Driven Development first and how it works to benefit and appreciate it. No methodology is perfect and it depends on people and how they use it. Just because a methodology when applied correctly is appropriate and should result in success, it does not mean that it will be applied correctly and appropriately. Adopting TDD can be very beneficial but it can also be a complete disaster when people get things wrong, especially when lacking experience, motivation or understanding.

8. References

[1] Lasse, K. (2007) Test Driven, Practical TDD and Acceptance TDD for Java developers, Greenwich: Manning Publications

[2] Chaplin, D. (2001) *Test First Programming*, TechZone

[3] Smith, N. (2002) *Write Unit Tests*, Visual Studio Magazine

[4] Marick, B. (2000) *Testing for Programmers*, testing.com

[5] Mackinnon, T. & Freeman S. & Craig P. (2000) *Endo-Testing: Unit Testing with Mock Objects*, conference, "eXtreme Programming and Flexible Processes in Software Engineering - XP2000"

[6] Corbett, N. (??) *Test-driven development, a Portable Methodology*, available from: <http://www.developer.com/design/article.php/10925_3622546_1> [accessed 28 Nov 2007]

[7] Miller, K-W. (2004) *Test-driven development on the cheap: text files and explicit scaffolding*, University of Illinois at Springfield, Dept. of Computer Science

[8] Olan, M. (2003) *Unit testing: test early, test often*, Computer Science and Information Systems, Richard Stockton College

[9] Hammell, T. & Gold , R. & Snyder , T. (2004) *Getting started with Test-driven development*, JavaWorld.com

[10] Ambler, S. (2003) Agile *Database Techniques: Effective Strategies for the Agile Software Developer*, USA: New York

[11] Fowler, M (2006) *Continuous Integration*, Martin Fowler.com

[12] Cogley, J. (2003) *Test-driven development with NUnit and C#*, <http://www.thycotic.com>

[13] Gorman, J. (2005) *Agile Java Development. Test-driven development using JUnit & Eclipse*, Parlez UML

[14] Palermo, J. (2006) *Guidelines for Test-driven development*, Visual Studio 2005 Technical Articles

[15] Definition of "Test-driven development", "EJB", "Dependency injection" and "Reflection", "Session beans", Wikipedia, online encyclopaedia

[16] Hohpe, G. & Istvanick W. & Craig P. (2002) *Test-driven development in Enterprise Integration Project*, ThoughtWorks

[17] Jeffries, R. (2001) *What is Extreme Programming?*, XProgramming.com, XP Magazine

[18] Ranganathan, V. & Pareek, A. (2006) *An Introduction to the Enterprise JavaBeans 3.0 (EJB 3) Specification*, Dev2Dev article

[19] Unknown (2008) *Software Testing and Quality Assurance Glossary*, Applied Testing and Technology, Inc., www.aptest.com [accessed April, 2nd 2008]

[20] Panda, D. (2004) *Simplifying EJB development with EJB 3.0*, The Server side, a Java community

[21] Nygard, M-T. & Karsjens, T. (2000) *Test infect your Enterprise JavaBeans*, Learn how to test your J2EE components live and in the wild, JavaWorld.com

[22] Coffin, R. (2005) *Getting Started with EJB 3.0 and Enterprise Bean Components*, Valtech Technologies: Texas

[23] Verré, C. (2007) *Dependency Injection and Unit Testing*, JavaRanch Journal, Volume 6, Number 1, Managing Editor: Dittmer, U.

[24] Provost, P. (2003) Test-driven development in .NET, available from: <http://www.codeproject.com/dotnet/tdd_in_dotnet.asp> [acceded November, 28th 2007]

[25] Fowler, M. (2004) *Inversion of Control Containers and the Dependency Injection pattern*, Martin Fowler.com

[26] Ben, B. (2008) *Defining terminology, Unit Testing vs. Integration Testing vs. System Testing*, Quickduck

[27] Palermo, J. (2005) *Simple dependency injection to get you started with unit testing*, Software management and CTO, Headspring Systems

[28] Sawyer, D. (2004) *Integration Testing*, article in the edition October 2004 of SQL Server Magazine

[29] Schneider, A. (2001) *JUnit best practices*, Techniques for building resilient, relocatable, multithreaded JUnit tests, JavaWorld.com

[30] Ela, J. (2007) *Order of test,* Answer posted in a thread of “house of fusion” website in the “CFUnit” category, reply to Harris, D.

[31] Keith, M. & Schincariol, M. (2006) *Pro EJB 3: Java Persistence API*, United States of America: Apress

[32] Stott, W. & Newkirk, J. (2004) *Improve the Design and Flexibility of Your Project with Extreme Programming Techniques*, Microsoft MSDN Magazine April

[33] Leponiemi, J. (2003), *Model-View-Controller*, PGUI

[34] Miller, C. (2002) *The Desktop Fishbowl, Six Rules of Unit Testing*, Charles blogs

[35] Enterprise JavaBeans Technology, Java Platform, Enterprise Edition (Java EE), *The Java EE 5 tutorial*, Sun Microsystems (Official website)

[36] Ritzkopf, P. (2006) *Effects of Test Driven Development, an Evaluation of Empirical Studies* Informatik XI, Embedded Software Group, RWTH Aachen

[37] Adzic , G. (2008) *When TDD goes bad*, available from: <http://gojko.net/2008/02/25/when-tdd-goes-bad> [accessed 19 April 2008]

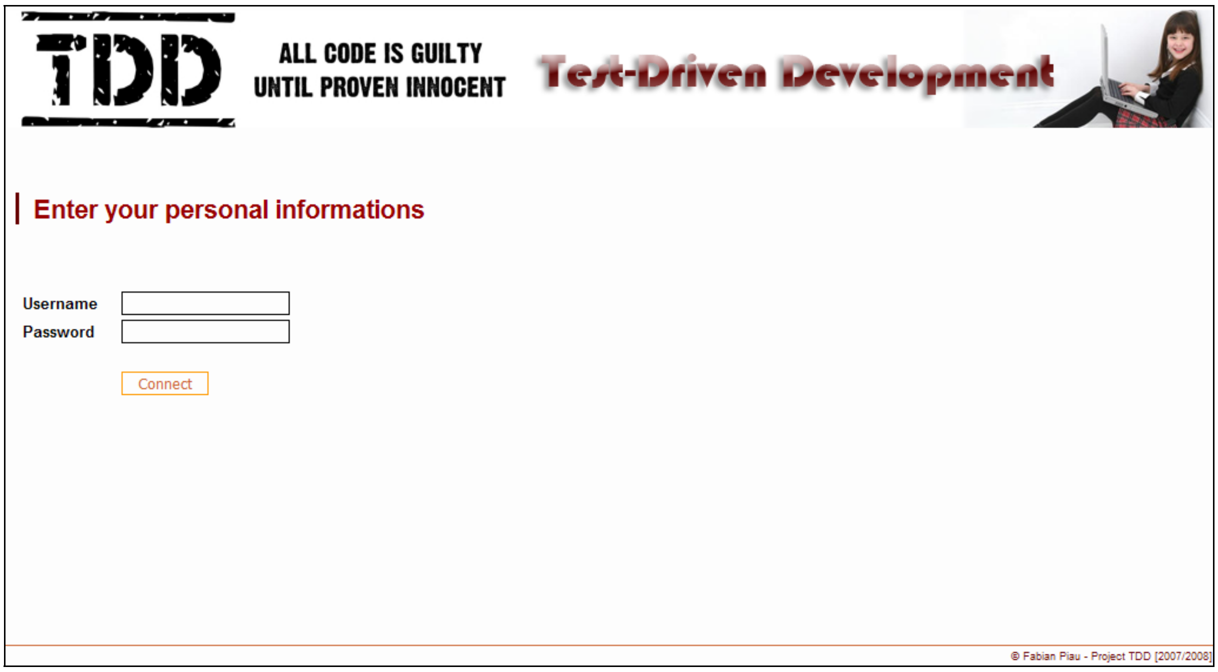
[38] Goyal, D. & Varma, V. (2004) *Introduction to JavaServer Faces (JSF)* Sun Microsystems official presentation

[39] Geary, D. & Horstmann, C. (2007) *Core JavaServer Faces (Second Edition),* Prentice Hall

