FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Test Automation in Enterprise Application Development

Ana Clara Fernandes Castro

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Carlos Pascoal de Faria (PhD)

June 17, 2013

Test Automation in Enterprise Application Development

Ana Clara Fernandes Castro

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Raul Fernando de Almeida Moreira Vidal (PhD)
External Examiner: José Francisco Creissac Freitas Campos (PhD)
Supervisor: João Carlos Pascoal de Faria (PhD)

June 17, 2013

Abstract

The subject of software testing has an increasingly pressing importance for software companies. In spite of all the progress in this area, it still faces a lot of challenges, which impacts the success of the testing processes.

Enterprises face the challenge of being capable of implementing a fruitful testing process, providing reliable results without consuming a significant part of all the development process. Despite being a rather unperceived activity, at least comparing with the development of new features, the testing process has a great influence on the performance of enterprises. Nowadays, enterprises more than ever, face the battle of increasing competition and to measure up to the clients expectations. All of this with the pressure of reducing costs.

It was having to deal with these challenges that Sysnovare, Innovative Solutions SA launched an initiative to improve its testing process. This was the motivation for this project: improve and automate the testing process on the context of its application development process.

Considering the architecture of the applications developed by the company, a new testing process and infrastructure were defined, focusing in two types of tests: testing the web user interface layer and testing the business logic layer, implemented as Oracle PL/SQL functions and procedures.

For automating the testing of user interfaces, we used Selenium as the basis for our work. It is a tool to test user interfaces, based on the actions capture and replay concept. To adapt the tool to the company's needs it was necessary to add improvements for it.

For automating the business logic tests, we worked with the FitNesse framework, using the DbFit fixture. This tool required several extensions to the features it already had, in what concerns the Oracle support of the project.

The new testing process was enriched with the definition of activities, rules, participants and the integration of the testing tools, which contributed to the automate concept.

These three contributions resulted on an enriched testing process and an enhancement of testing tools which support the automation of the testing process. This work definitely had a positive impact on the company's processes with proven results on this area.

Resumo

Testes de *software* é um tema bastante marcante e com considerável valor para as empresas de desenvolvimento de *software*. Esta é uma área que ao longo dos últimos anos tem sofrido uma considerável evolução; tem vindo a ser uma grande aposta em temas de investigação, no entanto, ainda apresenta bastante desafios que condicionam o sucesso dos processos de teste de *software*. Cada vez mais, as empresas são exigentes com os resultados proporcionados por este processo de testes. Num ambiente empresarial competitivo e sem fronteiras, apesar de os resultados de uma actividade de teste serem menos perceptíveis, não são seguramente menos fundamentais para aumentar o desempenho de uma empresa. Talvez por ser uma actividade menos perceptivel, é-lhe requerida que produza resultados confiáveis e rigorosos mas sem incorrer em elevados custos financeiros, humanos e temporais.

Estando alerta para o impacto dos testes de *software* e estando a enfrentar os desafios que esta área apresenta, a Sysnovare, Innovative Solutions SA lançou-nos o desafio de melhorar o seu processo de teste de *software*. E foi assim, que surgiu a motivação para a realização deste projeto: integrar no seu processo de desenvolvimento de *software*, um processo de teste melhor e mais autónomo. Assim, foram definidos um processo de teste e ferramentas para o auxiliarem, tendo em conta dois níveis de teste: testes ao nível da *User Interface* e ao nível da lógica de negócio, que neste caso específico é implementada utilizando a tecnologia Oracle PL/SQL.

No que concerne automatizar os testes ao nível da *User Interface*, recorreu-se à utilização de uma ferramenta de testes recente, o *Selenium*. Esta é uma ferramenta que proporciona o teste de interfaces gráficas recorrendo ao conceito de *capture and replay*. Com o objetivo de tirar o máximo partido deste conceito, foi necessário integrar funcionalidades nesta ferramenta, por forma a adaptar-se aos requisitos das interfaces da Synovare.

Ao nível da automatização dos testes à lógica de negócio, trabalhou-se com a *FitNesse Framework*, recorrendo ao componente do *DbFit*, responsável por estabelecer a ligação com as componentes de base de dados. O trabalho ao nível desta ferramenta, teve como ênfase a parte do projecto que oferece suporte às base de dados Oracle.

Finalmente, foi possível construir um processo de teste, com atividades, participantes e regras definidas. Fundamentalmente, houve a integração das ferramentas de teste que conferem mais automatismo ao processo de teste.

Estas três vertentes do trabalho resultaram de uma resposta ao desafio proposto pela Sysnovare, Innovative Solutions SA e contribuíram positivamente para a evolução da componente de testes integrada no processo de desenvolvimento da Sysnovare, Innovative Solutions SA.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

API	Application Programming Interface
BDD	Behaviour Driven Development
ERP	Enterprise Resource Planning
FIT	Framework for Integrated Tests
GUI	Graphical User Interface
HTML	HyperText Markup Language
IDE	Integrated Development Environment
JDBC	Java Database Connectivity
MVC	Model-View-Controller
PB	Product Backlog
PL/SQL	Procedural Language/Structured Query Language
TDD	Test Driven Development
UI	User Interface
XP	Extreme Programming

Chapter 1

Introduction

1.1 Context

Software testing is a very important component of the software development process. It is the area, among other things, responsible for evaluating the accuracy and effectiveness of the produced software. This is a subject which raises a lot of discussion, due to every different possible techniques, methods, interpretations and concepts. The evidence which underpins this statement is the vast number of existing articles around this topic. It is hard to find an article discussing this subject which does not underline the importance of the testing process. It is such a fundamental activity that nowadays it is seen as "an activity that should encompass the whole development process" [IEE04].

It is from realising the importance of the testing process that Sysnovare, Innovative Solutions SA, from now on referenced as Sysnovare, has begun to invest on this area. Achieving the excellence on a testing process is a dream difficult to accomplish. Nonetheless, it is possible to outline a clear definition of the objectives for this investment and to come up with a clear plan for its implementation.

The goal of this project is twofold: establish rules for the testing process and develop an associated infrastructure for test automation for enterprises applications that can be applied in the specific context of Sysnovare. The definition of the testing process is a sort of a case study since it has a specific implementation at the Sysnovare context. In what concerns the automation of the process, it is a work which has a generic approach. It includes investigating and improving the existing testing tools. This work aims at contributing to the automation concept, what can be useful for other contexts besides Sysnovare.

Sysnovare is the enterprise where this project is going to be developed. It is a small to medium enterprise with a large variety of products: enterprise resource planning (ERP), document management and academic management. All these products have a significant community of everyday users. Sysnovare products are mainly built for a web context which implies a requirement for

this project: the tests have to reach both the business logic and the user interface layers of their products.

1.2 Problem

Software testing is an activity performed with the aim of evaluating and enhancing product quality. It consists on verifying, countless times, the product's behaviour to ensure it is according to the expectations, performing the uncountable operations which the product supports and performing them on many possible different ways. It is this group of activities that stand the affirmation which claims that a product is bug free. However, it is impossible to say, with a hundred percent certainty, that a product has no defects. It is even harder when the laborious activities which corroborate this statement are only performed manually. Unfortunately, by manually testing it is very difficult to assure that an application has no defects. Since, this activity, when done manually, is very prone to failures.

This is the scenario at Sysnovare: a company which tests its products manually. Woefully, this reality does not lead to bug free products. In spite of testing the products before they are released to the client, it is common to realise that the product is not behaving as it was expected. For an enterprise this is the worst that can happen. On the one hand, the person who performs this activity tends to become disappointed with the results of his tiring task. On the other hand, the clients become less confident on the products they are using.

Therefore, Sysnovare launched this challenge: improve and automate its testing process in the context of its applications' development process. They aim at achieving a reliable and effective way to test their applications. They want to go beyond a very time consuming task with unreliable results, to a less time consuming, yet more reliable, testing activity.

To conquer this goal it is mandatory to introduce the automation concept on the testing process. This also includes exploiting the use of tools to help on this activity. Moreover, redefining the testing process also includes defining new participants, with new roles for this process. As far as possible, it is necessary that the participants of the testing process are as much independent as possible. This should be an activity executed not only by the developers, but also by the analysts or even better, by the end user of the products. Offering the client a way to evaluate the accuracy of the products avoids the aforementioned drawback of having clients who do not trust the product they are using. It also contributes for a positive appraisal to the product, which is a very important factor on nowadays competitive market.

1.3 Objectives and Contributions

The central objective of this work is to systematize and partially automate the testing process at Sysnovare, with the aim of improving its effectiveness, efficiency and accessibility. The effectiveness is conquered by having a process which enables more reliable results, the efficiency is gained

if you spend less time testing and by accessibility we mean the ability of non-technical people to take part of testing activities. To achieve this it was necessary to conquer smaller objectives.

The first objective is to add an effective testing process to the existing development process. This requires the definition of the activities that should represent the testing process. It is also necessary to define who performs the activities and how. Moreover, there is the definition of rules to be applied to the different contexts where the tests take place. Some of these rules also determine the progressing of the testing workflow.

To be able to implement a successful testing process, it is mandatory to have the right tools to do so. Therefore, the second objective of this project is on exploring the available tools that allow the definition of tests. We want to test the project's logic without a significant programming effort. After knowing what is already available on the software testing world to help on this activity, we needed to adapt it to the Sysnovare requirements. In this case, these requirements are related with the technologies Sysnovare is using.

In a context of web applications it is not enough to test an application at the bottom level - business logic. In fact, the user interface is what has more impact for the end user. Normally, it is through the application interface that the user decides for accepting the product or not. It is the way that a non-technical user has to evaluate the behaviour of a product. Therefore, part of our work is also at the interface level. Once again, we check what is already available and how we have to adapt it to our needs.

Hand to hand with these last two objectives, there is the aim of also working to achieve two concepts already mentioned: the automatism of the testing process and the independence of the users who perform the tests.

Therefore, in our search for tools to integrate on the testing process, we looked for something which allows replaying a test, without a lot of set up activities.

In what concerns the independence of the users who take part in the testing process, we looked for tools which have the capacity of avoiding programming effort. In case they do not offer it, we worked to adapt them, so it would be possible for non-technical people to test the applications.

Having a well defined approach to the testing process activities and a pair of tools helping on this activity, enabled reaching a considerable variety of tests which can be run automatically. With this automatism, less effort is required to both developers and analysts, every time a new product version is released and has to be tested.

1.4 Document Structure

This document is structured as follows.

The first chapter introduces the objectives for this project. It describes which are the motivations that guided this project and in which context it was developed.

Moving on to the next chapter, chapter two, it is concerned with contextualizing the environment where this project was developed. It gives an overview of the Sysnovare development

process, which is essential to later understand the decisions that were made. It also addresses, in more detail, the challenges which were the motivators of this project.

Chapter three provides a review on the existing literature in what concerns different approaches of software testing. It continues with the analyse of two testing tools: FitNesse and Selenium. These are the chosen tools to use on the tests at the Application Programming Interface (API) and Graphical User Interface (GUI) levels. Therefore, it is important to see what they already offer.

The following chapters describe the work developed.

In chapter four we present a suggestion for the testing process. We present the activities that should take place on each phase, who should perform them, how and using which tools. In the meanwhile, a collection of rules is also proposed.

Chapter five is about the work developed at the API testing level. We describe the work implemented on top of FitNesse and which are the achieved results for each implementation. We also say how FitNesse was integrated at Sysnovare.

The work developed at the GUI testing level is described in chapter six. Similarly to the API level, we describe the functionalities we added to Selenium and the benefits they brought to Sysnovare GUI tests.

Finally, chapter seven gives an overview of all the work described throughout the document, the results of this project and the remaining work to achieve the excellence on testing Sysnovare products.

Chapter 2

Problem Analysis

The present project emerged from a group of identified issues that are affecting the effectiveness of developing and maintaining software at Sysnovare. So being, we think it is important to focus on a few details that can give us the big picture on how Sysnovare works.

So, in this second chapter, we take a look at how the development process is organized: what is the development process, how the work is divided, scheduled and distributed among the available resources. We highlight how the testing process is performed, since this is a theme that is going to hold extensive discussion throughout this project.

