A laboratory exercise in testing database applications

Javier Tuya, Claudio de la Riva, José García-Fanjul

University of Oviedo

Dpto de Informática – Gijón / SPAIN

tuya@uniovi.es, claudio@uniovi.es, jgfanjul@uniovi.es

Abstract

We describe a laboratory exercise, developed in small teams, that integrates the design of unit test cases, functional test cases and the use of support tools (JUnit, CVS, and a helpdesk system).

1. Introduction

For years we have been involved in teaching several Software Engineering courses, including a module of software testing. However, due to the time assignments, software testing includes only the main concepts along with black-box and white-box techniques. Exercises carried out by the students include the design of tests form a short specification or a piece of source code. This prevents the students from getting a bigger picture of testing, as they do not experience the issues of testing in the context of an application and working in teams.

On the other hand, most of the applications that our students will work with after graduating have a significant database component, and so, they must deal with the particularities of testing in this context. The program inputs and outputs are the database itself, and few database loads (test cases) must be carefully designed to exercise all SQL queries.

Two years ago, we designed an integrated laboratory practice for students to practice all of the above issues, which are described below.

2. Description

This laboratory exercise simulates the life cycle of testing and maintenance of a software application in a software engineering course for graduate students. It spans along a semester: 12 open laboratory hours grouped into 6 sessions, plus some additional homework. Before beginning the exercise they receive four additional hours of training on testing SQL and the tools that they will use in the exercise, plus two laboratory hours for practicing the use of the tools.

2.1. Tested artifacts

The students work in a small application for payments by direct debit, which is structured into two tiers: user interface written in Java/Swing and business objects written in Java). The architecture is depicted in Figure 1.

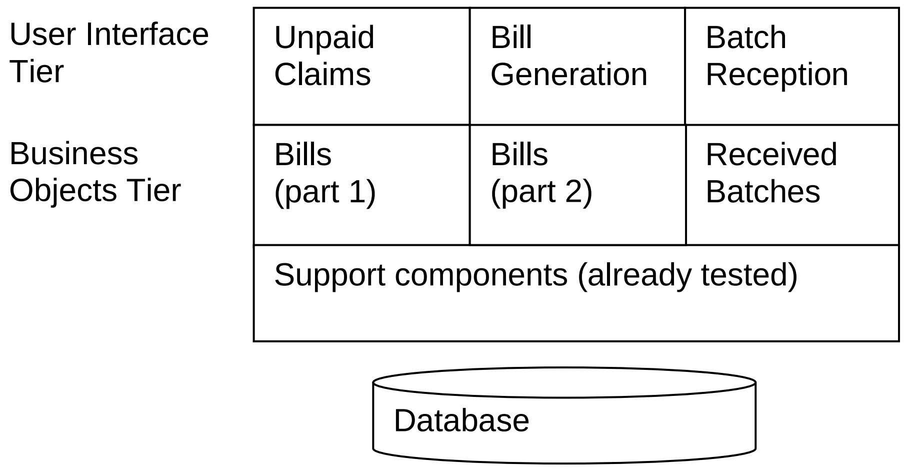
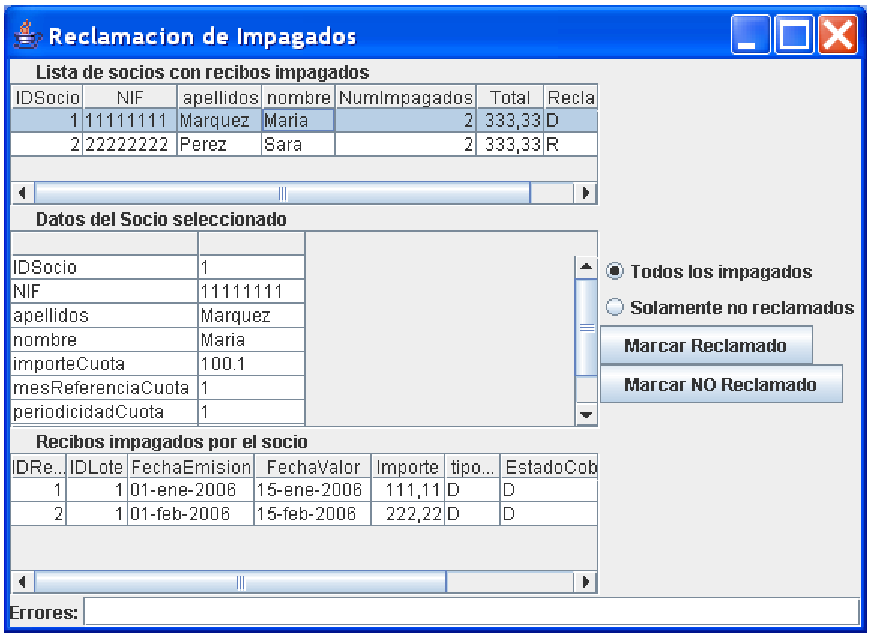