9. Appendices

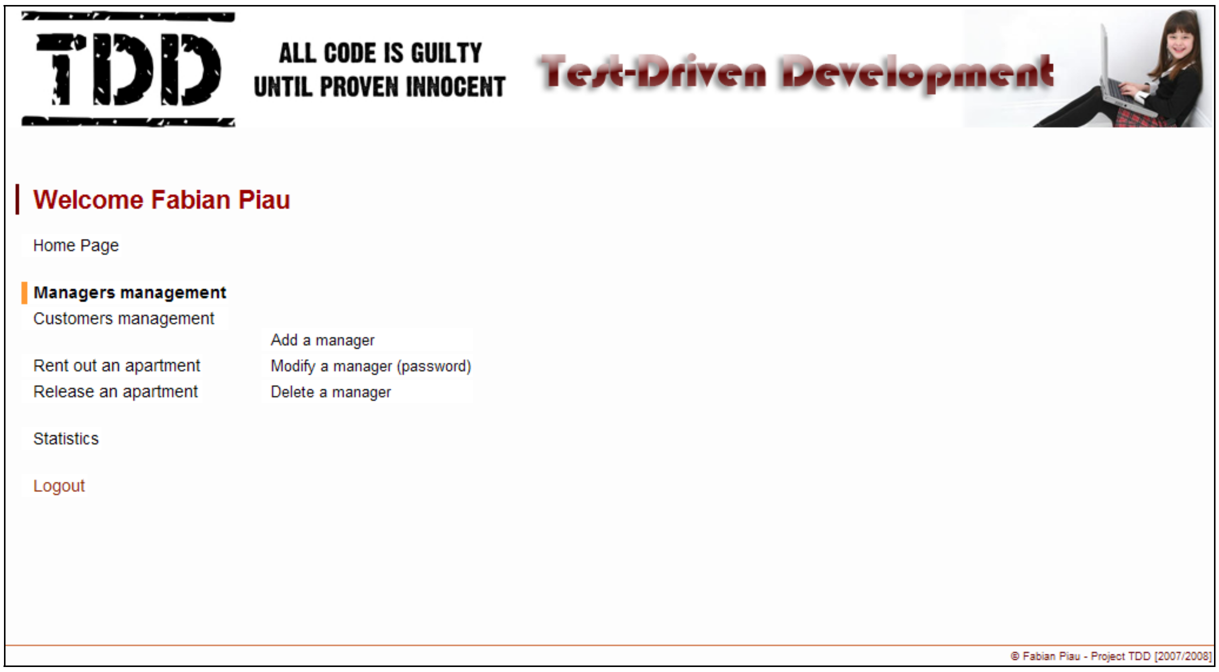
9.1 Sample application screenshots

Here are some screenshots of my final sample application.



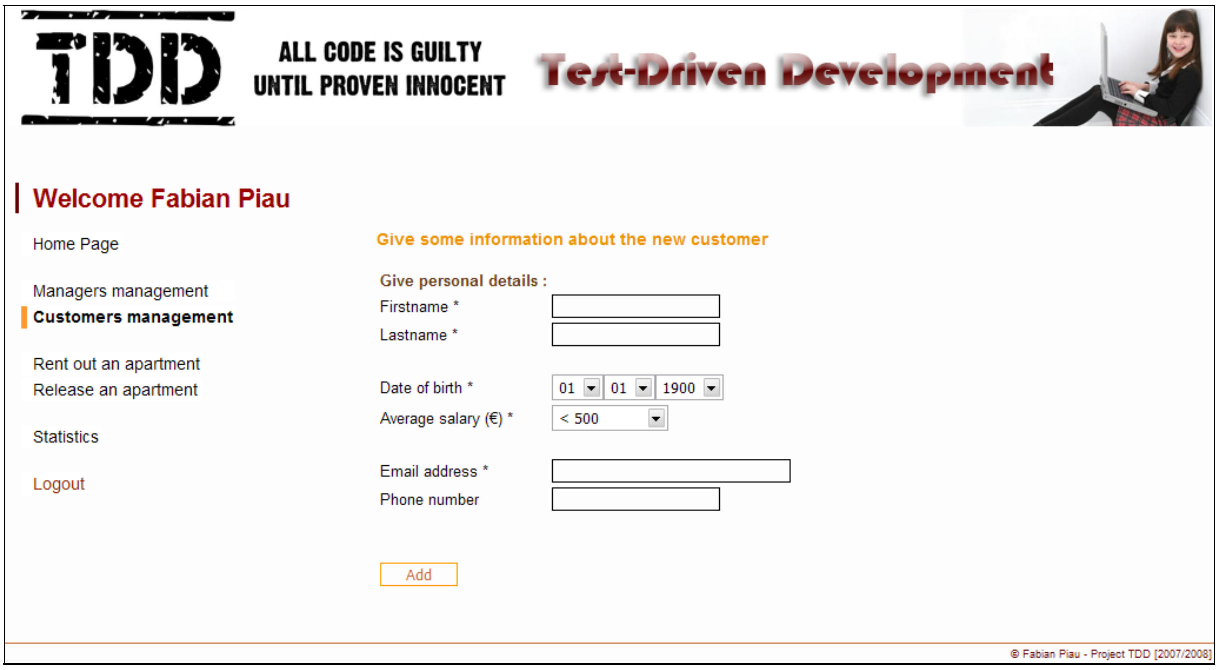
Enter your personal informations

Figure A-9.1-1: Login page



Welcome Fabian Piau

Figure A-9.1-2: Manager management sub-menu



Welcome Fabian Piau

Home Page
Managers management
Customers management
Rent out an apartment
Release an apartment
Statistics
Logout

Give some information about the new customer

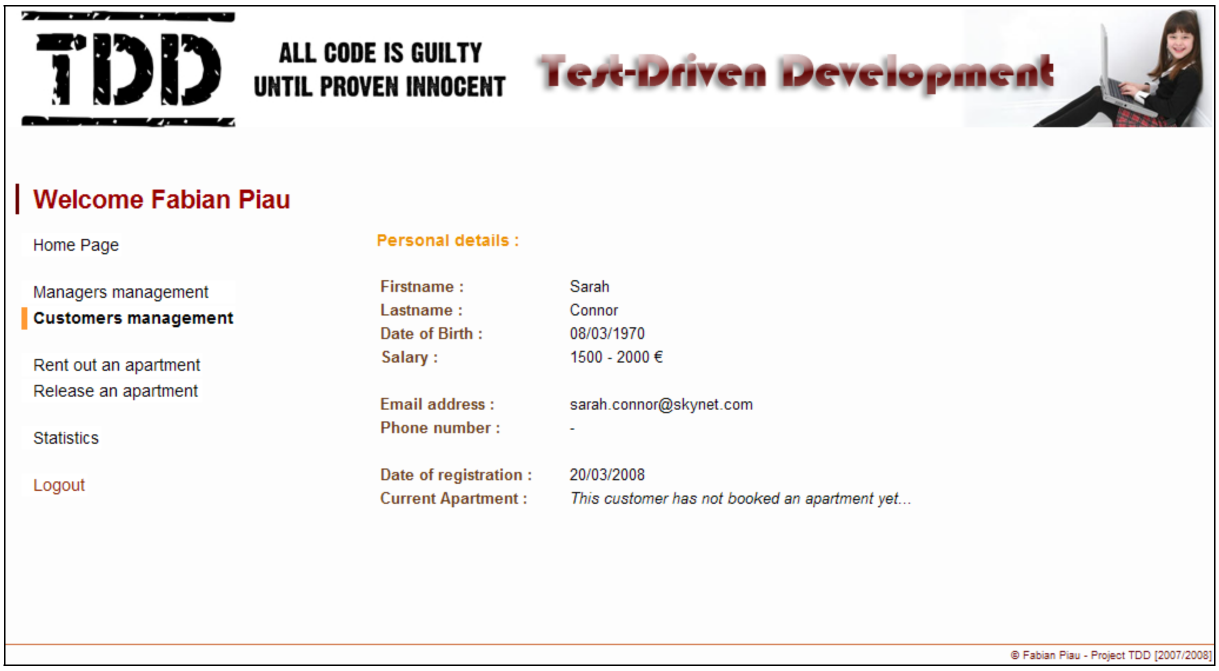
Give personal details :

Firstname *	
Lastname *	
Date of birth *	01 01 1900
Average salary (€) *	< 500
Email address *	
Phone number	

Add

© Fabian Piau - Project TDD [2007/2008]

Figure A-9.1-3: Customer management, add a new customer function



Welcome Fabian Piau

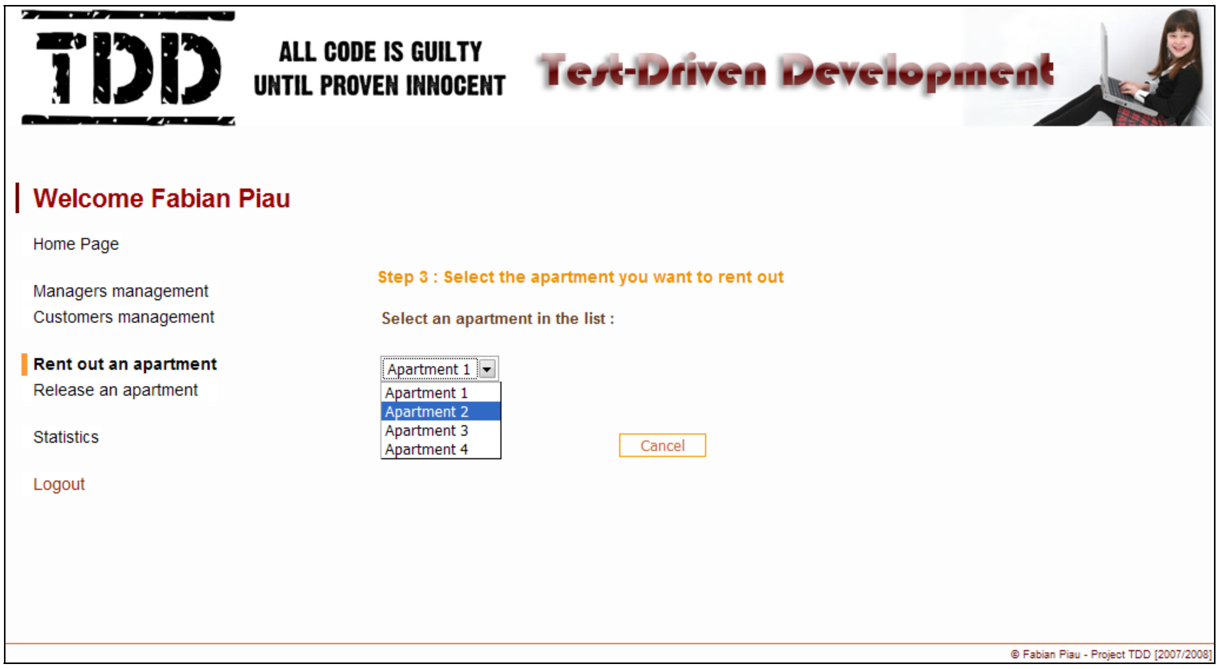
Home Page
Managers management
Customers management
Rent out an apartment
Release an apartment
Statistics
Logout