Moreover, we believe that part of presenting the working practices at Sysnovare, includes describing the architecture of its projects. Thus, in this chapter we include a section where we outline the nature of the company's software projects.

This chapter concludes with the identification of the challenges which are going to guide this project. These challenges focus on the Testing area due to its impact on the quality of Sysnovare products and the satisfaction of its clients.

2.1 The Software Development Process

"One of the major innovations in software development methodology of the last few years has been the introduction of agile principles" [VBCJ09]. It is a trend in today's enterprises to use agile methods. Sysnovare is an example of this tendency. The Agile Methodology fulfils the needs of the actual scenarios of developing and maintaining software products. Agile methodologies are characterized by their easy adaptability to a constantly changing environment, where the clients' needs change, the time and financial pressure are crucial and the existence of a global competition, makes that Sysnovare must be at the same productivity level of industry-leaders.

Regarding all the different types of agile methods, the one which best represents the way the work is done at Sysnovare is the Scrum method, which has been adapted to the company's own needs and aims.

The Scrum method was proposed in 1995 by Ken Schwaber [VBCJ09], when it became clear, to the majority of people working in big projects, that the development of software was not something that could be planned, estimated and completed successfully using the traditional methods [VBCJ09]. It is on this main concept that Scrum relies on: many unpredictable factors can have impact on a project during its development.

2.1.1 The Team

At Sysnovare the development process is based on the Scrum method but with some adaptations to the company's objectives and needs: SysScrum. Adapting Scrum was necessary due to the different levels of maturity of the projects owned by Sysnovare. The company owns a set of products that have been under development and evolution for a long period, without following a defined methodology. These projects, nowadays, require a lot of maintenance, making it difficult to be integrated in the SysScrum method. However, to the new developments and to the new products an effort to apply the SysScrum is being done.

The SysScrum team is small, formed by three to four elements, each element having a very specific role on the Scrum implementation. First, we have the analyst who is responsible for contacting with the client and discussing requirements, having a deep understanding of what the client is expecting, as well as defining and prioritizing the features.

A second actor in this process is the Scrum Master, who, in this case, is also responsible for the project as the project manager. Having the same person carrying out these two roles can cause some controversy in view of what is defined for each role of a Scrum team member. However, Sysnovare is a small company with no more human resources to delegate these functions. The Scrum master decides which tasks are carried through in each version of the product and assigns them to the developers. But most important, he is the team member who has the best knowledge of the product and its architecture, which is of major importance when planning products' versions.

Finally, there are the team members, normally one or two, who are responsible for implementing the new features for the latest version. Usually, the developers belong to one or two project teams which results in developers being assigned tasks of different projects. Even though this 'multi project' discipline is not always possible to avoid, an effort is being made to reduce the constant changes of the project teams. Sysnovare is aware of the benefits of having stability on its development teams. The increase of productivity strongly relies on the stability of the teams, thus leading to its developers to be more knowledgeable about the product business.

2.1.2 The Process Lifecycle

Having described who takes part in the development process, we carry on looking in detail at how the development process is organized at Sysnovare. This textual representation is illustrated by the diagram in figure 2.1.

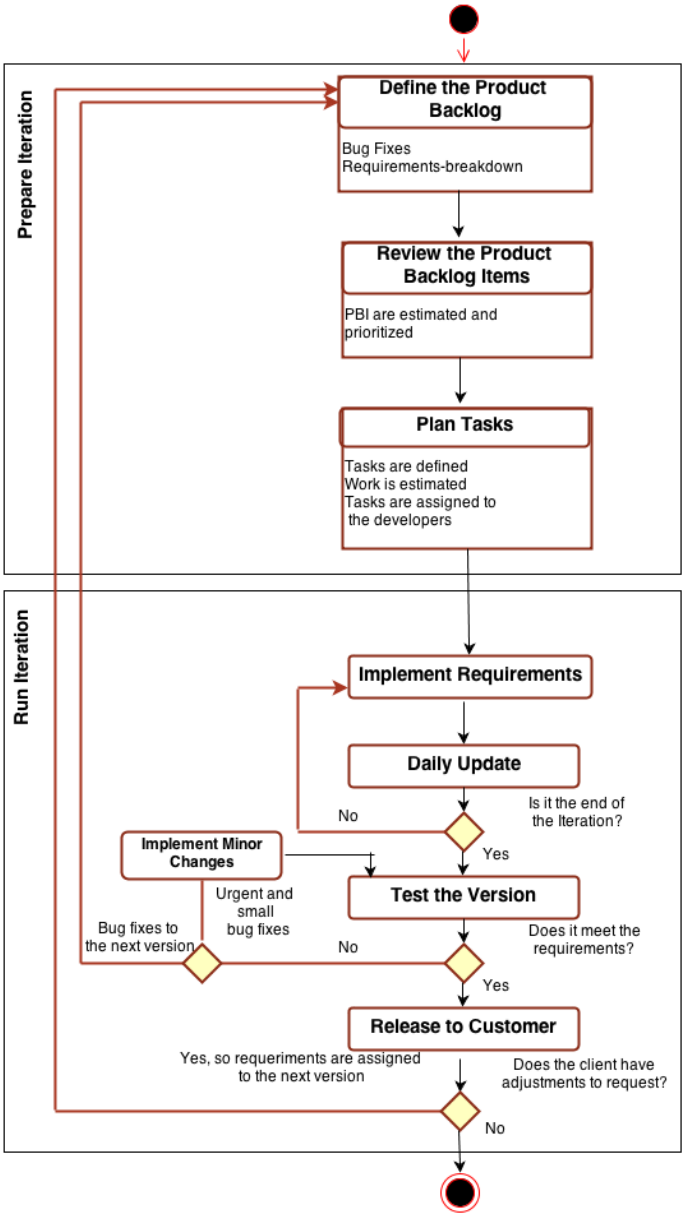

Figure 2.1: Release development process at Sysnovare

Each SysScrum cycle, designated by *Iteration*, starts with the gathering of the client's needs by the element responsible to interact with the client - the analyst. Normally, this person organizes the client's requirements according to his priorities.

After this first step, a meeting between the analyst and the project manager is set. It is during this meeting that the projects' versions are decided: their due dates and the items of the Product Backlog to be include on them. Therefore, prior to this meeting there is a previous work which consists on dividing the high-level concepts in smaller requirements, what is normally done by the project manager, due to his knowledge of the product's architecture. At this point, it is time to prepare the Product Backlog.

For each product's version, a Product Backlog (PB) is built. It contains, not only the requirements-breakdown, but also bug fixes, which may have been generated on the previous released product's versions. The items included on the PB are, once again, prioritized depending on factors such as business value, customer importance and available resources. At the end of this second step, the PB items are transformed into new tasks. New tasks are defined, the estimation of work to be done for each task is set, and expressed in hours, and each task is included in versions of the product to be released. So, the next stage is to assign the tasks to a developer and prepare the Scrum board.

Each project has a Scrum board where the tasks are placed. This board consists of three columns: *ToDo, Working on It* and *Done*. At the beginning of the *Iteration*, every task is on the *ToDo* column and then, according to the work of the developers, the task is placed in the other two columns. A task is completed when it is placed on the *Done* column, meaning it has already been implemented and basic tests have been run. Consequently, they can now take part of the next released version. It is possible to realise that when a task is on the *Working on It* state, it means that the feature is being developed or that it is being tested. Since this Project is concerned with the Testing subject we consider it essential to explain in detail how the testing process works.

2.1.3 The Testing Process

The quality of the delivered software is highly affected by the effectiveness of the testing process. Therefore, the testing process should have the least possible pitfalls. At Sysnovare the testing process is all done manually, which means that it is very prone to failure.

Another point which does not positively contribute to the effectiveness of the testing process is the lack of documentation: not all the features of a product have formally defined test cases. Mainly, the most uncommon actions are left out of the tests.

However, Sysnovare is moving on to a better testing process. There are some products which already include a defined collection of test cases. This job required an investment of time to carefully think on the test cases, discuss and improve them. These test cases were built specially to test the user interface. They define the inputs the system is capable of handling and which is the expected behaviour of the application.

To look in some detail at how the testing process is organized, it is important to mention that not all the releases of a product are tested the same way. There are two types of releases:

Version: Testing a version includes testing all the features of a product, from the basic ones to the more complex. The features are described in a list and each one is associated with a group of test cases which are manually covered every time a version is released. After manually tested, a feature is set as having passed or not the test.

Patch: In case we are testing a patch, not all the product's features are tested but only the features related to the items included in the patch. A batch of test cases is also built and once again, the tester, who also plays the role of a consultant, manually checks all the test cases and identifies the feature as being correctly implemented or not.

2.1.4 The Release

The release is the last step of the development process. When an *Iteration* time is finished, it is the moment to release the product's version associated with the *Iteration*. At this point, if some of the planned requirements are not finished, they are post-poned to the following version. Similarly, if any of the implemented requirements do not succeed in passing the testing phase, it is also assigned to the following planned version.

With the requirements which were developed and well-succeeded on the preliminary tests, a new version of the product is released. In the first place, the version is installed on a demo environment, where, once again, the software is tested by the product's analyst. Any bugs identified are added to the PB or immediately fixed, depending on the complexity and urgency of the issue.

Finally, when the version is ready to be released to the customer, it is installed in the production environment. From here on, any bug detected is communicated by the client to the company's help-desk. It is analysed by the analyst and then added to the PB.

2.1.5 The Project Management Tool

Through this first section of the second chapter, a lot has been said about organizing PBs and products' versions. Now we will focus on how this is done. Therefore, during this subsection we will discuss how the *Gestor de Projectos* tool works, since it is an important tool on all the management of the Sysnovare products.

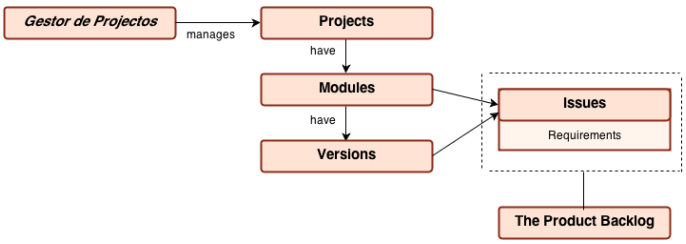

Figure 2.2: Main concepts managed by the tool *Gestor de Projectos*

Gestor de Projectos (GP) is the name of the tool which was developed with the aim of controlling the developments made in a project. It keeps a history of the changes made on the project's code, when they were made, by whom and why.

In the field of managing projects, GP offers features which allow the project's responsible to manage the project's versions. Each product is divided in modules, as it is explained in the following section, and for each module there is a group of versions. These versions are the ones which are decided during the planning of a SysScrum *Iteration*.

As it is possible to see in figure 2.2, each version has to include a group of tasks to be developed. These developments are specified by *Issues*. An *Issue* is an item of the PB. It is by creating new *Issues* that a PB is built. Then, to create a new version it is necessary to choose from the existing *Issues* the ones which will take part of the version.

Using this method it is easier to keep the work on track: what was done, what is being done and what still has to be planned. This also enables following the versions status and specifications: it is possible to know which versions were launched, with which features and when.

2.2 Product's Architecture

In a context where web applications have faced a tremendous growth, it is normal to notice a significant development in this area, particularly in subjects related to the construction and maintenance web applications. On that account and "because the Model-View-Controller design paradigm's main purpose is to separate business logic from presentation" [Vla03], this Model-View-Controller (MVC) architecture provided a refreshing look at how web applications should be thought and organized.

The idea of having applications divided in three logical parts came with the MVC architecture: the model, the view and the controller parts.

The model "is the database interface" [Fla04] as well as a connection point to the business rules.

The view is responsible for getting and showing information. It "provides the front-end user interface to the application" [Fla04].

Finally, the controller "sits between the model and the view" [Fla04]. It "takes instructions from the user, interacting with the view, executing them through calls to the model, then taking the results" [Fla04] and passing them to be displayed by the view.

Therefore, in what concerns the architecture of the applications developed at Sysnovare, it aims at following this MVC concept.

2.2.1 Product's Structure

Sysnovare has a clear specification regarding the organization of its products, not only in what concerns the architecture of the products, but also on how to divide the different components of the products.