Figure 1: Architecture of the application under test

Each of the tiers is divided into modules (3 modules at each tier) which are the basic unit of assignment to the students. The database comprises five tables.

Each module has some faults that have been injected by the instructor, but unknown to the students. The injected faults combine some easy to detect bugs with more complex ones, but in all cases, they can be corrected by performing small changes in either the java code or the SQL statements.

Figures 2, 3 and 4 present the user interface for each of the three modules.



Reclamacion de Impagados

Lista de socios con recibos impagados

IDSocio	NIF	apellidos	nombre	NumImpagados	Total	Recla
1	11111111	Marquez	Maria	2	333,33	D
2	22222222	Perez	Sara	2	333,33	R

Datos del Socio seleccionado

IDSocio	1
NIF	11111111
apellidos	Marquez
nombre	Maria
importeCuota	100.1
mesReferenciaCuota	1
periodicidadCuota	1

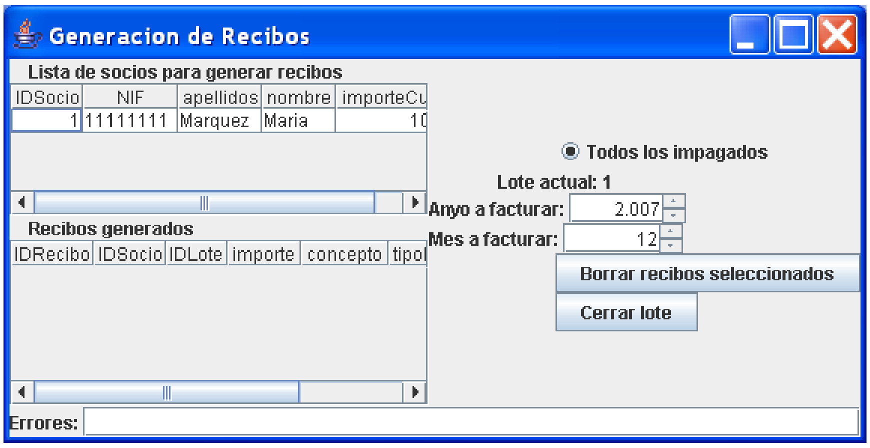
[✓ Todos los impagados] Solamente no reclamados

Recibos impagados por el socio

IDRe...	IDLote	FechaEmision	FechaValor	Importe	tipo...	EstadoCob
1	1	01-ene-2006	15-ene-2006	111,11	D	D
2	1	01-feb-2006	15-feb-2006	222,22	D	D

Errores:

Figure 2: User Interface for Unpaid Claims


Generacion de Recibos

Lista de socios para generar recibos

IDSocio	NIF	apellidos	nombre	importeCu
1	11111111	Marquez	Maria	10

Todos los impagados

Lote actual: 1

Anyo a facturar: 2.007

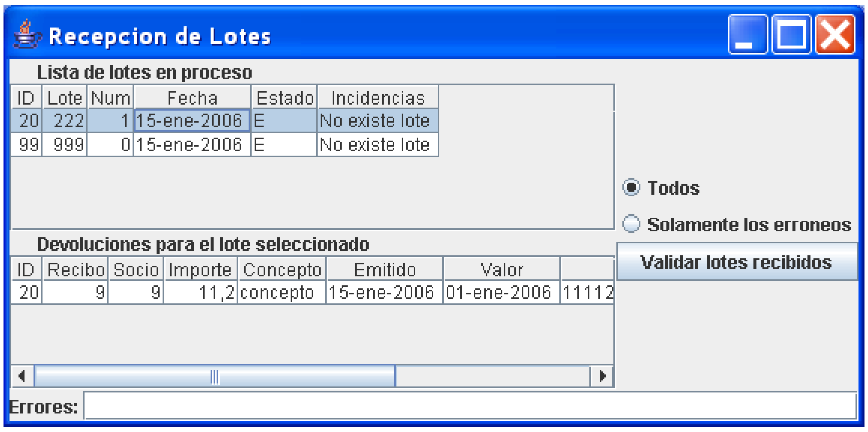
Mes a facturar: 12

Recibos generados

IDRecibo	IDSocio	IDLote	importe	concepto	tipol

Errores:

Figure 3: User Interface for Bill Generation


Recepcion de Lotes

Lista de lotes en proceso

ID	Lote	Num	Fecha	Estado	Incidencias
20	222	1	15-ene-2006	E	No existe lote
99	999	0	15-ene-2006	E	No existe lote

[✓ Todos] Solamente los erroneos

Devoluciones para el lote seleccionado

ID	Recibo	Socio	Importe	Concepto	Emitido	Valor	
20	9	9	11,2	concepto	15-ene-2006	01-ene-2006	11112

Errores:

Figure 4: User Interface for Batch Reception

Each student receives two assignments for testing:

- A business component to perform unit tests
- A user interface component to perform functional tests

In this way, the entire class is organized in teams composed by three students. Each team has an independent copy of the application, which is visible and shared by all members of the team.

All source code is available to the students plus some documentation composed by:

- Work procedures (3 pages): Describes the whole testing and maintenance process.
- Work instructions (6 pages): Provides details about how to configure the environment and basic instructions for testing and using the tools.
- Use cases (2 pages): Textual description of the functionality of each module.
- Data model (3 pages): Database schema and data dictionary.

2.2. Testing tools

One goal of the exercise (although secondary) is to train the students in the use of a development environment that integrates the following tools:

- Eclipse with JUnit: to automate the tests (both for business and user interface components).
- CVS: for sharing the code and test cases among the members of each team.
- Helpdesk: A software bug reporting database to notify the faults detected and control the process.
- Clover: to measure code coverage.
- Data load support methods: Part of the support components are a set of simple methods to load data in the database, which facilitates this task and avoids the need to use additional tools like DBUnit.

2.3. Test and Maintenance Process

The process is depicted in Figure 5. Each assignment is sent to each student by way of a ticket recorded in the Helpdesk system, which controls the workflow of the process. During the process the student will change the state of each of the assigned tickets, place annotations and send them to another student. The final goal will be to achieve that all assigned tickets be closed by following the process that is depicted in Figure 2 and described below.

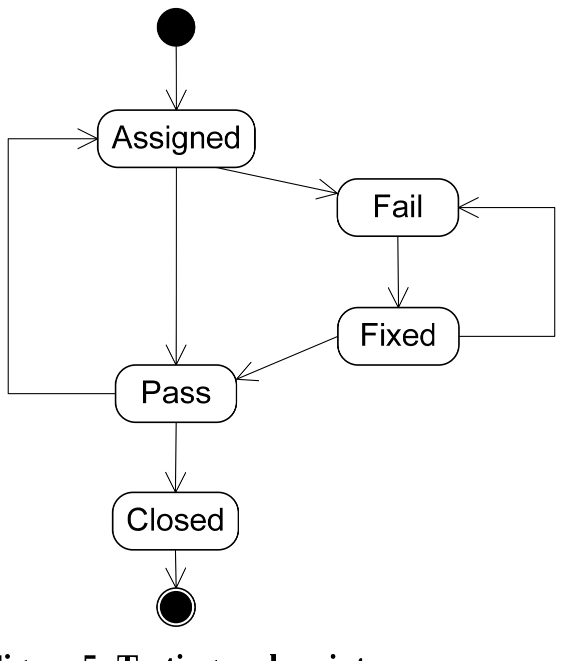


Figure 5: Testing and maintenance process

Students play alternatively the roles of tester and developer for different components as described below:

- The tester receives a ticket including the assignment for testing a module (state *Assigned*). He must check-out the project components, develop the tests and automate them using JUnit.
- Once finished the testing he will check-in the project including the test cases and will notify the problems found. If some failure is detected he records a bug description in the ticket and forwards it to the developer (state *Fail*). If there are no failures detected he forwards it to the instructor (state *Pass*)
- When the developer receives a ticket in state *Fail* he will correct the source code in order to solve all faults. Then the ticket will be returned to the tester (in state *Fixed*). Usually, a dialog between tester and developer must be conducted to clarify the bug reports if needed.
- When the tester receives a ticket in state *Fixed* he must check whether all issues that were notified by him have been corrected. If so, he will forward the ticket to the instructor in state *Pass*. If not, the ticket will be returned to the developer.
- The instructor receives all tickets for all assignments (state *Pass*) once all testing and maintenance have been done. Then he evaluates the work carried out by the tester (for grading) and then executes a set of automated tests whose goal is to detect the injected faults. If some of these faults remain in the application, then the ticket is returned to the tester (state *Assigned*) including a short clue about the faults that have not been detected and the testing cycle begins again. If there are no faults the ticket is closed.

The instructor has visibility over all assignments (helpdesk and source code) and then he can participate in all discussions and take some decisions (behaviours classified as faults that are not, clarify the specifications, etc.). Part of the student's work is performed in the laboratory with the instructor support and part of the work can be performed by the students at home (all resources are available on the Internet).

3. Discussion

The first perception of the students is highly motivating, because it is a completely different task compared with other practices taken in other subjects. However when they begin the testing of the first components (business objects), they are sometimes frustrated. Some of the causes are:

- Specifications do not tell everything: The documentation received by the students is composed by the use cases, the database, and a few comments in each method to be tested. Considering each of these in isolation, it is not possible to get the whole picture of the specification. Even when carrying out a unit test of a single method, all the above information must be read, understood and synthesized in order to be able to detect failures. This task is difficult, but a very important part of the learning process.
- Specifications are ambiguous or apparently inconsistent: The database documentation is precisely defined, but, however, the use cases and method comments use natural language and sometimes do not refer exactly to the attribute and table names of the database as in his documentation. An additional effort must be carried out in the interpretation of the specification and removing ambiguities.

Again, most of the potential ambiguities can be resolved by considering the context of the whole application.

- No failures are found: After writing some tests in JUnit, it is common to not find any bug. However, at least the injected faults are present, as everybody is aware, which is very disappointing. The main reason is a poor understanding of the specification and/or a poor test case design.
- A reported bug is not understandable: This is a problem of communication. Students must develop the ability to communicate effectively and failures must be precisely reported in the helpdesk. If not, the developer can do this work.

At the end of the first testing cycle, less than half of the injected faults have been detected. For the second cycle, the student receives a clue about the remaining faults, which is helpful both for detecting the fault and for reasoning about how he must have designed the test cases to detect this kind of bug.

This laboratory exercise has been conducted during the last two years. We have found that at the end the experience is useful and pleasant for the students. However, there are some additional problematic issues that may be subject to a discussion on how to effectively teach testing in order to avoid them, in particular:

- Irrelevant test cases: The application is designed to avoid the necessity of testing the validation of user interface and database fields. Then students must focus on the behaviour of the application and on the database states and changes. Many tests may be irrelevant as sometimes they check only a small part of the functionality.
- More white-box than black-box (testing of business processes): As the source code is available, on some occasions the test cases are designed only to cover the code, forgetting key issues about the specified behaviour, leading to irrelevant test cases.
- Difficulties to automate functional tests (testing of user interface): When the assignment consists of testing the user interface, the overhead imposed by the automation of the test cases often hinders the task of designing good test cases.
- Poorly documented and difficult to maintain test cases: On many occasions the effect of automating test cases leads to many tests that perform very small database loads using a large amount of source code (cut & paste).
- Communication problems: Problem reports do not always include enough information to describe in detail the problem detected, so the developer is unable to correct the fault and more overhead in discussions is needed.

Acknowledgements

This work was partially funded by the Department of Education and Science (Spain) and ERDF funds under the National Program for Research, Development and Innovation, projects IN2TEST (TIN2004-06689-C03-02), Test4SOA (TIN2007-67843-C06-01) and the RePRIS Software Testing Network (TIN2005-24792-E, TIN2007-30391-E).