Personal details :

Firstname :	Sarah
Lastname :	Connor
Date of Birth :	08/03/1970
Salary :	1500 - 2000 €
Email address :	sarah.connor@skynet.com
Phone number :	-
Date of registration :	20/03/2008
Current Apartment :	*This customer has not booked an apartment yet...*

© Fabian Piau - Project TDD [2007/2008]

Figure A-9.1-4: Customer management, view customer function

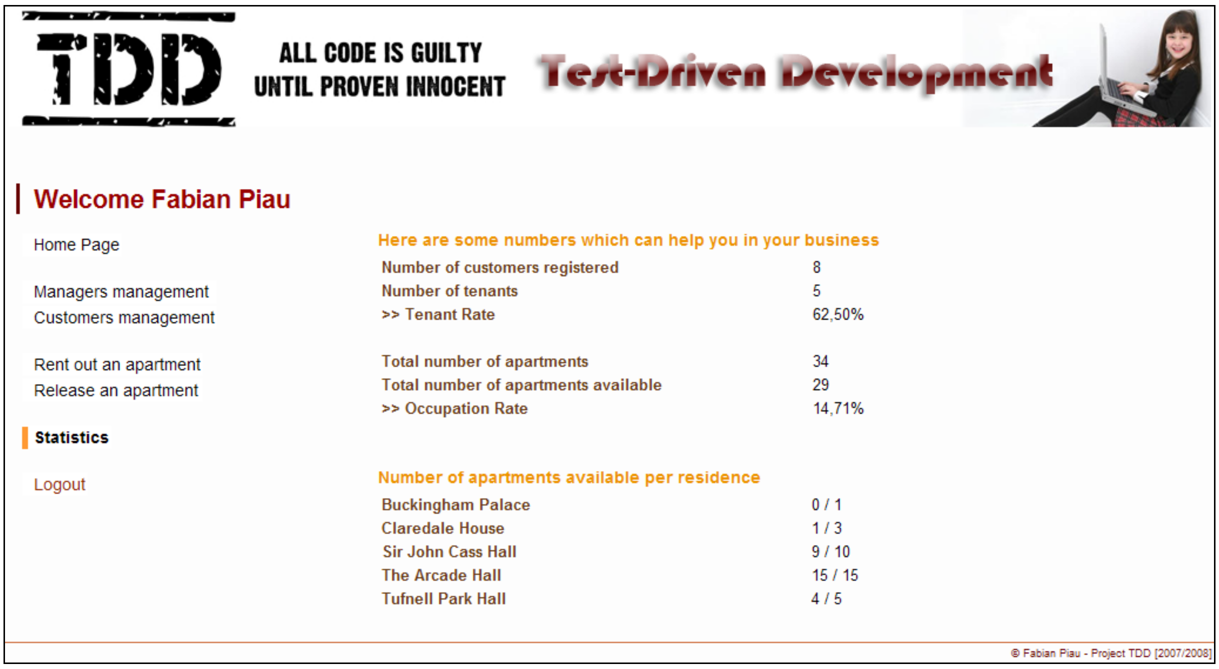


Welcome Fabian Piau

Step 3 : Select the apartment you want to rent out

Select an apartment in the list :

Figure A-9.1-5: Rent out an apartment function, step 3: apartment selection



Welcome Fabian Piau

Here are some numbers which can help you in your business

Number of customers registered	8
Number of tenants	5
>> Tenant Rate	62,50%
Total number of apartments	34
Total number of apartments available	29
>> Occupation Rate	14,71%

Number of apartments available per residence

Buckingham Palace	0 / 1
Claredale House	1 / 3
Sir John Cass Hall	9 / 10
The Arcade Hall	15 / 15
Tufnell Park Hall	4 / 5

Figure A-9.1-6: Statistics function

9.2 Relevant source codes

Interesting source codes of my application in relation with TDD.

UnitTest.java

List of all the tests of my application

```
package model.test;

import javax.naming.Context;
import javax.naming.InitialContext;

import junit.framework.TestCase;

import model.entity.Manager;

import model.stateless.ApartmentServiceBeanMock;
import model.stateless.ApartmentServiceRemoteEJB;
import model.stateless.CustomerServiceBeanMock;
import model.stateless.CustomerServiceRemoteEJB;
import model.stateless.ManagerServiceBeanMock;
import model.stateless.ManagerServiceRemoteEJB;

public class UnitTest extends TestCase {

    private ManagerServiceRemoteEJB ManBean;
    private CustomerServiceRemoteEJB CustBean;
    private ApartmentServiceRemoteEJB AptBean;

    private ManagerServiceRemoteEJB createManagerBean () {
        try {
             final Context context = new InitialContext();
             ManBean =
(ManagerServiceRemoteEJB)context.lookup("ManagerServiceEJB");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        return ManBean;
    }

    private CustomerServiceRemoteEJB createCustomerBean () {
        try {
             final Context context = new InitialContext();
             CustBean =
(CustomerServiceRemoteEJB)context.lookup("CustomerServiceEJB");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        return CustBean;
    }
```

1/3

```
    private ApartmentServiceRemoteEJB createApartmentBean () {
        try {
             final Context context = new InitialContext();
             AptBean =
(ApartmentServiceRemoteEJB)context.lookup("ApartmentServiceEJB");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        return AptBean;
    }


    @Override protected void setUp() {
        //ManBean = new ManagerServiceBeanMock();
        //CustBean = new CustomerServiceBeanMock();
        //AptBean = new ApartmentServiceBeanMock();
        ManBean = createManagerBean();
        CustBean = createCustomerBean();
        AptBean = createApartmentBean();
    }


// TEST VERIFYLOGIN
    public void testMissInformation() {
        assertNull(ManBean.verifyLogin(null, "MyPassword"));
        assertNull(ManBean.verifyLogin("MyUsername", null));
        assertNull(ManBean.verifyLogin(null, null));
    }

    public void testInvalidInformation() {
        assertNull(ManBean.verifyLogin("piauf", "yyyy"));
        assertNull(ManBean.verifyLogin("xxxx", "181186"));
    }

    public void testValidInformation() {
        Manager m = ManBean.verifyLogin("piauf", "181186");
        assertEquals (m.getUsername(), "piauf");
        assertEquals (m.getPassword(), "181186");
    }


// TEST MANAGER ADMINISTRATION
    public void testAdministrationManager() {
        assertNull(ManBean.verifyLogin("lukel", "jumper"));
        // test Add Manager
        assertTrue(ManBean.addManager("Lucky", "Luke", "lukel", "jumper"));
        Manager m = ManBean.verifyLogin("lukel", "jumper");
        assertEquals (m.getUsername(), "lukel");
        assertEquals (m.getPassword(), "jumper");
        // test Add Same Manager
        assertNotNull(ManBean.verifyLogin("lukel", "jumper"));
        assertFalse(ManBean.addManager("Lucky", "Luke", "lukel",
"jumper"));
```

2/3

ManagerServiceBeanmock.java

My own mock object to test the manager functions

```
package model.stateless;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

import javax.ejb.Remove;
import javax.ejb.Stateless;

import javax.persistence.NoResultException;
import javax.persistence.Query;

import model.entity.Manager;

public class ManagerServiceBeanMock implements ManagerServiceRemoteEJB {
    private List<Manager> listManag = new ArrayList<Manager>();

    public ManagerServiceBeanMock() {
       listManag.add(new Manager("Piau", "Fabian", "piauf", "181186"));
       listManag.add(new Manager("Bond", "James", "bondj", "007"));
    }

    public Object mergeEntity(Object entity) {
        return null;
    }

    public Object persistEntity(Object entity){
        return null;
    }

    public List<Manager> queryManagerFindAll(){
        return null;
    }

    public void removeManager(Manager manager){

    }

    public Manager verifyLogin(String username, String password){
        for(Manager m : listManag)
            if (m.getUsername().equals(username) &&
(m.getPassword().equals(password)) ) {
                 return m;
            }
        return null;

        }
```