Staring with the organization of the different product components, it is important to mention that all the applications are divided in modules: application's units which encapsulate related concepts and that can be individually developed an independently tested and deployed.

Normally, a module is required to have a minimal set of dependencies on other modules. But since dependencies exist they are handled by specific business rules built with the aim of establishing the connection between modules. Furthermore, it is important to mention that this modularity was chosen mainly because this way applications remain flexible, maintainable and stable even as features are added and removed [BCH+11].

To finalize, another point in favour of this modular approach is that it enables easier extension of the applications, which is always an important advantage when developing web applications that are growing in the market.

2.2.2 Packages' Organization

Sysnovare web applications are written in PL/SQL. These PL/SQL web applications consist on a number of procedures that allow the interaction between the back office, the database, and the browsers. This is one of the functionalities offered by PL/SQL packages: the ability to hide logic and data from view [Feu13]. Due to this, packages are considered a powerful and important element of the PL/SQL language. These elements not only help on a better organized code structure as they are also known as the most important construct for building reusable code [Feu96].

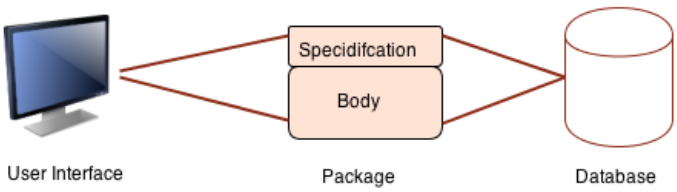

Figure 2.3: PL/SQL Package Interface

A package is an object which gathers functions and procedures. As figure 2.3 shows, a package always has a specification, "which defines the package items that can be referenced from outside the package" [Feu13]. The other component is the body, which implements, for example, procedures and functions.

As previously mentioned in this chapter, the company has a very well defined architecture for its applications. It is on the name of the packages where it is easy to identify the three MVC layers concept. Package names start with the module name followed by a small group of keywords. Knowing the meaning of these keywords, it is possible to identify to which layer of the MVC model the package belongs to, as it is illustrated in figure 2.4.

Starting with the Controller layer: the packages ending on BR, BRA, BRM and BRMA have functions and procedures related to the business rules and they are mainly an interface to the packages which interact directly with the database.

The packages which directly interact with the database are those whose name ends with DM and DMA. The functions written in these packages perform select, update, insert or delete operations. They directly change the model or they directly retrieve information from the model. So, these packages are placed at the model layer.

Lastly, the view layer, which contains the packages whose name ends with the keywords MAIN, WEB, AJAX and JS. These packages provide the front-end user interface to the application. The user accesses the application calling procedures from the MAIN or AJAX packages where calls to the WEB or JS packages are made. It is on the WEB packages where html elements, such as tables or forms, are drawn. The JS packages' name stands for Javascript and it is where the javascript functions are implemented. This group of the user interface packages are highly dependent on the Controller packages, since they interact with the database by calling the functions and procedures described on the business rules packages.

It is important to mention that normally the procedures and functions implemented on the Model and Controller layers are shared by more than one procedure. This means that modifying a business rule can affect the behaviour of more then one procedure, which can be seen both as an advantage or disadvantage.

The advantage of reusing the same procedure relies on the fact that when a bug exists it is easier to correct just a procedure. However, it can happen that once we altered a procedure, it can have a negative side effect on other places where it is used. Drawbacks which can easily not take part of this list if we have a feasible testing method.

2.3 Testing Challenges

After the global overview on how Sysnovare works and develops its applications, in this third section we will analyse what can be improved concerning the testing area.

As it is possible to conclude after reading section 2.1.3, the testing process has no strict rules that should be followed in all applications, every time a version has to be tested. However, it is a subject of most importance in what concerns the quality of the software produced in any enterprise and the confidence it creates in any customer. Being very aware of how challenging this area can

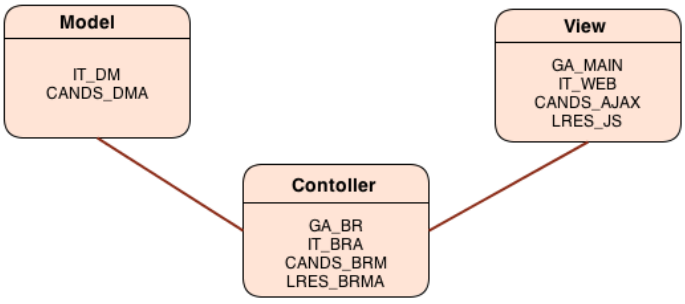

Figure 2.4: MVC naming conventions (package suffices)

be, since there are many perspectives to take into account, Sysnovare knows that its testing area requires a deep discussion and analysis.

2.3.1 The Tests' Specification

We can start by pointing out, as it is possible to see in figure 2.1, that the development process does not formally include a phase to specify the tests. A phase to carefully analyse and discuss the requirements is essential for short and long term benefits. It impacts not only the testing process but also the development process.

On a short term, there are several advantages. First, it is guaranteed that the requirements are deeply analysed, resulting in a fully understanding of what has to be developed.

Secondly, it is important for the developer, to know exactly what is expected from the requirements he is going to implement. Having a detailed specification and examples of the expected results, will have a positive impact on the development process.

The last short term advantage is related to the *Test the Version* step (on figure 2.1). When we have specifications of the test cases, it is easier to assess whether a version meets the requirements or not.

Considering a long term perspective, completely defined test cases enable accurate tests of the global applications. This guarantees that if a requirement passed on the original tests and the specification did not change, then the requirement continues meeting the client's needs as it continues passing the first defined test cases.

2.3.2 The Testing Actors

Another challenge of this testing subject is having different participants defining and accepting the test cases. Having more than one person on this activity can bring different perspectives to the problems. Currently, it is the consultant who performs the basic tests. But if we have another actor - the client, for example - , we will be more confident on the effectiveness of the test cases.

Moreover, the client will be involved in the testing process, which will result on a client more aware of all the details related to its application. It can also result on a client who recognizes the complexity and dimension of the application. This can contribute to an increase of the product value.

It will be important to have the consultant and the client working together on defining the test cases. When the tests are approved by the client, it is more likely that if a requirement passes the testing phase, no adjustments will be requested by the client.

This raises an important point: how can the test cases be built to enable a person with no technical expertise to take part in this activity?

2.3.3 Test Management

It is also worth having a second look at how the existing test cases are currently managed. For each new version of a product, a worksheet is created with all the application requirements and the

corresponding test cases. The test cases are specified by a group of steps and values that should be followed to achieve a defined result. Depending on the result obtained, the test is marked as passed or not, on the corresponding columns.

There is room for improvement on this approach of managing the test cases, since a list of requirements and test cases easily becomes extensive and difficult to manage.

2.3.4 Test Automation

Testing the whole application is without doubt a top challenge. It is hard to cover all the possible test combinations, specially when manually testing. And even if the tests cover a big part of the application features, are the tests' results reliable? Testing large applications can be very time-consuming, exhausting and tiring, which can lead to accidental mistakes.

Therefore, it is important to consider moving from the manual tests to an automatic process. The automation, even if not complete, can, above all, enable more reliable results. Furthermore, an automatic process can be a solution for one of the biggest testing constraints: time.

In addition, there is a key point when testing web applications which is how to have quality data to run the tests. It is important to be able to deal with this data management difficulty. An automatic testing process, can again be the answer to solve this major challenge.

2.4 Conclusion

This chapter arises with the idea that to understand the problems studied during this work, it is fundamental to know the context from where they have emerged.

As it is possible to observe, Sysnovare perfectly fits in the characteristics of the majority of the small to medium enterprises developing web applications nowadays. It has to deliver good quality software, with short development time and with minimal human resources requirement. Furthermore, as a young company, it is trying to improve its methods. It started by the development process: adopting an agile methodology. It decided to shift the focus to the testing work. Testing software is not a trivial task. It can reach a high level of complexity but, on the other hand, the benefit is an improvement on the company's results.

However, to achieve this improvement, the testing process is required to be effective, thorough, the least time-consuming and, most important, to demonstrate reliable results. The answer to achieve a testing process with these characteristics shuns manual testing. In fact, this solution is, above all, very prone to failure.

An automated testing process can deal with some of the issues raised with the manual testing process. No doubt that this approach requires a deep study on the subject and a long period of experimentation and analysis of a future solution. Automated testing is a theme that has been raising a great deal of discussion and no perfect solution has been achieved yet. That is why a lot of experimentation is required. This way it is going to be possible to have more results to analyse and to contribute to the enhancement of an automatic testing process. This can result in more reliable applications, receiving more credit from their users.

Chapter 3

State of the Art

The scope of this project is the definition and implementation of a testing process at Sysnovare. The project focuses on three main issues: the definition of a testing process, the automation of tests and the possibility of having non-technical people defining tests at different levels, API and UI.

To develop this work, it is fundamental to first study what has already been done in this area. Therefore, during this third chapter, we are going to review the relevant literature and analyse tools which can be useful to achieve this project's goals.

3.1 Testing Methodologies and Approaches

As it was stated on chapter 2, one of the topics addressed in this project is the testing stage of the software development process. Since we aim at including the testing step on the current implementation of the agile methodology mentioned on section 2.1, during this section we are going to discuss different ways of thinking the tests, in what concerns testing methodologies.

Therefore, we focused our research on agile software development approaches where the testing component plays an extremely important role. Ideally, we want the tests to guide the code implementation.

In the next three sections we are going to closely analyse an agile software development process, the Extreme Programming, and two testing approaches, the Test Driven Development and the Behaviour Driven Development.

3.1.1 Extreme Programming

Extreme Programming (XP) is a software development process which emerged from the agile methodologies concept. The XP approach is based on four vital foundations [LJ04], from which we emphasize two: communication and feedback.

Communication, at any level, is always a key factor when working as a team. With this scenario, it is important to be aware of how powerful communication can be. It is important to assure that this activity is part of every day life of all the team members, guaranteeing that all the team members share the same ideas, are working with the same knowledge of the requirements and are working to achieve the same objectives.

At this software development process, it is not only the communication between the team members that matters, but also the communication with the client. This communication has proved to be a key factor for the success of the implementation of the XP method [LJ04]. First of all, it enables a more accurate understanding of what the client is expecting. But, most important, it is the influence that the communication has on the other foundation of this development process, already stated, the feedback.

The feedback is the core element that guides the development process: XP is obsessed with feedback [LJ04]. The feedback received by the client is a key element. On the XP method, this feedback is, mainly, retrieved by the acceptance tests defined by the client. It is the definition of concrete criteria to assess a software which can bring forth its quality. This may also result in a satisfied client who knows that he can rely on the software he is using.

Now, we are going to bring focus on the organization of the XP software development process.

3.1.1.1 The Team

On the XP method it is crucial that all the elements who form a project team are always focused on the same goals. Normally, a XP team is composed of five elements [LJ04] as it is possible to see in figure 3.1.

Figure 3.1: Roles in a typical XP team

Starting with the Client element, this is someone who is not the actual Client, but definitely knows the client's needs and knows in detail the context where the software is going to be used. Therefore, this element is capable of setting the requirements and also the matching acceptance criteria. He is also the one who knows what can bring value to the business [LJ04]. Then, the Analyst assists the Client in defining the requirements.

The Tester, as a role is responsible for implementing the tests. But, prior to this activity, it is the tester who helps the Client defining his acceptance tests [LJ04]. Due to the Tester know-how on how to approach features and design their acceptance tests, this aid can be very helpful for the Client. Then, it is based on the defined acceptance tests that the Tester builds the unit tests. For

unit tests, the same team member assumes the tester and developer roles. This is a topic which we are going to address in more detail during this section.

Of course, the XP team includes Developers to produce the code. It is interesting to observe that with this agile methodology, the Developers are organized in an uncommon way, following the saying: "two heads really are better than one" [LJ04]. The objective of implementing this pair programming practice is that all the produced code is reviewed by more than one person [LJ04]. For this to work, every Developer develops software following a defined code style, so that everyone can read and improve someone's code. Normally, this results in better coding.

Finally, there is the Project Manager element. This person is not only responsible for managing the human resources but he is also responsible for managing the project leading to the accomplishment of the objectives set to it [LJ04]. The Project Manager allocates developers to the teams, according to their know-how on the needed technologies to develop the proposed features. It is fundamental that the Project Manager can guide his team to achieve the business goals, despite all the obstacles that might appear.

3.1.1.2 The Planning and the Development Process

This subsection is about how the software is built on the XP method. Here we describe the practices of the XP method that enable delivering software with very low defect rates, making this approach more attractive [LJ04].

The main idea around XP practices seems to be the simplicity. When talking about how the iterations are planned the simplicity rule is also applied. There are two main moments of the planning process: the release planning and the iteration planning.

- **Release Planning**

  The first moment is the release planning. Here the final decisions are on the Client element of the team, mentioned on the previous 3.1.1.1 section. This is due to the type of the decisions taken at this step: what to do next and what to do in a near future [LJ04]. The objective is gathering all the requirements and give them a value of importance. However, as it is frequently discussed on the development software world, it is difficult to predict all the needs of the client. Therefore, the planning decided at this point is not static. It is a dynamic process, which is constantly being reviewed and adjusted to the emergent necessities of the client.

  Having the ensemble of requirements the Developers, superficially, analyse them and estimate their cost, in terms of time. According to this estimation, it is the Client who decides which features he wants for the next release and the ones which will come on the following releases. This way, the progress of a software is in accordance with the client's expectations. And most important, this progress can be a hundred percent noticed by the client. Here is where the second moment of the planning process comes in.

- **The Iteration Planning**

A very unique characteristic of the XP method is its very small series of fully integrated releases [LJ04]. The planning of iterations takes into account that normally a iteration is of a two weeks period. And at the end of each iteration the team has to release running, tested software, delivering the features chosen by the Client.This released version has to be completely workable, not only for the client's purpose of testing it but also for working with it [LJ04].

When planning the iterations, the cost of the features is once again estimated, but this time in a grained level. The planning also takes into account that every produced code has to be refactored.

The refactoring of the code is a key step on the XP method. Therefore, there is no way of not taking it into account when planning an iteration. The main idea of the refactoring process is to keep the design simple [LJ04]. The code is reviewed, the duplications are removed and the cohesion is checked. To verify that this aim of improving the design does not change what was previously built and was working properly, there are the tests. They assure that what was working, still works after the improvement of the code design.

3.1.1.3 The Testing

The XP development software process is obsessed with tests [LJ04]. Tests have an important impact on one of the already mentioned foundations of the XP approach which is the feedback, since tests are considered a type of feedback. Formally defined tests and automated tests provide reliable information on which features are working properly [LJ04]. And most important, when referring to automated tests, this feedback is instantaneous [EMT05].

The XP method includes tests at two levels: acceptance tests and unit tests.

The acceptance tests are defined by the Client team element (3.1.1.1 section). The fact that these tests are defined by someone who knows in detail the environment where the software is going to be used, enables an important validation of the implemented features. Getting this feedback of how well the features are implemented allows the verification of the correctness of the features prior to installing them on the client. Thus, prior to to the client working with the software, it is important that the software passes all the tests defined by the client.

Adopting the XP practices includes continuously running the tests. This way, a rigorous quality control is made, since it is assured that once a test runs, the team keeps it running correctly thereafter [WMT12].

The Client acceptance tests are defined by the Client, but implemented by a developer. This developer has the role of Tester (3.1.1.1 section) and besides implementing the acceptance tests, the Tester also implements the unit tests. These are the tests which result from the philosophy of the XP: write tests first and then write the code [LJ04].

Writing tests first obliges people to make design decisions first [EMT05]. It is proven [MH02] that having a more concrete idea of what is going to be developed results in a quicker and assertive

development process. This concept of writing tests first indicates that the XP teams practise the test-driven-development (TDD).

3.1.2 Test Driven Development

Test Driven Development (TDD) is a model for developing software that emphasises the importance of testing. This model can be generically described as a model in which the computer programme is designed by writing first small tests [Itk12]. The definition of these tests is based on the requirements, since the tests are supposed to cover all the possible situations of a requirement. By having tests based on requirements and producing code to pass on each test, it is assured that the code being produced also completely meets the requirements.

However, on TDD the advantage of preparing the tests first is not only on having fully understood requirements. When the development process is based on the TDD approach, a developer has constant feedback about the code being produced. The tests exist and they are run as often as new code is produced which allows a rich feedback for the work being done.

Furthermore, with a well defined test suite, the quality of a project is always under control. The automatic tests run as often as the developer wants which enables the identification of early defects. Moreover, with these tests it is possible to check if old working functionalities were negatively affected by recent developments.

3.1.2.1 Test Driven Development Characteristics

The TDD consists of four phases:

- write the test;
- run the test and fail the test;
- develop the feature matching the test;
- refactor the code.

At a first glance, these are very simple steps and not difficult to follow. We are going to examine in more detail how TDD works, while following its dynamics in figure 3.2 [EMT05]. (i) As it can be seen, it starts by picking up a feature. (ii) For the chosen feature very small and simple tests are written. Normally, these first tests evaluate basic things like null values or edge cases. (iii) The test is run and, as it is expected, it fails. (iv) Therefore, to the written tests the matching code is added. (v) The aim is that this code passes on the previously defined tests. But it is also fundamental that the tests already defined for the project keep getting the green light (using the traffic light analogy). (vi) Finally, after getting the green light on all the tests, it is possible to refactor the implemented solution, improving the way it was developed, for example. As the production code passes on the tests, it is verified an increase on the complexity and robustness of the tests. New tests are defined validating other cases of the feature and this leads to the implementation of the matching code. It is possible to notice that there is an incremental

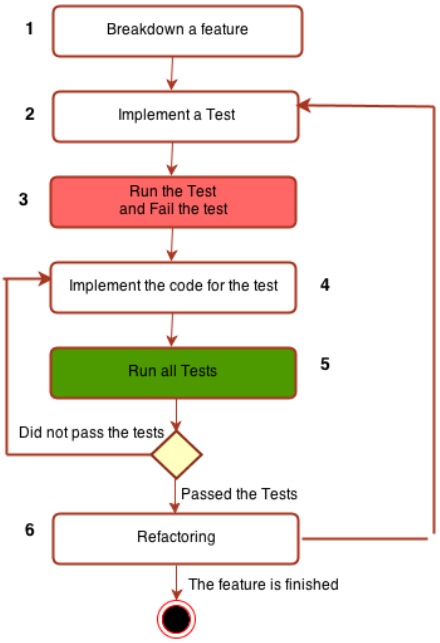

Figure 3.2: The TDD process steps

development until the feature is completed. This gradual growth of a feature is also a unique characteristic of the TDD process, once again related to the concept of first defining the tests.

This concept of development process seems to be easy to follow. However, there are development teams that face some challenges on implementing these five plain steps. The sceptics of a TDD adoption say that the adoption of this development process requires a lot of effort. But, more than this, they argue that this test first development slows down their projects [EMT05]. The truth is that TDD requires time to implement the tests. However, this can be seen as a gain of time in two ways:

- implementing the tests forces breaking down a feature and knowing it in detail. This can lead to a faster coding phase, since the developer is confident on what he is going to develop;
- implementing the tests decreases the bug density of projects [Sei09]. Normally, the test coverage is very satisfying. Therefore, the software will have few bugs left.

3.1.2.2 Test Driven Development Advantages

When talking about the advantages of using TDD there is an idea which emerges: TDD enhances both product quality and programmer productivity [EMT05]. This idea demonstrates how tightly integrated the testing and the programming activities are.

To have a positive contribution on the product's quality and to optimize the developer's work there are some characteristics of the TDD concept worth exploring.

In TDD, tests are written before implementation. This process requires the developer to breakdown the features and to completely understand the requirements. Therefore, it is said that this

approach allows a better task understanding since the developer has to express the functionalities without ambiguities [EMT05].

There is also the belief that this approach results on developers focusing more on every small detail of the features [EMT05]. Tests are made for each small detail of a feature, followed by the implementation of the code. Since this code matches a test case with a very limited scope, while developing the functionality for a single test, the developer is focused on a small piece of the whole feature.

Furthermore, this incremental development makes it easier to identify the reasons for not having a green light on all the tests. If a test fails after a short burst of testing and code production, the problem is easy to identify due to its limited context [EMT05]. This also enables detecting small problems before they get bigger and scale to situations which require significant rework effort.

3.1.2.3 Test Driven Development on this Project

This approach is all on the benefits of developing one test at a time and writing tests before implementation [EMT05]. If we aim at implementing a development process where the tests are the core of the development process, there is no doubt that the test-first approach is a good practice to start using. Probably, adopting TDD will require a period of adaptation, since right now at Sysnovare the developers are not used to start developing a feature by running the tests. But, it is expected a long term benefit: less bugs to fix.

3.1.3 Behaviour Driven Development

Behaviour Driven Development (BDD) is a software development methodology which evolved from the previous discussed TDD. During the last years the BDD approach, developed by Dan North, has gained more visibility as an agile development approach [SW11]. Similarly to the TDD practices, on the BDD approach the tests are also what guide the software development process.

However, one of the major concerns of BDD is how the communication between developers and stakeholders can be improved [SWD12], highlighting the communication issue. But, it is not only about the communication that the BDD is concerned about. In fact, this different approach on the software development processes, appeared from the need of responding to a few issues to which the TDD practices can not answer.

Dan North, the pioneer of this method, says that developers "wanted to know where to start, what to test and what not to test, how much to test in one go, what to call their tests, and how to understand why a test fails" [Nor06]. To solve these difficulties, a new approach on software development processes emerged.

3.1.3.1 Behaviour Driven Development Characteristics

There are three characteristics that stand out on this development process:

- **An Ubiquitous Communication** With the aim of providing a common understanding [SWD12] between all project members, the BDD approach suggests the usage of natural vocabulary [Nor06]. The the natural language provides a common understanding of a system between both the developers and clients [SWD12]. This enhances the participation of all the project members, encouraging a group work between all members. The use of this ubiquitous communication is specially useful and important during the elicitation of requirements [SW11] and the definition of acceptance tests. But it continues to be used throughout the development life cycle [SW11].

  To use this ubiquitous communication it is necessary to define the ubiquitous language. In fact, this is done at an early phase of the software development process: the analysis phase [SW11], to allow an early understanding between all team members. Defining the language means choosing a group of words, the vocabulary, that can be used to describe the business behaviour [SW11]. Even during the implementation phase, the developers are supposed to use the defined vocabulary to define methods, functions or classes names [SW11].

- **The Iterative Process** One of the issues that this approach is willing to answer is the developers difficulty on finding a starting point for all the process. Therefore, in BDD, the analysis starts with the identification of the expected behaviours of the system [SW11]. At this point, it is very important to have the participation of the client, to produce a collection of business outcomes. It is from breaking down these outcomes that the system's features are defined [SW11].

  Furthermore, a feature is still divided in smaller units: user stories. The user stories describe the interactions between the users and the system [SW11]. These interactions can occur in different contexts. In BDD the different contexts are called scenarios, which are defined by the clients and work as acceptance tests [SW11], since they define how the system should behave when it is in a specific state and an event happens [SW11]. At this point, a question emerges: how are the scenarios defined? And this leads us to the final characteristic of the BDD method.

- **Scenarios' Description** To describe the different scenarios for a feature, besides the need of the ubiquitous language, it is also necessary to have a common model to follow. Before writing the scenario, it is necessary to define the story which normally follows the template described below [SW11]:

  [StoryTitel](Short description of the story]

  As a [Role]

  I want a [Feature]

  So that I can get [Benefit]

  Then, it is possible to define the scenario. The template to write scenarios is as the one described below [SW11]:

Given [Context]

When [Event Occurs]

Then [Outcome]

It is possible to notice that there are key words that have to be used since they are directly mapped to define the acceptance tests.

3.1.3.2 Behaviour Driven Development on this Project

With BDD it is possible to define tests using a set of defined words from a natural language. In the scope of this project, where one of the goals is to have non-technical people defining the tests, this approach can be applied, specially on the definition of the UI tests. To apply this approach to test the behaviour of an application, it is necessary to have the right tools for it. A topic we will discuss during the following sections.