1/3

```
        // test Delete non-exist Manager
        assertFalse(ManBean.deleteManager("Lucky", "Luc", "lukel"));
        // test Delete Manager
        assertTrue(ManBean.deleteManager("Lucky", "Luke", "lukel"));
        assertNull(ManBean.verifyLogin("lukel", "jumper"));
        // test Delete Same Manager
        assertFalse(ManBean.deleteManager("Lucky", "Luke", "lukel"));

         // test Modify Password Manager
        assertTrue(ManBean.changePasswordManager("Piau", "Fabian", "piauf",
"181186"));
        // test Modify Password non-exist Manager
        assertFalse(ManBean.changePasswordManager("Lucky", "Luke", "lukel",
"jolly"));
    }



// TEST CUSTOMER ADMINISTRATION
    public void testAdministrationCustomer() {
        // test Add Customer
        assertTrue(CustBean.addCustomer("The", "Daltons", "01/05/1950",
"daltons@farwest.fr", "> 3000", ""));
        assertFalse(CustBean.addCustomer("The", "Daltons", "01/01/1900",
"daltons@farwest.com", "< 500", "666666666"));

        // test Delete Customer
        assertTrue(CustBean.deleteCustomer("The", "Daltons"));
        assertFalse(CustBean.deleteCustomer("The", "Daltons"));

        // test See Customer
        // trivial
    }


// TEST APARTMENT ADMINISTRATION
    public void testgetApartmentValidInformation() {
        // test valid apartment
        String info = AptBean.getApartmentInformation(new Long(1));
        assertEquals (info, "Apartment <b>1</b> in the residence <b>Sir
John Cass Hall</b>");
        info = AptBean.getApartmentInformation(new Long(2));
        assertEquals (info, "Apartment <b>2</b> in the residence <b>Sir
John Cass Hall</b>");
    }

    public void testgetApartmentInvalidInformation() {
        // test invalid apartment
        String info = AptBean.getApartmentInformation(new Long(50));
        assertEquals (info, "Error, the appartment this customer has booked
does not exist.");
    }
```

3/3

```
    private Manager existManager (String lastname, String firstname, String 
username) {
        for (Manager m : listManag)
            if (m.getLastname().equals(lastname) && 
(m.getFirstname().equals(firstname)) && (m.getUsername().equals(username))
) {
                 return m;
            }
        return null;

    }

    private Manager existManagerById (Long managerid) {
       return null;
    }

    public Boolean addManager(String lastname, String firstname,
                              String username, String password) {
        if (existManager(lastname, firstname, username) == null) {
            listManag.add(new Manager(lastname, firstname, username, 
password));
            return Boolean.TRUE;
        }
        return Boolean.FALSE;
    }

    public Boolean deleteManager(String lastname, String firstname, String 
username){
        Manager m = existManager(lastname, firstname, username);
        if (m != null) {
            listManag.remove(m);
            return Boolean.TRUE;
        } else
            return Boolean.FALSE;

    }

    public Boolean deleteManagerById(Long managerid){
        return null;
    }

    public Boolean changePasswordManager(String lastname, String firstname,
                                         String username, String password)
{
        Manager m = existManager(lastname, firstname, username);
        if (m != null) {
            m.setPassword(password);
            return Boolean.TRUE;
        } else
            return Boolean.FALSE;
    }
```

2/3

```
    public Boolean changePasswordManagerById(Long managerid, String 
oldpassword, String newpassword) {
        Manager m = existManagerById(managerid);
        if (m != null) {
            m.setPassword(newpassword);
            return Boolean.TRUE;
        } else
            return Boolean.FALSE;
    }


    @Remove
    public void removeBean() {
    }
}
```

3/3

9.3 UML Diagram of the prototype application

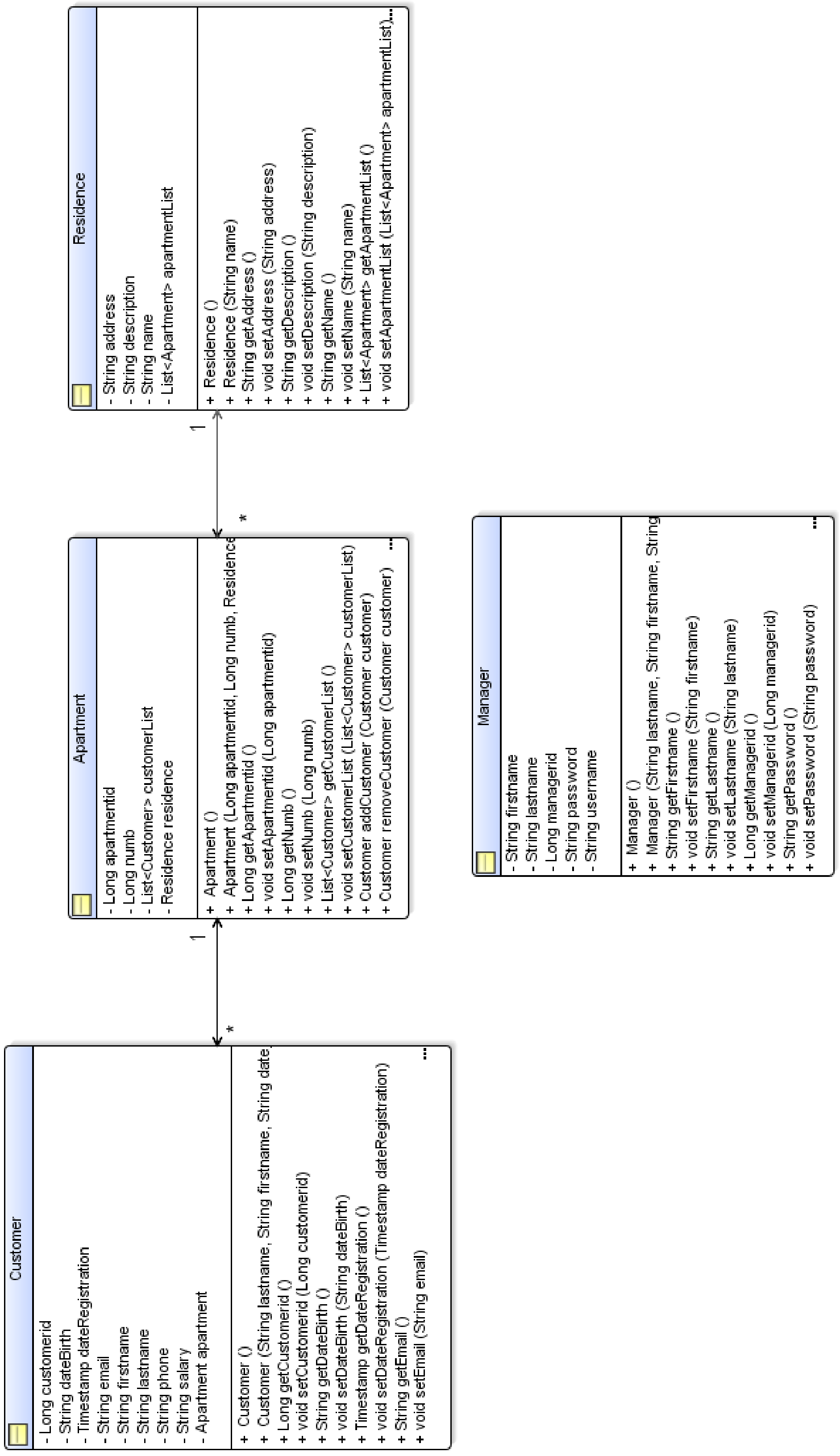

Residence
- String address
- String description
- String name
- List<Apartment> apartmentList
+ Residence ()
+ Residence (String name)
+ String getAddress ()
+ void setAddress (String address)
+ String getDescription ()
+ void setDescription (String description)
+ String getName ()
+ void setName (String name)
+ List<Apartment> getApartmentList ()
+ void setApartmentList (List<Apartment> apartmentList)
1
*
Apartment
- Long apartmentid
- Long numb
- List<Customer> customerList
- Residence residence
+ Apartment ()
+ Apartment (Long apartmentid, Long numb, Residence
+ Long getApartmentid ()
+ void setApartmentid (Long apartmentid)
+ Long getNumb ()
+ void setNumb (Long numb)
+ List<Customer> getCustomerList ()
+ void setCustomerList (List<Customer> customerList)
+ Customer addCustomer (Customer customer)
+ Customer removeCustomer (Customer customer)
1
*
Customer
- Long customerid
- String dateBirth
- Timestamp dateRegistration
- String email
- String firstname
- String lastname
- String phone
- String salary
- Apartment apartment
+ Customer ()
+ Customer (String lastname, String firstname, String date
+ Long getCustomerid ()
+ void setCustomerid (Long customerid)
+ String getDateBirth ()
+ void setDateBirth (String dateBirth)
+ Timestamp getDateRegistration ()
+ void setDateRegistration (Timestamp dateRegistration)
+ String getEmail ()
+ void setEmail (String email)
Manager
- String firstname
- String lastname
- Long managerid
- String password
- String username
+ Manager ()
+ Manager (String lastname, String firstname, String
+ String getFirstname ()
+ void setFirstname (String firstname)
+ String getLastname ()
+ void setLastname (String lastname)
+ Long getManagerid ()
+ void setManagerid (Long managerid)
+ String getPassword ()
+ void setPassword (String password)