3.2 Tools for API Testing

"The only way to know if an application programming interface (API) makes sense is to use it" [Adz09] or to test it. This section of the 3*rd* chapter is about knowing which are the possibilities to test an API before using it, which is feasible when the software development process has as a first step the tests definition.

To adopt a development process steered by tests, it is fundamental to have good tools helping on this task. The objective is to find tools that can be used in a simple way, not to bring too much complexity to the testing process. Tools which do not require a lot of effort to be used and no specific know-how on any programming language, nor technical proficiency.

After analysing and trying some tools with the purpose of implementing API tests, we decided to go for a relatively recent tool which we will look in detail during the next section.

3.2.1 FitNesse

FitNesse is a web wiki front-end for a framework called Framework for Integrated Tests (FIT) [Adz09]. FIT is an acceptance testing framework developed by Ward Cunningham which aims at providing a fast and easy way to run tests [Adz09]. But Robert Martin and Micah Martin wanted to go further. They realized that in some way the developers' and clients' vision of a feature was not exactly the same. So, they seeked a way of allowing clients to read the tests built by the developers or, even better, to allow clients to define their own acceptance tests [Adz09].

They came up with the idea of using the already working FIT but adding a layer above, as an interface to allow people with non technical know-how to define their own tests. This was the purpose for developing FitNesse.

FitNesse provides an environment in which the user can define his tests, run them and, easily, read the tests results [Adz09]. Moreover, its principle is enabling "non-technical users to collaborate on writing tests" [Adz09]. Having a business analyst or a client setting up their own tests is the main goal of this tool. To sum up, the idea behind FitNesse is having "non-technical people involved with the testing process" [Adz09].

3.2.1.1 FIT, FitNesse and DbFit

While FitNesse main purpose is to enable an interface where to define the tests, there is another project which enables FitNesse tests to be executed directly against a database [Adz08]: DbFit.

DbFit is an open source project, available at https://github.com/benilovj/dbfit, which, as illustrated on figure 3.3, combined with Fit and FitNesse results on a database unit test tool [Adz08].

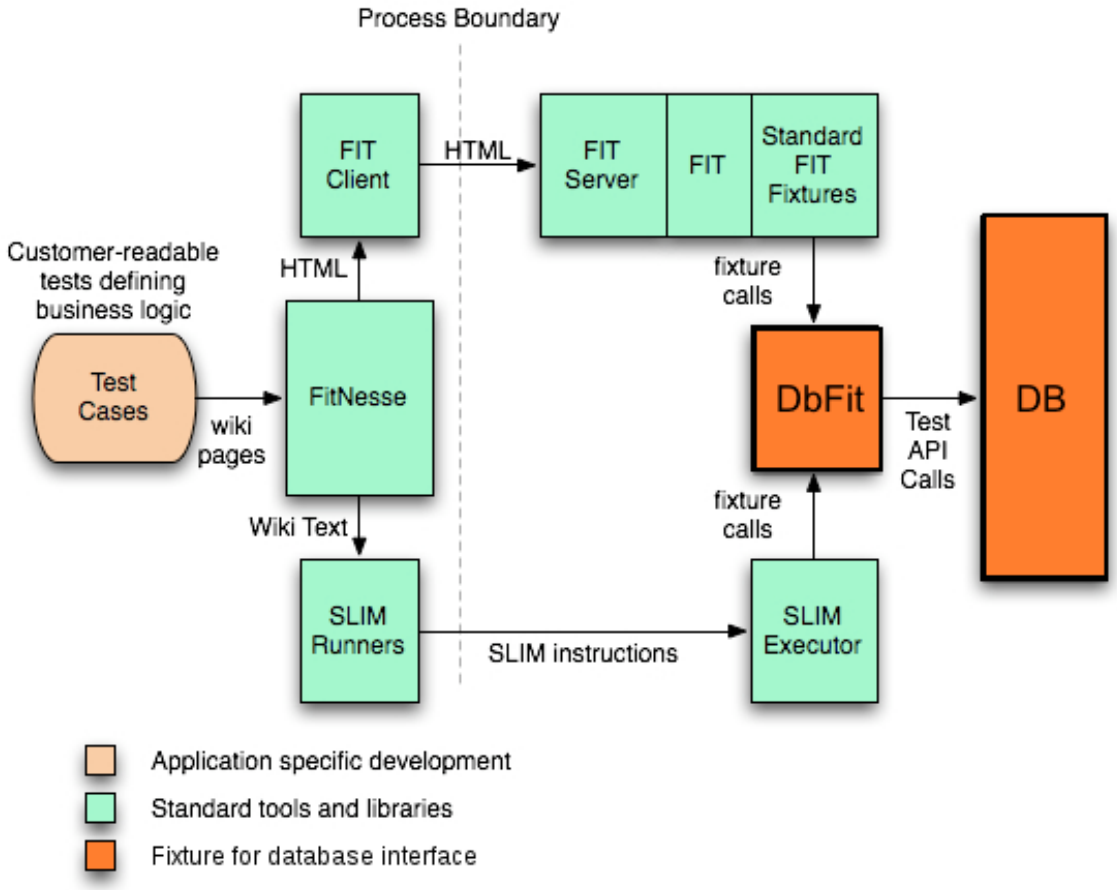

Figure 3.3: FIT, FitNesse and DbFit [Adz08]

As it is possible to see on on figure 3.3 DbFit is a fixture which is integrated to work with FIT and FitNesse. A fixture is a class which is called by FIT to process a test table every time the test button is pressed [Adz08]. The fixture is only responsible for retrieving the results which would be used by FitNesse to determine whether to turn output cells green or red [Adz08].

The DbFit fixture offers support for different types of databases: Postgres, MySQL and Oracle, for example. It is written in Java, using the JDBC API to interact with databases, and is organized in two parts: one is common to every database types; then there are specific classes for each type of database. In this case we will focus on the part of the project which is specific to the Oracle database.

As we will see on the following section, FitNesse enables evoking PL/SQL functions and procedures allowing the comparison of the obtained results with the expected ones. Therefore, even though the main purposes of FitNesse is not unit testing, it can still be used to test business rules.

3.2.1.2 FitNesse in Practice

During this section we are going to introduce some examples on how FitNesse works. The following figure 3.4 is an example of a test definition.

```
!|execute procedure|tw_exp.get_department_name|
|department_id|?|
|1|Computer Science|
|2|Mechanical Engineering|
|3|Chemical Enginneering|
|4|Biological Engineering|
```

Figure 3.4: The FitNesse Syntax

This is how testing tables are created: through a minimal wiki interface and a simple syntax.

This test definition is testing the `get_department_name` function. As an input for the function execution, it receives just an argument - `department_id` - and it has an output which is identified by the `?` symbol. The test is evaluating the returned value for the name of the departments with `id` from 1 to 4. For each of these input arguments the expected result is the value defined on the second column of each row.

After defining the test, it is necessary to save the test and then a new interface is available. This new wiki page allows the user to check the defined tests, this time on a tabular form as illustrated in figure 3.5. The authors of this tool agreed that this tabular form is what makes it "very easy to write tests and view results" [Adz09].

execute procedure	tw_exp.get_department_name
department_id	?
1	Computer Science
2	Mechanical Engineering
3	Chemical Enginneering *expected* Chemical Engineering *actual*
4	Biological Engineering

Figure 3.5: The FitNesse result table

The results are displayed on the same tabular form, using the green-red light to indicate if a test passed or not, as we can see in figure 3.5. In case a test fails, it also indicates which was the obtained result.

3.2.1.3 FitNesse on this Project

Overall, FitNesse is a tool which allows defining tests, running them and obtaining a result. There is no doubt that with FitNesse it is easy to read the tests: identifying what is being tested, which are the test inputs, which are the expected results and the obtained results.

However, DbFit has some constraints in what concerns the support for Oracle PL/SQL data types. Data types such as VARCHAR or NUMBER work well on DbFit, when used as parameters on procedures/functions or as returned values on functions.

On the other hand, DbFit launches errors from the JDBC API in case we try to test a procedure which returns a BOOLEAN type, for example. It also raises an exception when the procedures return a RECORD type or has a RECORD type as a parameter. The same happens with TABLE types, for example.

Furthermore, DbFit does not execute the right PL/SQL procedure/function when there is more than a procedure/function with the same name.

So far, it is possible to see that there is still work to be done on this tool so it can be used to test Oracle PL/SQL programs.

3.3 Tools for GUI Testing

Testing web interfaces can get very complex when an application keeps on growing. It is hard to assure that a user who tests an application of extensive dimension, by manually covering the interfaces will not miss a button, a menu or any other feature present in the application. These minor details can sometimes have an undesirable impact.

In fact, it is necessary to automate the process of testing web applications, specifically, the user interface. Therefore, on this third section we are going to explore a tool that came with purposes of automating browsers [Sel04].

3.3.1 Selenium

Selenium is an "open source browser automation and testing library" [Adz09]. It appeared in 2004, by Jason Huggins, with the promise of automating browsers [Sel04]. Selenium is a capture and replay tool which is capable of capturing the screen status and later comparing it to another screen.

Defining a test in Selenium is as simple as pressing the record button of its IDE and using the web application as the user normally does on a everyday usage. After pressing every desired button, typing text into the text fields and checking the pages' contents, the user can stop recording.

At this moment, the test is defined and ready to be run, to test on its own the web interface of the application.

To run the test, the user clicks on the play button. It is possible to check that, on its own, the browser is performing the same actions recorded earlier. A test fails when the image captured is not the same as the one captured during the tests definition. As it is possible to see, Selenium is a very simple tool and easy to use.

3.3.1.1 Selenium in Practice

Selenium offers a Firefox plugin, the Selenium-IDE, which enables the definition of test suites. As is displayed on figure 3.6, while covering the application interfaces, Selenium stores a list of commands which later are executed to replay the tests.

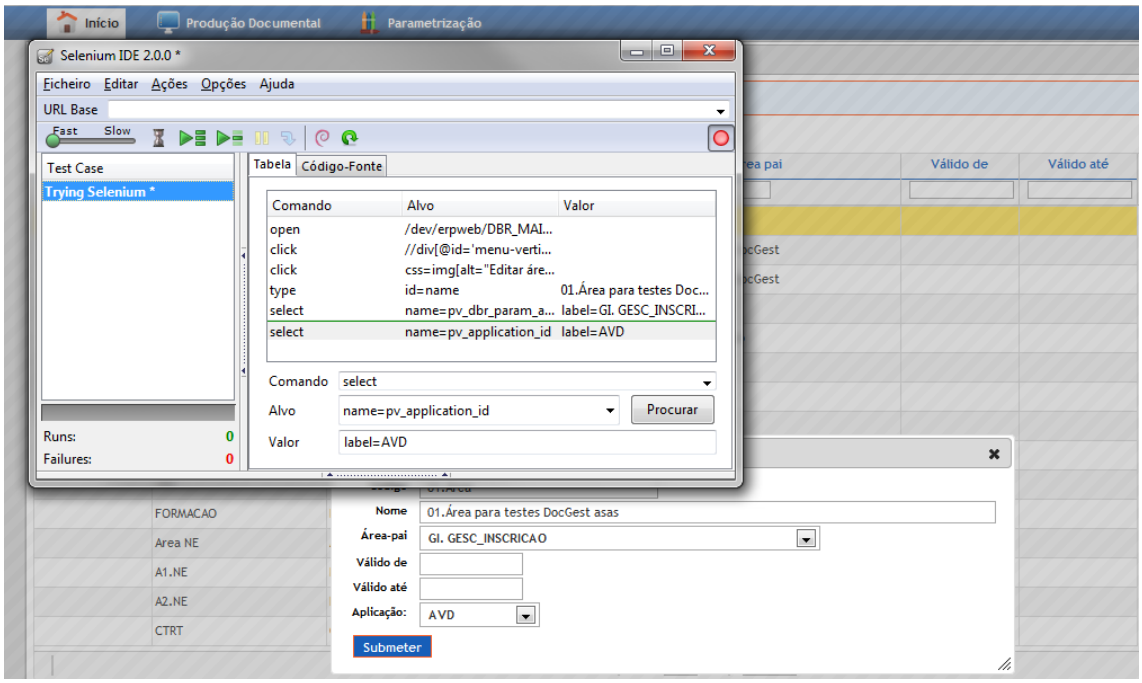

Figure 3.6: Selenium in use

The commands represent the actions the user performs (click, type, open) and they are followed by a reference to the HTML element on which the action was performed. These references are designated by *Locators*. Selenium can look up a HTML element using various possibilities of *Locators*, for example: id, name and xpath. By default, Selenium first tries to identify an element using its id attribute, when this is not available on the element, it searches for the name attribute and it continues running down the available *Locators* until it is capable of identifying the element.

It is important to mention that despite the Selenium-IDE being only available as a Firefox plugin, the tests defined on this browser are not restricted to it, which would be a significant drawback of the testing tool. The tests developed with Selenium-IDE can be run against other browsers, however they have to be executed by the Selenium-RC server [Sel04].

3.3.1.2 Selenium on this Project

Selenium-IDE allows the definition of test suites by using it as a Firefox plugin. The objective of using Selenium is being able to define the tests by just clicking on the interface and not having to use any specific knowledge of HTML concepts nor the javascript language. It is mainly due to these characteristics of this testing tool that we believe it is ideal for this project since it enables non technical people being independent on using it.

Unfortunately, after a few experiences while trying to use this tool on Sysnovare applications we have faced some issues. These issues can easily become true challenges for non technical users. For example, it was not possible to use Selenium to click on images only visible with the hover action and typing on autocomplete elements was also not possible. These are just two examples of constraints of this tool. They can be overcome if the user, defining the test, types himself commands or part of it as a work around for these issues. However, this solution is not applicable for the non technical users and above all this is the opposite to what Selenium aims at offering to its users.

Therefore, it is possible to see that there is still work to be done on this tool if we want to conquer the objective of having non technical users independently using the tool.

3.4 Conclusion

Throughout this chapter we looked into detail on the existing information regarding the three different subjects involved on this project: testing methodology, API testing tool and GUI testing tool.

Looking at the XP methodology proven to be very interesting. It is a methodology where tests have an extremely high importance. In fact, they guide the whole process of developing a feature for a product. The reading and analysis performed at this level aimed at evaluating the possibility of including or adapting these ideas on the testing process to be suggested.

Furthermore, it was fundamental to devote time to study, discover and explore the two testing tools mentioned on this chapter: FitNesse and Selenium. This allowed deciding if they are useful for the objectives of this project. Moreover, this exploration phase also allowed knowing what to expect from each tool, which are their limitations, analyse whether the existing limitations are possible to overcome or not. With this information we were better prepared to define a work plan that can lead us to fulfil this project objectives.

Chapter 4

Proposed Testing Methodology

The objective of this chapter is to describe the ideas which arise from the integration of testing practices at Sysnovare.

On chapter 2 we have stated that the testing process was not an area widely explored by Sysnovare. There were no testing process participants nor activities defined. An area with few investment however extremely important for the quality of the software. Therefore, the testing process could be greatly explored and enriched with various new contributions and new ideas. So, the work we have done at this level focused on defining rules for the process and, finally, how the suggested testing tools fit on this picture.

4.1 Release Life Cycle

To expose the testing process it is important first framing it on the four main moments of a release life cycle, which was previously covered, in more detail, on chapter 2. Here we bring this topic again to contextualize the ahead description of the testing process workflow.

The release life cycle shown on figure 4.1 comprises the phases which are gone through to achieve the ultimate aim: deliver features to a client.

Each of the phases of the release cycle are named after the database environment where they take place. The first phase of this cycle is the *Development*. It is the phase defined to develop the features included on the release.

Then, there is the *Testing* phase. It is the second moment of the release cycle. It is devoted to: (i) analyse whether the developed features are according with what was previously decided; (ii) check the behaviour of the application; (iii) find bugs; (iv) find unhandled exceptions. If this pursuing for weaknesses on the application is fruitful, the release cycle returns to phase one. It is time to correct the identified bugs. Only when there is no flaw, on the scope of the features defined for the release, can the cycle move forward to phase number three: *Acceptance*.

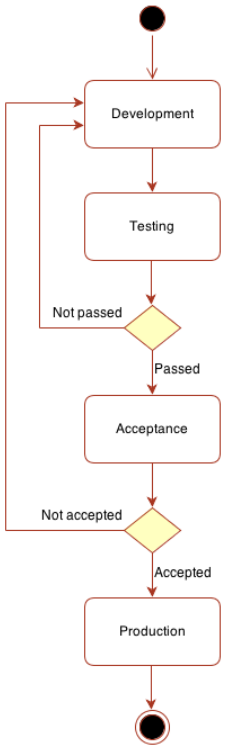

Figure 4.1: Release Life Cycle

The *Acceptance* phase is reserved for the client to test the application before it is deployed. This takes place on a database schema that replicates the production environment as reliably as possible. The client has the opportunity to test the release freely and then choose to accept it or not. When the release is not up to the requirements, the client should, at this point, ask to fix the spotted bugs. Once again, the release cycle returns to phase one. The bugs are fixed, the testing phase is once again covered and we are back to the phase when the client approves the behaviour of the application.

So, we reach the final phase which is releasing the version of the product for usage in the production environment.

4.2 Testing Activities

Despite just one of the aforementioned steps being called *Testing*, all the other steps also require a testing activity and are included on the testing process workflow which is described throughout this section. This section includes the detail of the testing process: who takes part on each moment, how each task is done and which are the defined rules for the testing activity as a whole.

Testing Sysnovare products requires testing them on four different contexts: the development, the testing, the quality and the production schemas. The difference between each schema and its purpose was briefly mentioned on the previous section. Taking these differences into account we based the definition of the testing process. Depending on the schema, the testing actors vary, the

activities and the way they are performed are also different and there are specific rules for each schema.

Figure 4.2 illustrates the testing activities during a release (section 4.1), organized based on the four different schemas. Moreover, as the scope of this work has two different test layers - API and GUI - there is a clear division for each layer. These testing layers, depending on the schema we are working on, have different scopes on the testing process. But this is something which we are going to specify through the following description of the testing process.

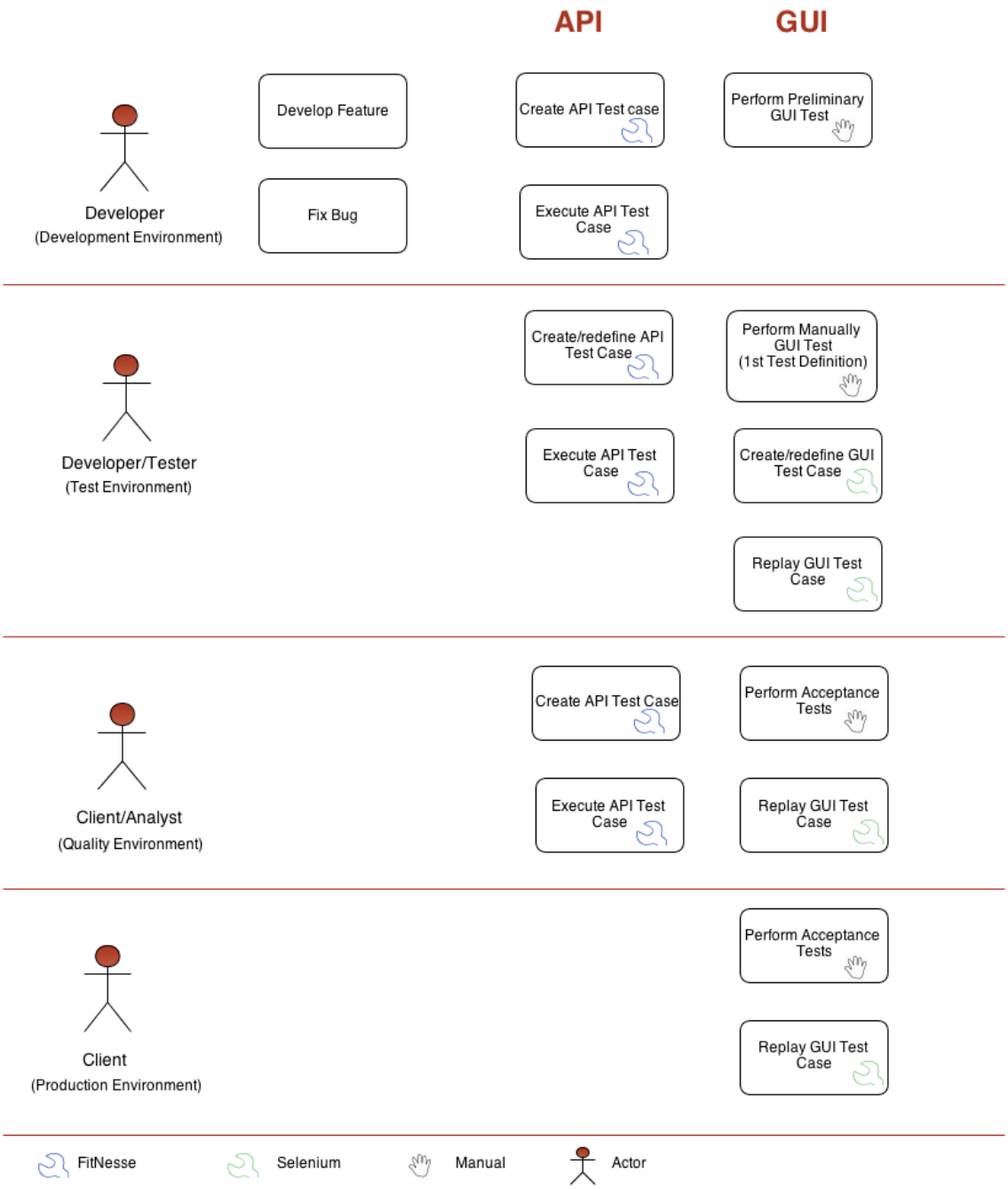

Figure 4.2: Testing activities and participants

The testing process starts at the first phase of a release life cycle: *Development*. After developing the features, the developer is expected to test the module on which he has been working.

Testing the module, in what concerns the API scope, may require the definition of new tests. Each time a developer implements code for a new business logic, he is responsible for creating the respective test cases (*rule number 1*). Reinforcing the importance of the API tests, the definition of these tests contributes to the robustness of the business rules. Besides creating new tests, the developer is obliged to run the already existing tests. This allows verifying whether the already working business logic is still correctly implemented. Fortunately, this part of the testing process can rely on the precious help of the automatism of this activity, only possible thanks to FitNesse. This execution has an output which includes the errors found during the tests' execution. This output defines whether the developer can close his task. When all the tests are green, the developer can set the feature as being completed (*rule number 2*). Otherwise, the developer has to fix the code or change the existing tests. In case a business rule has changed its logic and the tests do not reflect this change, as it is expected, the tests will fail. Therefore, when the business logic changes, it is necessary to reflect the changes on the defined tests.

On the other hand, there are the GUI tests. Unfortunately, on the Development environment, these tests do not have the pleasing help of a testing tool. At this first phase of the release, it is not guaranteed that the user interfaces are completely defined and validated. So, we believe that investing on a test definition at this phase is not worth the work. At this point, the interface validation is only about checking whether the interfaces are conforming structurally and browsable.

The second phase on this testing process workflow is called *Testing*. This phase takes place at a database schema created specifically to test products. It is a schema with artificial data to perform tests. Tests which were defined based on the data used to automatically populate the database.

After the developer installs the release on this schema, he should perform the preliminary tests. The activity consists on ordering the execution of both the API and GUI tests. This preliminary step is necessary to identify possible errors due to an unsuccessful installation. After this first evaluation, the tester should also run the tests, with the aim of verifying the accuracy of the business logic and the correct behaviour of the interfaces. At this point, the tester has the opportunity to redefine the API tests. The need of redefining the tests can come from the analysis of the defined tests, concluding that some application states were not being taken into account. When the API tests are redefined at this phase, they should be uploaded to GP in order to take part on the test suite on a following development (*rule number 3*).