Figure A-9.3: Full UML diagram

9.4 Simplified Entity Relationship Diagram of the prototype application

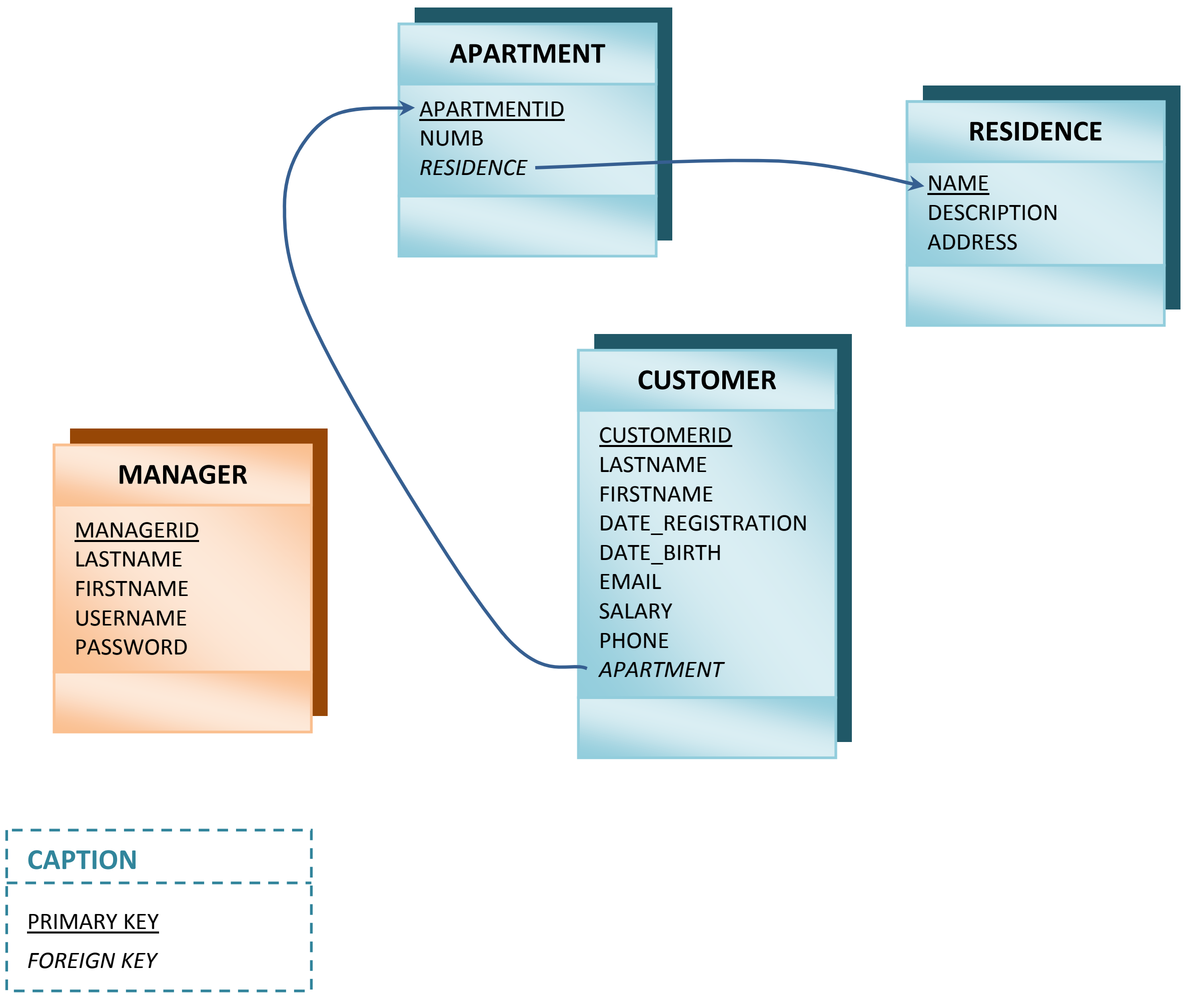


Figure A-9.4: Simplified Entity diagram

9.5 A code refactor example

Before refactoring

```
    public Boolean addManager(String lastname, String firstname,
                              String username, String password) {
        Manager m = new Manager();
        m.setLastname(lastname);
        m.setFirstname(firstname);
        m.setUsername(username);
        m.setPassword(password);

        Manager m2 = new Manager();
        String str = "SELECT m FROM Manager m WHERE m.username  = :username";
        Query query = em.createQuery(str).setParameter("username", username);
        try  {
            m2 =  em.find(Manager.class, ((Manager)
query.getSingleResult()).getManagerid());
        }
        catch(NoResultException e) {
            persistEntity(m);
            return Boolean.TRUE;
        }
        return Boolean.FALSE;
    }

    public Boolean deleteManager(String lastname, String firstname, String username){
        Manager m = new Manager();

        String str = "SELECT m FROM Manager m WHERE m.firstname  = :firstname AND
m.lastname = :lastname AND m.username = :username";
        Query query = em.createQuery(str).setParameter("firstname",
firstname).setParameter("lastname", lastname).setParameter("username", username);
        try  {
            m =  em.find(Manager.class, ((Manager)
query.getSingleResult()).getManagerid());
        }
        catch(NoResultException e) {
            return Boolean.FALSE;
        }
        em.remove(m);
        return Boolean.TRUE;
    }

    public Boolean changePasswordManager(String lastname, String firstname,
                                         String username, String password) {
        Manager m = new Manager();

        String str = "SELECT m FROM Manager m WHERE m.firstname  = :firstname AND
m.lastname = :lastname AND m.username = :username";
        Query query = em.createQuery(str).setParameter("firstname",
firstname).setParameter("lastname", lastname).setParameter("username", username);
        try  {
            m =  em.find(Manager.class, ((Manager)
query.getSingleResult()).getManagerid());
        }
        catch(NoResultException e) {
            return Boolean.FALSE;
        }
        m.setPassword(password);
        return Boolean.TRUE;
    }
```

During refactoring (change, add, deletion)

```
    //  New method to search a manager
    private Manager existManager (String lastname, String firstname, String username) {
        Manager m = new Manager();
        String str = "SELECT m FROM Manager m WHERE m.firstname  = :firstname AND 
m.lastname = :lastname AND m.username = :username";
        Query query = em.createQuery(str).setParameter("firstname", 
firstname).setParameter("lastname", lastname).setParameter("username", username);
        try  {
            m =  em.find(Manager.class, ((Manager) 
query.getSingleResult()).getManagerid());
        }
        catch(NoResultException e) {
            return null;
        }
        return m;
    }

     public Boolean addManager(String lastname, String firstname, 
                              String username, String password) {
        Manager m = new Manager(~~);~~lastname, firstname, username, password);
        // Creation of a new constructor in the manager entity
        ~~m.setLastname(lastname);~~
        ~~m.setFirstname(firstname);~~
        ~~m.setUsername(username);~~
        ~~m.setPassword(password);~~

        ~~Manager m2 = new Manager();~~  // useless variable
        String str = "SELECT m FROM Manager m WHERE m.username  = :username";
        Query query = em.createQuery(str).setParameter("username", username);
        try  {
            ~~m2 =~~  em.find(Manager.class, ((Manager) 
query.getSingleResult()).getManagerid());
        }
        catch(NoResultException e) {
            persistEntity(m);
            return Boolean.TRUE;
        }
        return Boolean.FALSE;
    }

    public Boolean deleteManager(String lastname, String firstname, String username){
        // I call the new method
        Manager m ~~= new Manager();~~ = existManager(lastname, firstname, username);

        ~~String str = "SELECT m FROM Manager m WHERE m.firstname  = :firstname AND~~ 
~~m.lastname = :lastname AND m.username = :username";~~
        ~~Query query = em.createQuery(str).setParameter("firstname",~~ 
~~firstname).setParameter("lastname", lastname).setParameter("username", username);~~
        ~~try  {~~
            ~~m =  em.find(Manager.class, ((Manager)~~ 
~~query.getSingleResult()).getManagerid());~~
        ~~}~~
        ~~catch(NoResultException e) {~~
            ~~return Boolean.FALSE;~~
        ~~}~~
        ~~em.remove(m);~~
        ~~return Boolean.TRUE;~~
        if (m != null) {  // The condition is different because of the new method
            em.remove(m);
            return Boolean.TRUE;
        } else
            return Boolean.FALSE;
    }

    public Boolean changePasswordManager(String lastname, String firstname, 
                                         String username, String password) {
        Manager m ~~= new Manager();~~ = existManager(lastname, firstname, username);
        String str = "SELECT m FROM Manager m WHERE m.firstname  = :firstname AND 
m.lastname = :lastname AND m.username = :username";
```

```
        Query query = em.createQuery(str).setParameter("firstname",
firstname).setParameter("lastname", lastname).setParameter("username", username);
        try  {
            m =  em.find(Manager.class, ((Manager)
query.getSingleResult()).getManagerid());
        }
        catch(NoResultException e) {
            return Boolean.FALSE;
        }
        m.setPassword(password);
        return Boolean.TRUE;
        if (m != null) {  // The condition is different because of the new method
            em.remove(m);
            return Boolean.TRUE;
        } else
            return Boolean.FALSE;
    }
```