In what concerns the GUI tests, there are four possibilities on how to perform them. (i) In case we are testing an interface which had no changes since the last release and had already GUI tests defined, it is a simple task of ordering to run the automatic Selenium test. (ii) The second possibility is an interface which was redefined since the last version and had already a test defined for it. If we run this test, we will probably obtain an error. Therefore, as the interface was redefined, the Selenium test also has to be redefined (*rule number 4*). (iii) There is also the case when an interface cannot be automatically tested since it has no defined test. It is on this *Testing* schema that

the tester has to perform the task of capturing the Selenium test (*rule number 5*). (iv) Finally, there is the possibility of manually testing the interface just to check if it is browsable. This happens when an interface is not completely defined or approved. In order not to invest resources and time on defining a test and after having to reformulate it, the test is performed manually.

Moving on to the next phase, *Acceptance*. This phase takes place at a database schema called *Quality Environment* which replicates the *Production Environment*. Here the tests are performed by the client. He should be able of freely and autonomously run the automatic tests, API and GUI.

In this case, the API tests cannot be the same of the previous phase. At this database we do not have total control on the data, as we do on previous schemas. Therefore, the API tests require an extra effort of defining them. A task on which the tester should help the client. On a following iteration it is important, prior to run the tests, to verify if the data from a previous test is still valid. In case it is not, it is again necessary to define the API tests.

Regarding the GUI tests, in this case, they do not require any special attention, except for those interfaces which do not have defined tests, which have to be manually tested. The others just require the replay of the defined tests, using Selenium.

Finally, there is the last moment of the testing activities: testing the product on the *Production* environment. Here, there are certain specificities which it is necessary to take into account.

At this environment, it is not possible to create, edit or delete the existing data. Therefore, it is risky to perform the API tests. Normally, these tests do not only involve returning information. Sometimes, they require a previous update on existing data. That is why this phase does not include any tests at the API layer, nor automatically nor manually.

It is also necessary to pay attention to which GUI tests we are automatically running. If the tests just perform actions of opening windows or dialogue boxes, we do not have to worry about them. However, the majority of the testing suites include inserting or updating information that is displayed on the interface. Due to the importance of not changing the data on this environment, it is safer to perform these tests manually.

4.3 Rules

Throughout the previous section, while describing the testing activities, we have defined some rules for the testing process. They were mainly related with who defines the API tests, when they are defined, what to do with the new test definitions and how a development activity is influenced by these tests. These totalized a number of five defined rules.

However, these are not the only rules which we have come up with for the testing process. To add to these five rules there are three more rules worth to mention.

Rule number 6

This rule is related with the GUI tests. It is an aspect which influences the type of the performed tests - manual or automatic - on each of the aforementioned phases.

Defining interface tests with Selenium still requires time investment. But even though this time effort is not too long, we believe there is no need to perform this task more than once. The repetition of this task could occur if we define a test for an interface which is still not stable. Therefore, to define an interface test, it is mandatory to assure that it was already validated.

Rule number 7

On chapter 2 we have said that a release can consist of a version or a patch and we explained the differences between the two. Basically, the release of a patch includes the correction of bugs or the improvement of existing features. It does not include developments of new features. If there is no new business logic nor changes on the interfaces, testing a patch always means running the automatic tests, which were used on the last released version.

To sum up, in a context of releasing a patch there is no space for changes on tests both at the API and GUI layers.

Rule number 8

The last rule is concerned with the identification of bugs on the *Production* environment. Identifying bugs at this phase is not positive for the applications image and does not reflect an effective testing process. It is important to remember that to get to the *Production* environment, the bug passed undetected through three phases. Three phases of the testing process where the defined tests were not capable of detecting the bug.

Therefore, the identification of a bug on the *Production* environment has to lead to the definition of a new test case. This way, we widen the scope of the test cases and work on the increasing of the thoroughness and reliability of the testing suites.

4.4 Conclusion

We have set as an objective the definition of an effective testing methodology to the development process of Sysnovare. We approached this topic by first studying how the development process happens at Sysnovare. We concluded that this process is not coherent for all the Sysnovare products, however, there is a pattern they are trying to achieve. It was considering the development process to which they are converging that we defined a testing process to fit on it.

This resulted on a group of activities (figure 4.2) and rules. It is a process with defined activities, by whom they should be performed and how. We contributed to the automatism of this process, by integrating the testing tools on the definition of how the activities should be performed.

Regarding the defined rules, they are a fundamental contribution for this methodology since they guide and uniform the behaviour of those who take part on the testing process.

Therefore, we believe we contributed to the enrichment and effectiveness of the Sysnovare testing methodology, adding a great improvement on the testing area.

Chapter 5

API Test Automation

This chapter focus on the work developed at the API test layer. This is the layer of web applications where all the business logic is defined. On this project context, where the applications are developed using the PL/SQL programming language, the business logic is grouped in packages, in the majority of the situations. Therefore, our first task was choosing a tool which proved the more helpful on evaluating the accuracy of the implemented business rules. Specially, we focused on a tool which was able to help automating this level of testing.

The chosen tool is FitNesse, more precisely DbFit. This tool was already described on section 3.2. On that section, a first overview of the limitations of DbFit concluded that were needed several improvements. Constraints regarding the support of the Oracle data types, like Boolean and Record types, and the overload of procedures or functions are the examples of the identified constraints.

The work developed at this layer is aimed at improving DbFit so it could be used efficiently to test the business logic of Oracle applications. As it was also previously stated, one of the major advantages of using DbFit is the possibility of suggesting improvements and new features to the tool and help on their implementation, since this is an open source tool, with an organized an active project on GitHub.

Therefore, during the following sections of this chapter, we are going to describe, in more detail, the constraints related to DbFit that were the focus of our work, the path made to achieve the implemented solutions, together with the other developers of this tool, and which are the improvements obtained with this work. Afterwards, we describe how FitNesse was integrated at Sysnovare and which were the obtained results.

Finally, we conclude this chapter stating the next steps regarding new suggested features to implement on DbFit and to enrich it in what concerns the Oracle module. Moreover, we comment the following actions to continue the work of integrating the DbFit at Sysnovare.

5.1 Modules integrated with FitNesse

During the next sections, following the description of each of the improvements on DbFit, there is a section to summarize the impacts that each improvement had on Sysnovare applications. Therefore, prior to describing the implemented work, this chapter starts by detailing the modules of Sysnovare products that were used with the purpose of giving relevant feedback on the use of the new testing tool.

There were two modules which were used to launch the use of FitNesse at Sysnovare: Automatic Payments and Payments Management.

The first module, Automatic Payments [AUTOPAY], is a module which was developed to be integrated with other projects, adding the concept of electronic payment to the applications. AUTOPAY has a very well defined collection of requirements. The business logic is based on solid and hardly changeable rules which makes it easy to develop the necessary code to fulfil the business requirements. It is a module which is very isolated, it is not dependent on any other modules of the applications. These three characteristics - straightforward, solid and independent - were vital to the decision of this module as an experimental module to test with FitNesse and DbFit and assess their adequacy regarding the existing needs of Sysnovare applications.

Table 5.1 provides key information on this module: number of business rules to test, number of clients using this module and number of schemas where the module is installed and has to be tested.

Table 5.1: Automatic Payments Module's Properties

Properties	Number
Business Rules	38
Clients Using the Module	6
Schemas where the Module is Installed	8

As table 5.1 illustrates, there is a relevant number of business logic which has to be tested for this module. With the automation of the testing process, using FitNesse, it would not be surprising to expect a decrease on the time needed to test it. Moreover, the tests will verify the accuracy of the AUTOPAY module, bringing more confidence to the six clients using it.

The other module, already interacting with FitNesse, is the Payments Management [GPAG] module. This module is integrated in a major, industry-leading product for academic management. Giving a brief description of this module: it is used to create registers of what a user has to pay and, from this starting point, manage a collection of other actions related with the created payment.

Unlike the previous module, this module required some work on organizing the code. This testing tool is designed to test procedures and functions very isolated. Therefore, it is ideal for projects organized following the MVC concept, described on section 2.2.2. However, GPAG was not organized by the established rules which influences on the possibility of being correctly tested.

Some refactoring on the code was made: isolating, for example, queries used on the view layer and reallocating them to the model layer; reusing the same procedure on different places and rethinking some procedures. Having the code organized following the established rules it is easier to test the module using FitNesse.

We chose this module taking into account the variety of PL/SQL procedures and functions which are intrinsic to it. It was known that with this variety, it would be possible to evaluate how well FitNesse fits Sysnovare needs.

GPAG, as it is possible to see on table 5.2, has a vast number of business rules to test. It is also being used on a relevant number of clients. So, the more effective the tests on this module can be, the better for both the Client and Sysnovare.

Table 5.2: Payments Management Module's Properties

Properties	Number
Business Rules	55
Clients Using the Module	7
Schemas where the Module is Installed	9

5.2 Support for Oracle Boolean data type

In PL/SQL there is a data type to represent *Boolean* that can take any of the following values: true, false and null [Agr05]. This data type can be used on procedures or functions as the data type of parameters or it can be used as the return data type of a function. However, JDBC, the API used by DbFit to connect with databases, does not have this flexibility for handling Boolean data types.

The JDBC documentation suggests a workaround to handle this data type. The basic approach includes translating the *Boolean* value to a *Number*, so that JDBC can work with the *Number* data type [ORA11].

5.2.1 Solution

The discussion around this implementation began following the suggestion on the JDBC API: converting the *Boolean* value *true* to a 1 and the value *false* to a 0. However, very accustomed with the DbFit project, its contributors agreed that a wrapper to convert the *Boolean* type to a string was a better alternative. It was also decided that instead of including this logic on the main class of the Oracle support, a specific class would be added to handle *Boolean* data type.

Therefore, it was created a new class *OracleBooleanSpCommand* specific to support the *Boolean* type. This class has methods responsible for building the wrapper which generates the database call, as it is shown on listing 5.1.

```
1 public void generate() {
```

```
2         append("declare\n");
3         genBool2Chr();
4         genChr2Bool();
5         genWrapperSp();
6         append("begin\n");
7         append("    ");
8         genCall();
9         append(";\n");
10        append("end;\n");
11        append("\n");
12    }
```

Listing 5.1: DbFit database call to support Boolean types

In this method, a call to convert the *Boolean* type into a *Char* is made. Basically, it is a PL/SQL function (listing 5.2) which receives a *Boolean* parameter and, depending on its value, returns a string: 'true' or 'false'.

```
1  function bool2chr( p_arg BOOLEAN ) return VARCHAR2
2      is
3      begin
4          if ( p_arg is null )
5          then
6              return null;
7          elsif ( p_arg )
8          then
9              return 'true';
10         else
11             return 'false';
12         end if;
13     end bool2chr;
```

Listing 5.2: Extract from DbFit procedure to convert Boolean into Char

A fundamental part of this development was building tests to validate the new feature. So, we have built, suggested and validated different types of tests to validate the various possibilities of using *Boolean* types on PL/SQL: *in*, *out*, *in out* parameters and return values.

It is important to mention that we had two alternatives to handle this issue: add the feature to DbFit or implement a wrapper, albeit on the Sysnovare code side. After carefully evaluating the two possibilities we decided that in spite of the second alternative being less complex to implement and requiring less effort, it would be more useful if all the necessary testing logic stays abstracted behind DbFit. Otherwise, testing a function with *Bollean* types would require setting up activities, instead of only evoking the function to test.

5.2.2 Results

After the development made to support the *Boolean* data type, we recorded an increase on the number of functions whose test is supported by the DbFit. As it is possible to see on figure 5.1, this contribution to the project makes it possible to test, approximately, more 15% of Sysnovare functions.