After refactoring

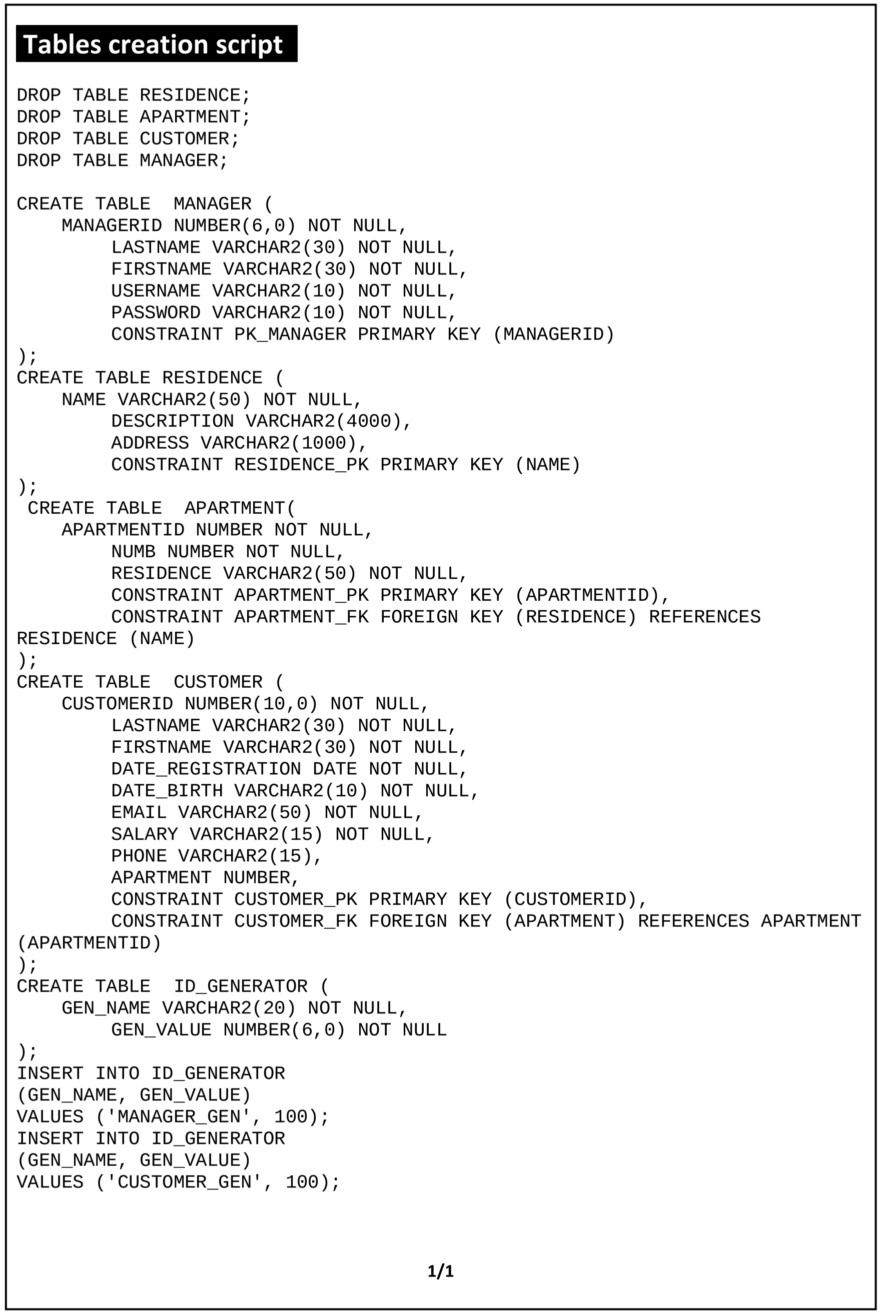
```
    private Manager existManager (String lastname, String firstname, String username) {
        Manager m = new Manager();
        String str = "SELECT m FROM Manager m WHERE m.firstname  = :firstname AND
m.lastname = :lastname AND m.username = :username";
        Query query = em.createQuery(str).setParameter("firstname",
firstname).setParameter("lastname", lastname).setParameter("username", username);
        try  {
            m =  em.find(Manager.class, ((Manager)
query.getSingleResult()).getManagerid());
        }
        catch(NoResultException e) {
            return null;
        }
        return m;
    }

    public Boolean addManager(String lastname, String firstname,
                              String username, String password) {
        Manager m = new Manager(lastname, firstname, username, password);
        String str = "SELECT m FROM Manager m WHERE m.username  = :username";
        Query query = em.createQuery(str).setParameter("username", username);
        try  {
            em.find(Manager.class, ((Manager) query.getSingleResult()).getManagerid());
        }
        catch(NoResultException e) {
            persistEntity(m);
            return Boolean.TRUE;
        }
        return Boolean.FALSE;
    }

    public Boolean deleteManager(String lastname, String firstname, String username){
        Manager m = existManager(lastname, firstname, username);
        if (m != null) {
            em.remove(m);
            return Boolean.TRUE;
        } else
            return Boolean.FALSE;
    }

    public Boolean changePasswordManager(String lastname, String firstname,
                                         String username, String password) {
        Manager m = existManager(lastname, firstname, username);
        if (m != null) {
            m.setPassword(password);
            return Boolean.TRUE;
        } else
            return Boolean.FALSE;
    }
```

9.6 SQL scripts

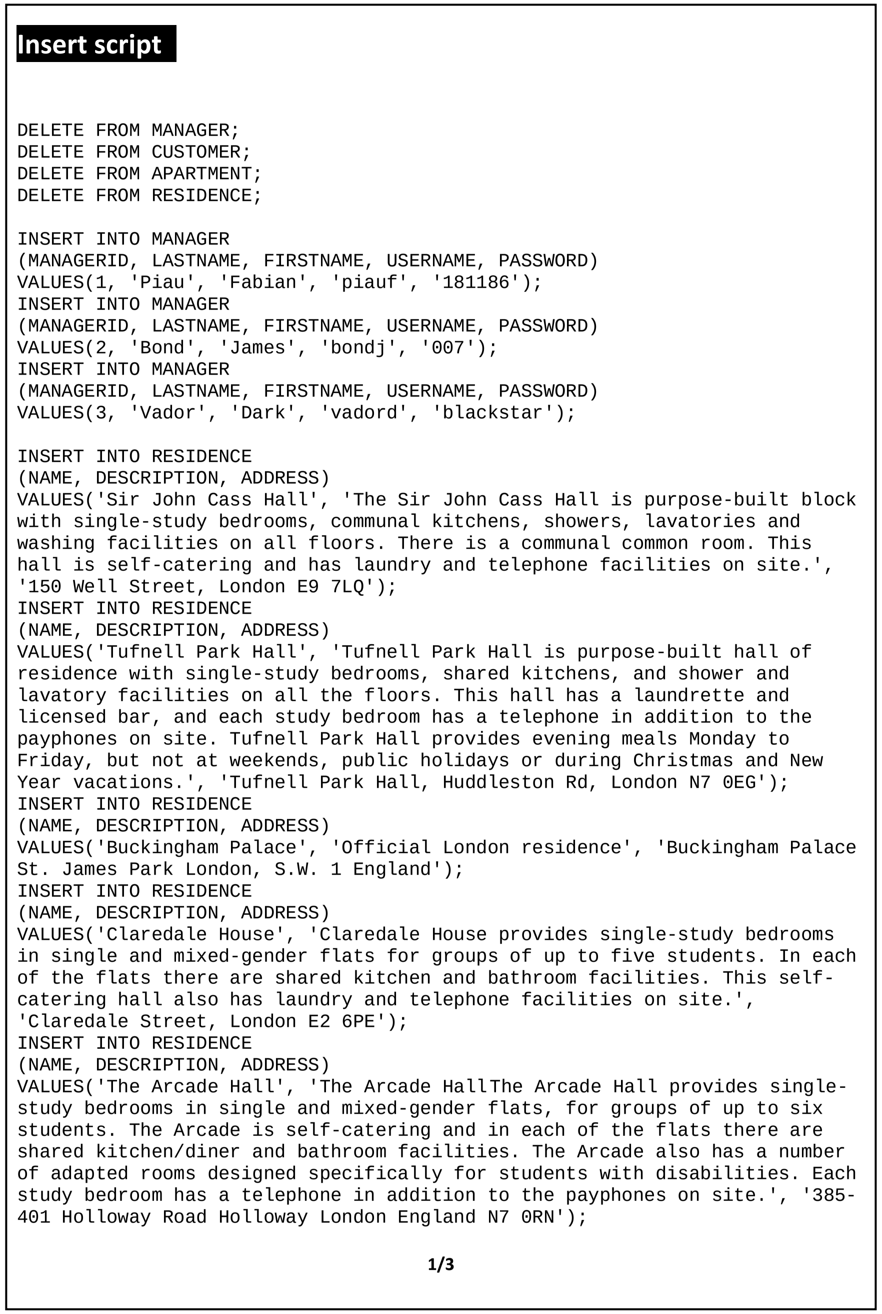


Tables creation script

```
DROP TABLE RESIDENCE;
DROP TABLE APARTMENT;
DROP TABLE CUSTOMER;
DROP TABLE MANAGER;

CREATE TABLE  MANAGER (
    MANAGERID NUMBER(6,0) NOT NULL,
        LASTNAME VARCHAR2(30) NOT NULL,
        FIRSTNAME VARCHAR2(30) NOT NULL,
        USERNAME VARCHAR2(10) NOT NULL,
        PASSWORD VARCHAR2(10) NOT NULL,
        CONSTRAINT PK_MANAGER PRIMARY KEY (MANAGERID)
);
CREATE TABLE RESIDENCE (
    NAME VARCHAR2(50) NOT NULL,
        DESCRIPTION VARCHAR2(4000),
        ADDRESS VARCHAR2(1000),
        CONSTRAINT RESIDENCE_PK PRIMARY KEY (NAME)
);
 CREATE TABLE  APARTMENT(
    APARTMENTID NUMBER NOT NULL,
        NUMB NUMBER NOT NULL,
        RESIDENCE VARCHAR2(50) NOT NULL,
        CONSTRAINT APARTMENT_PK PRIMARY KEY (APARTMENTID),
        CONSTRAINT APARTMENT_FK FOREIGN KEY (RESIDENCE) REFERENCES
RESIDENCE (NAME)
);
CREATE TABLE  CUSTOMER (
    CUSTOMERID NUMBER(10,0) NOT NULL,
        LASTNAME VARCHAR2(30) NOT NULL,
        FIRSTNAME VARCHAR2(30) NOT NULL,
        DATE_REGISTRATION DATE NOT NULL,
        DATE_BIRTH VARCHAR2(10) NOT NULL,
        EMAIL VARCHAR2(50) NOT NULL,
        SALARY VARCHAR2(15) NOT NULL,
        PHONE VARCHAR2(15),
        APARTMENT NUMBER,
        CONSTRAINT CUSTOMER_PK PRIMARY KEY (CUSTOMERID),
        CONSTRAINT CUSTOMER_FK FOREIGN KEY (APARTMENT) REFERENCES APARTMENT
(APARTMENTID)
);
CREATE TABLE  ID_GENERATOR (
    GEN_NAME VARCHAR2(20) NOT NULL,
        GEN_VALUE NUMBER(6,0) NOT NULL
);
INSERT INTO ID_GENERATOR
(GEN_NAME, GEN_VALUE)
VALUES ('MANAGER_GEN', 100);
INSERT INTO ID_GENERATOR
(GEN_NAME, GEN_VALUE)
VALUES ('CUSTOMER_GEN', 100);
```

1/1



Insert script

```
DELETE FROM MANAGER;
DELETE FROM CUSTOMER;
DELETE FROM APARTMENT;
DELETE FROM RESIDENCE;

INSERT INTO MANAGER
(MANAGERID, LASTNAME, FIRSTNAME, USERNAME, PASSWORD)
VALUES(1, 'Piau', 'Fabian', 'piauf', '181186');
INSERT INTO MANAGER
(MANAGERID, LASTNAME, FIRSTNAME, USERNAME, PASSWORD)
VALUES(2, 'Bond', 'James', 'bondj', '007');
INSERT INTO MANAGER
(MANAGERID, LASTNAME, FIRSTNAME, USERNAME, PASSWORD)
VALUES(3, 'Vador', 'Dark', 'vadord', 'blackstar');

INSERT INTO RESIDENCE
(NAME, DESCRIPTION, ADDRESS)
VALUES('Sir John Cass Hall', 'The Sir John Cass Hall is purpose-built block
with single-study bedrooms, communal kitchens, showers, lavatories and
washing facilities on all floors. There is a communal common room. This
hall is self-catering and has laundry and telephone facilities on site.',
'150 Well Street, London E9 7LQ');
INSERT INTO RESIDENCE
(NAME, DESCRIPTION, ADDRESS)
VALUES('Tufnell Park Hall', 'Tufnell Park Hall is purpose-built hall of
residence with single-study bedrooms, shared kitchens, and shower and
lavatory facilities on all the floors. This hall has a laundrette and
licensed bar, and each study bedroom has a telephone in addition to the
payphones on site. Tufnell Park Hall provides evening meals Monday to
Friday, but not at weekends, public holidays or during Christmas and New
Year vacations.', 'Tufnell Park Hall, Huddleston Rd, London N7 0EG');
INSERT INTO RESIDENCE
(NAME, DESCRIPTION, ADDRESS)
VALUES('Buckingham Palace', 'Official London residence', 'Buckingham Palace
St. James Park London, S.W. 1 England');
INSERT INTO RESIDENCE
(NAME, DESCRIPTION, ADDRESS)
VALUES('Claredale House', 'Claredale House provides single-study bedrooms
in single and mixed-gender flats for groups of up to five students. In each
of the flats there are shared kitchen and bathroom facilities. This self-
catering hall also has laundry and telephone facilities on site.',
'Claredale Street, London E2 6PE');
INSERT INTO RESIDENCE
(NAME, DESCRIPTION, ADDRESS)
VALUES('The Arcade Hall', 'The Arcade HallThe Arcade Hall provides single-
study bedrooms in single and mixed-gender flats, for groups of up to six
students. The Arcade is self-catering and in each of the flats there are
shared kitchen/diner and bathroom facilities. The Arcade also has a number
of adapted rooms designed specifically for students with disabilities. Each
study bedroom has a telephone in addition to the payphones on site.', '385-
401 Holloway Road Holloway London England N7 0RN');
```

1/3

```
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(1, 1, 'Sir John Cass Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(2, 2, 'Sir John Cass Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(3, 3, 'Sir John Cass Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(4, 1, 'Tufnell Park Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(5, 2, 'Tufnell Park Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(6, 3, 'Tufnell Park Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(7, 4, 'Tufnell Park Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(8, 5, 'Tufnell Park Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(9, 1, 'Buckingham Palace');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(19, 10, 'Claredale House');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(20, 1, 'The Arcade Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(21, 2, 'The Arcade Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(22, 3, 'The Arcade Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(23, 4, 'The Arcade Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(24, 5, 'The Arcade Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(25, 6, 'The Arcade Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(26, 7, 'The Arcade Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES // etc.
```

2/3

```
// ...
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(29, 10, 'The Arcade Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(30, 11, 'The Arcade Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(31, 12, 'The Arcade Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(32, 13, 'The Arcade Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(33, 14, 'The Arcade Hall');
INSERT INTO APARTMENT
(APARTMENTID, NUMB, RESIDENCE)
VALUES(34, 15, 'The Arcade Hall');

INSERT INTO CUSTOMER
(CUSTOMERID, LASTNAME, FIRSTNAME, DATE_REGISTRATION, DATE_BIRTH, EMAIL,
SALARY)
VALUES(1, 'Connor', 'Sarah', '20/03/08', '08/03/1970',
'sarah.connor@skynet.com', '1500 - 2000');
INSERT INTO CUSTOMER
(CUSTOMERID, LASTNAME, FIRSTNAME, DATE_REGISTRATION, DATE_BIRTH, EMAIL,
SALARY, APARTMENT)
VALUES(2, 'Balboa', 'Rocky', '19/03/08', '21/09/1965', 'rocky@punch.com',
'1000 - 1500', 2);
INSERT INTO CUSTOMER
(CUSTOMERID, LASTNAME, FIRSTNAME, DATE_REGISTRATION, DATE_BIRTH, EMAIL,
SALARY, APARTMENT)
VALUES(3, 'Potter', 'Harry', '20/03/08', '23/11/1913', 'harry@magic.com',
'< 500', 11);
INSERT INTO CUSTOMER
(CUSTOMERID, LASTNAME, FIRSTNAME, DATE_REGISTRATION, DATE_BIRTH, EMAIL,
SALARY, PHONE, APARTMENT)
VALUES(4, 'Wayne', 'Bruce', '18/03/08', '09/07/1966', 'batman@bat.fr', '>
3000', '0856741246', 3);
INSERT INTO CUSTOMER
(CUSTOMERID, LASTNAME, FIRSTNAME, DATE_REGISTRATION, DATE_BIRTH, EMAIL,
SALARY, PHONE)
VALUES(5, 'Jones', 'Indiana', '22/03/08', '19/06/1954',
'indiana@jones.com', '2000 - 2500', '0542156749');
INSERT INTO CUSTOMER
(CUSTOMERID, LASTNAME, FIRSTNAME, DATE_REGISTRATION, DATE_BIRTH, EMAIL,
SALARY, PHONE, APARTMENT)
VALUES(6, 'Jobs', 'Steve', '22/03/08', '29/10/1968', 'director@apple.com',
'> 3000', '0652781265', 8);
INSERT INTO CUSTOMER
(CUSTOMERID, LASTNAME, FIRSTNAME, DATE_REGISTRATION, DATE_BIRTH, EMAIL,
SALARY, APARTMENT)
VALUES(7, 'Kong', 'King', '22/03/08', '02/05/1900', 'king-kong@beast.org',
'< 500', 9);
```

3/3

9.7 How to deploy and start the sample application

At least, you need a database and also the *Java Enterprise Edition Development Toolkit* installed on your system.

The database I have used is *Oracle Database 10g Express Edition** (Oracle Database XE) which is entirely free. To have more information and if you want to download this software, you can visit this link <http://www.oracle.com/technology/products/database/xe/index.html>.

To download *Java Enterprise Edition*, the link is <http://java.sun.com/javaee/downloads/index.jsp> and select the *Java EE SDK* to download.

Once you have installed all this software on your system, you have to prepare the database. I created for you two automated scripts*. You will find the files at “TDD_Project\Data_files”. Simply run “CreateTables.sql” then “Insert.sql” in the SQL client from your database.

The second step is to start the server:

1. You can use Oracle’s standalone application server, oc4j. This server may be downloaded from Oracle’s website <http://www.oracle.com/technology/tech/java/oc4j/index.html>, or you may use the oc4j standalone server that comes with *JDeveloper* if it already installed on your machine.
2. Open the terminal (Start → Execute → type “`cmd`” → ok)
3. Change to the folder where is located your server. If you use the oc4j standalone server that comes with *JDeveloper*, the server is located in the folder *jdev_home*/j2ee/home and is packaged in the file oc4j.jar. (Note: *jdev_home* represents the folder where you have installed *JDeveloper*) Type “`cd` *`jdev_home`*`/j2ee/home`”
4. Then, type “`java -jar oc4j.jar`”
5. The first time you run the server you will be asked for a username and a password. Type “`tdd2008`” for example (Note: you need character AND numbers in the password, it is important that you remember this password)

Now you can deploy and execute the sample application:

1. Copy the folder “TDD_Project” on your hard drive
2. Open the terminal
3. Change the path to TDD_Project.
   Type “`cd` <*where you have copied the folder*>`\TDD_Project`”
4. Type “`ant clean`” to ensure a clean build
5. Type “`ant`” to compile, package and deploy EJB, Web and Client modules
6. Type “`ant run`” to run the application
7. Start your browser and go to the following URL :
   `http://localhosat:8888/tdd`

You will see the login page (Figure A-9.1-1), if you have some errors that you do not arrive to solve. You can try the second method on the following page.

* Note that you can choose another database, for example *MySQL Database* but you certainly will have to do some modifications in the SQL Scripts (creation of tables and insertion of data).

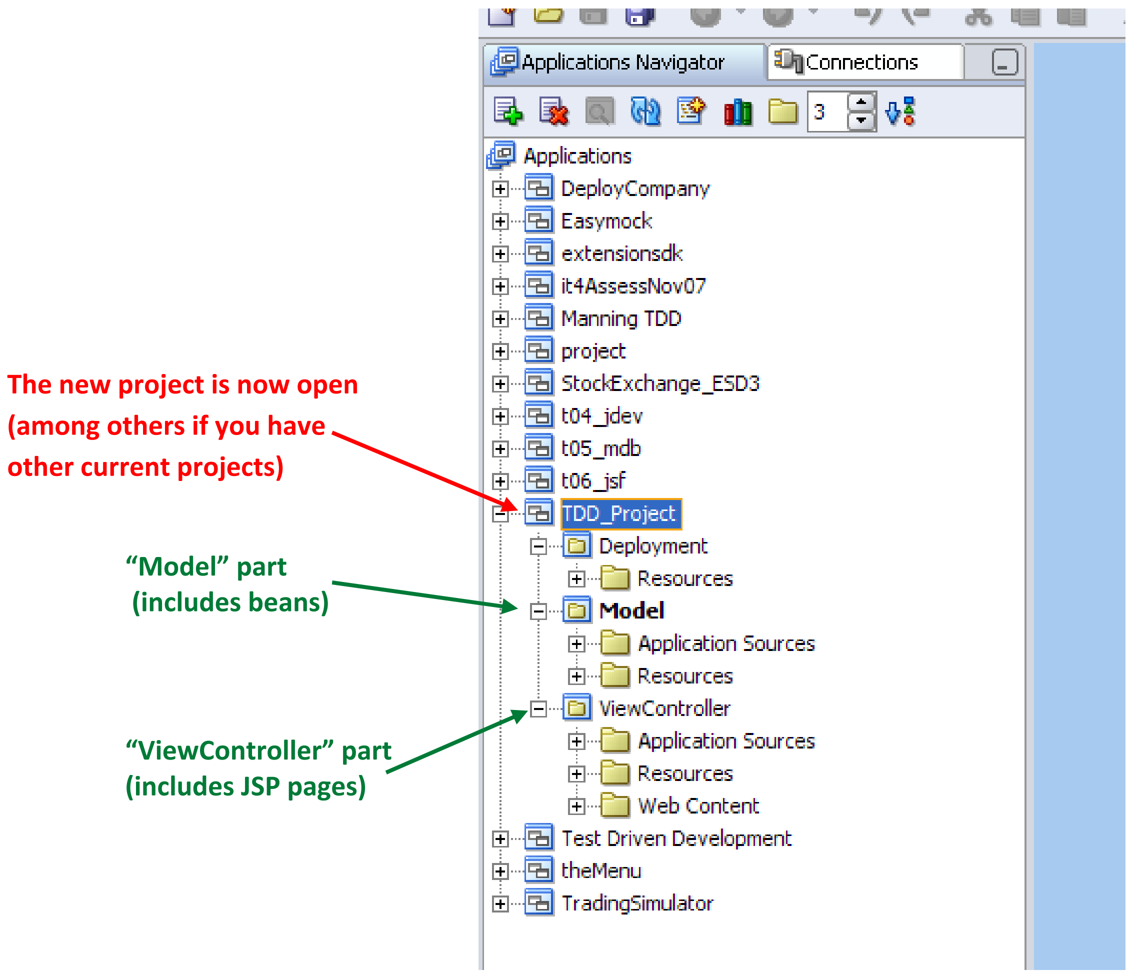
9.8 How to deploy and start the sample application using JDeveloper

If you have some difficulties to run the application with the first method, you can use this second method which will be simpler but need *JDeveloper* installed on your system.

First, follow the instruction of the first method until the second step. I think you already did it if you have tried the first method unsuccessfully.

You need to install *JDeveloper* which is a free IDE like *Eclipse* but developed by *Oracle*. You can download it at <http://www.oracle.com/technology/products/jdev/index.html>.

Once *JDeveloper* installed on your system and your database properly configured, open the *TDD_Project.jws* file into *JDeveloper*. You will find this file at the root of the folder "TDD_Project\". This will open the entire project including the *JavaServer Pages* and entities used in the application.


The new project is now open (among others if you have other current projects)
"Model" part (includes beans)
"ViewController" part (includes JSP pages)
Applications Navigator
Connections

```
Applications
+ DeployCompany
+ Easymock
+ extensionsdk
+ it4AssessNov07
+ Manning TDD
+ project
+ StockExchange_ESD3
+ t04_jdev
+ t05_mdb
+ t06_jsf
- TDD_Project
    - Deployment
        + Resources
    - Model
        + Application Sources
        + Resources
    - ViewController
        + Application Sources
        + Resources
        + Web Content
+ Test Driven Development
+ theMenu
+ TradingSimulator
```

Figure A-9.8: The project is opened in JDeveloper

Now, you need to configure the connection to your database in *JDeveloper*. In the following, I give you the configuration for an *Oracle Database 10g Express Edition*. If you use any other databases, refer to internet to have more explanations about how configuring your own database with *JDeveloper*.

Click the “Connections” tab (if not visible, use View / Connections Navigator). Right click on “Database”, select “New Database Connection...”, and follow the wizard:

1. **Type tab**
   Connection name: tdd (or any name you wish to use)
   Connection Type: Oracle (JDBC)

2. **Authentification tab**
   Username: the username of your database
   Password: the password of your database
   Role: leave blank
   Deploy Password: tick the box

3. **Connection tab**
   Driver: thin
   Host Name: localhost
   Port: 1521
   SID: XE
   Service name: leave blank

4. **Test tab**
   Test connection: Should display “Success”

Open your new connection and view tables (Structure & Data) to ensure that your connection to your database functions properly.

After all configuration steps are completed, deploy the business components into the server. To deploy the components, run any of the beans/entities in the project’s *Model* part (right click on the component → Run). This will deploy all the components to the embedded OC4J server. To access the *JavaServer Pages*, run the *index.jsp* page in the same way. You could find all the JSP pages in the V*iewController* part.
It will open directly your browser at the good address and the login page (Figure A-9.1-1) will be displayed.

9.9 Glossary

Code coverage	Code coverage is a measure used in software testing. It describes the degree to which the source code of a program has been tested.
Code refactoring	Code refactoring is any change to a computer program's code that improves its readability or simplifies its structure without changing its results.
Component	A minimal software item for which a separate specification is available.
Integration Testing	Testing of combined parts of an application to determine if they function together correctly. Usually performed after unit testing.
Testability	The degree to which a system or component facilitates the establishment of test criteria.
Test Case	Test Case is a commonly used term for a specific test. This is usually the smallest unit of testing.
Test Suite	A collection of tests used to validate the behaviour of a product or functionality. In most cases a Test Suite is a high level concept, grouping together hundreds or thousands of tests related by what they are intended to test.
Test Tools	Computer programs used in the testing of a system, a component of the system... e.g. *JUnit*.
Unit Testing	Testing of individual software components.

9.10 Acronyms

API	**A**pplication **P**rogramming **I**nterface
EJB	**E**nterprise **J**ava**B**eans
GUI	**G**raphical **U**ser **I**nterface
IDE	**I**ntegrated **D**evelopment **E**nvironment
J2EE	**J**ava **E**nterprise **E**dition version 1.**4** or less
J5EE	**J**ava **E**nterprise **E**dition version 1.**5**
JDBC	**J**ava **D**ata**b**ase **C**onnectivity
JNDI	**J**ava **N**aming and **D**irectory **I**nterface
JSF	**J**ava**S**erver **F**aces
JSP	**J**ava**S**erver **P**ages
MVC	**M**odel **V**iew **C**ontroller
TDD	**T**est-**d**riven **d**evelopment
TFD	**T**est-**f**irst **d**evelopment
UI	**U**ser **I**nterface
SQL	**S**tructured **Q**uery **L**anguage
XP	E**x**treme **P**rogramming