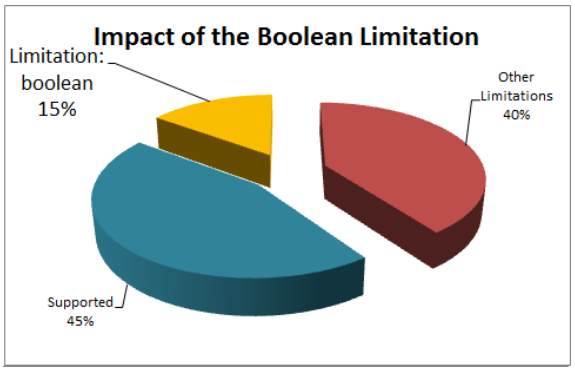

Figure 5.1: The impact of the Boolean data type limitation

As it is possible to conclude from the data discussed above, the support for Oracle Boolean data types, contributed positively to the performance of DbFit. We should emphasize that this development is not only useful for Sysnovare, but also of major importance for all of those using or wishing to use DbFit for Oracle databases. Due to the importance of this feature, it was promptly included on the ensuing version, the 2.0.0 RC5 version of DbFit.

5.3 Support for Oracle Record data type

The issue around the data type *Record* appeared while executing the first tests of the GPAG module. In this module 13% of the functions return this type of record. This type is declared as **%ROWTYPE** and it represents a row in a database table [ORA11]. Unfortunately, JDBC, once again, does not support this PL/SQL type. Therefore, similarly to what was done for the *Boolean* type, the *Record* type also requires a workaround to be possible to test functions on DbFit which return Oracle *Record* types.

So, after exploring even more the DbFit project, we have realized, by checking the examples available on its repository, that DbFit is already capable of returning data sets. DbFit handles the execution of queries, as it is possible to see on the example on figure 5.2.

However, the command *Query* has different handlers from the *Execute* command, which is the command used to test the business logic functions. Even though this is not the feature we are looking for, it is a starting point to begin the implementation of this support.

query	select id, name from departments
Id	name?
1	Computer Science
2	Mechanical Engineering
3	Chemical Engineering
4	Biological Engineering

Figure 5.2: DbFit query execution

5.3.1 Solution

The starting point of our implementation was, therefore, the known ability of DbFit to return a collection of results.

We begun by modifying the `OracleEnvironment` class adding the *Record* type to the group of *Cursor* types that can be normalised by DbFit, as it is shown on the 5.3 code snippet.

```
1  private static List<String> refCursorTypes = Arrays .asList(new String[] { "REF",
       "RECORD" });
```

Listing 5.3: DbFit Handling RECORD types

The following step and one of the most challenging steps, was to take advantage of the operations performed when the *Query* command is invoked and use almost the same procedures but restricting the number of results to just one. So, until now this is what we have been capable of doing in what concerns this issue.

The problem we are now facing is how to represent and subsequently map the expected results and the obtained ones. With the functions that only return a value, the place for the expected result on the test definition is simply represented by ? . However, when returning a *Record* type we are expecting more than a result. Until now, we have not been able to decide on how to handle this issue: (i) without affecting the actual behaviour of DbFit in what concerns the already working tests; (ii) without changing the *isOutput* method (a core method of all the project); (iii) and handle the definition of the tests with the expected results declared in a different order from the database columns of the *Record* type. This is all work in progress, fundamental for any Oracle testing tool.

5.4 Improve error messages

The last two sections discussed unsupported data types: boolean type and record type. Faced with these issues a solution to improve the error messages was suggested by the DbFit contributors.

The DbFit error messages emerge from the exceptions raised on the `OracleEnvironment` class. This particular exception has the type `UnsupportedOperationException`. The

`OracleEnvironment` class has a method - `getSqlType` - which makes the correspondence between the received data types from the `all_arguments` table and the SQL types. This operation is applied to every argument of the function being tested, from the input to the output arguments. In case the argument data type was not on the list of the SQL supported types, DbFit would raise an exception: PL/SQL type not supported.

Unfortunately, this message was generic in what concerns the specification of the unsupported data type. It was from the need to refine the error message that we took on this development. This development consisted on preserving the data type originated from the `all_arguments` table. Instead of normalising this type to the generic PL/SQL type, it is maintained and used on the final error message.

Improving the error messages for the unsupported data type is of great usefulness since it allows identifying the concrete source of a problem. The correct identification of a problem is the first step to accelerate its resolution.

Since this improvement consolidates the DbFit project, it was also included on the 2.0.0 RC5 version of the DbFit project.

5.5 Overloading of procedures or functions

The fourth problem, covered during the work developed for DbFit, is related with the overload of procedures or functions. In this context, the term overload means having more than a procedure or function with the same name. This scenario, of equal names, is only possible inside PL/SQL packages [Agr05]. The standalone procedures and functions are not allowed to have the same designations. But even inside packages, the overload has to respect a rule: the procedures or functions parameters have to differ in number, order or data type family [Agr05].

In contrast to DbFit, the PL/SQL compiler is capable of resolving sub-programs invocations, even if they have the same name as others. PL/SQL resolves the invocations by searching a declaration which matches with the invocation. The matching is achieved when the names and parameters list are the same [Agr05].

The following code snippet consists on three examples of possible overloaded procedures and functions. They are going to be used as an example to explain the implemented solution.

```
1  CREATE OR REPLACE PACKAGE TW_EXP
2  AS
3
4    FUNCTION t_department (department_id IN NUMBER)
5      RETURN VARCHAR2;
6
7    PROCEDURE t_department (department_id IN NUMBER);
8
9    PROCEDURE t_department (department_id IN NUMBER, department_name OUT VARCHAR2);
10 END TW_EXP;
```

Listing 5.4: Overloading examples

5.5.1 Solution

The work developed to overcome the *Overloading* issue was based on the rule which the Overloading follows. At a first glance, it may seem trivial to implement a solution based on having procedures with different number of parameters, different data types and different parameters ordering, specially because it is already the behaviour of the PL/SQL compiler, however, this development turned out to be a bigger challenge.

First, we started by identifying the point at which and how DbFit was obtaining the procedures to be executed. This search led us to the `getAllProcedureParameters` method on the `OracleEnvironment` class. This method builds a SQL statement which was the focus of our work. [1]

```
1 select argument_name, data_type, data_length,  IN_OUT, sequence
2   from all_arguments
3  where data_level = 0
4    and ((owner = ? and package_name is null and object_name= ? )
5       or (owner = user and package_name = ? and object_name = ?));
```

Listing 5.5: Extract from original DbFit query to get the arguments of a procedure

As it is possible to see on listing 5.5, this query retrieves data from the `all_arguments` table. The `all_arguments` table consists on a list of arguments of all the procedures and functions available on a database user [Agr05].

For the query in listing 5.5, the retrieved results are restricted by the procedure's name and the procedure's package. In case, the test being executed is the one indicated in figure 5.3 all the registers indicated in figure 5.4 will be included in the result.

execute procedure	tw_exp.t_department
department_id	?
1	Computer Science

Figure 5.3: Test executed for Overload example

With these results, we conclude that the only column which relates the records of the same procedure or function is the `Overload` column, which indicates the *n*th overloading ordered by its appearance in the source package [Agr05].

[1] The symbol ? represents parameters received by the query.

	PACKAGE_NAME	OBJECT_NAME	ARGUMENT_NAME	DATA_TYPE	DATA_LENGTH	IN_OUT	SEQUENCE	OVERLOAD
1	TW_EXP	T_DEPARTMENT	(null)	VARCHAR2	(null)	OUT	1	1
2	TW_EXP	T_DEPARTMENT	DEPARTMENT_ID	NUMBER	22	IN	2	1
3	TW_EXP	T_DEPARTMENT	DEPARTMENT_ID	NUMBER	22	IN	1	2
4	TW_EXP	T_DEPARTMENT	DEPARTMENT_ID	NUMBER	22	IN	1	3
5	TW_EXP	T_DEPARTMENT	DEPARTMENT_NAME	VARCHAR2	(null)	OUT	2	3

Figure 5.4: Output for the original DbFit query and the Overload column

Therefore, the first approach to the problem was including the procedure's or function's arguments name on the query and use the `Overload` column to group the results and to identify which was the right procedure.

```
select argument_name, data_type, data_length,  IN_OUT, sequence
  from all_arguments
 where data_level = 0
   and ((owner = ? and package_name is null and object_name= ? )
      or (owner = user and package_name = ? and object_name = ?))
   and NVL(overload, 1) = NVL((SELECT overload FROM (
                          SELECT overload, sum(tr) tr, sum(trf) trf FROM (
                            SELECT overload, count(*) tr, 0 trf
                              FROM all_arguments
                             WHERE data_level = 0
                               AND package_name = ?
                               AND object_name  = ?
                               AND position > 0
                            GROUP BY overload
                             UNION ALL
                            SELECT overload, 0 tr, count(*) trf
                              FROM all_arguments
                             WHERE data_level = 0
                               AND package_name = ?
                               AND object_name  = ?
                               AND argument_name IN ( " + params + ")
                               AND position > 0
                            GROUP BY overload)
                        GROUP BY overload)
                        WHERE tr-trf = 0), 1);
```

Listing 5.6: Extract of the modified DbFit query to get the arguments of a procedure

Unfortunately, this approach did not work for all the cases. With this first solution it was not possible to distinguish the `t_departments` function from the `t t_departments` procedure, as it can be seen on the retrieved results displayed on figure 5.5.

It is important to reflect on these results.

There is a record, identified by the value 2 on the `Overload` column, which should not take part in these results, taking into account that the test we are performing (5.3) is testing the

	PACKAGE_NAME	OBJECT_NAME	ARGUMENT_NAME	DATA_TYPE	DATA_LENGTH	IN_OUT	SEQUENCE	OVERLOAD
1	TW_EXP	T_DEPARTMENT	DEPARTMENT_ID	NUMBER	22	IN	1	2
2	TW_EXP	T_DEPARTMENT	DEPARTMENT_ID	NUMBER	22	IN	2	1

Figure 5.5: Output for the first approach on changing the original DbFit query

`t_departments` function and not the `t_departments` procedure (characterized with the 2*nd* `Overload`).

On the other hand, there is a result which is missing. It is the register which represents the return value of the `t_departments` function. This register is not on the retrieved results, since the query 5.6 is restricting the registers where the `position` attribute is bigger than zero, which is not the case of the output arguments of a function.

The second approach to the *Overload* issue, included removing the `position` restriction and use the information we have on the output arguments, given by the test declaration, which resulted on the following query:

```
1  select argument_name, data_type, data_length,  IN_OUT, sequence
2    from all_arguments
3   where data_level = 0
4     and ((owner = ? and package_name is null and object_name= ? )
5        or (owner = user and package_name = ? and object_name = ?))
6     and NVL(overload, 1) = NVL((SELECT overload FROM (
7                           SELECT overload, sum(tr) tr, sum(trf) trf FROM (
8                             SELECT overload, count(*) tr, 0 trf
9                               FROM all_arguments
10                              WHERE data_level = 0
11                                AND package_name = ?
12                                AND object_name  = ?
13                              GROUP BY overload
14                               UNION ALL
15                              SELECT overload, 0 tr, count(*) trf
16                                FROM all_arguments
17                               WHERE data_level = 0
18                                 AND package_name = ?
19                                 AND object_name  = ?
20                                 AND NVL(argument_name, 'RETURN_NAME') IN ( " + params
                                       + ")
21                              GROUP BY overload)
22                         GROUP BY overload)
23                         WHERE tr-trf = 0), 1);
```

Listing 5.7: Extract of the modified DbFit query to get the arguments

This time, the registers retrieved were the ones illustrated on figure 5.6

	PACKAGE_NAME	OBJECT_NAME	ARGUMENT_NAME	DATA_TYPE	DATA_LENGTH	IN_OUT	SEQUENCE	OVERLOAD
1	TW_EXP	T_DEPARTMENT	(null)	VARCHAR2	(null)	OUT	1	1
2	TW_EXP	T_DEPARTMENT	DEPARTMENT_ID	NUMBER	22	IN	2	1
3	TW_EXP	T_DEPARTMENT	DEPARTMENT_ID	NUMBER	22	IN	1	2

Figure 5.6: Output for the second approach on changing the original DbFit query

Even though we were set out for the wished result, these retrieved registers still had a register which belonged to the `t_departments` procedure.

So, the last modification to the original query consisted on adding a restriction based on the number of arguments obtained on the test definition. Using this value, it is possible to restrict even more the query saying the number of registers - *trf* - obtained by the object name `t_departments` have to be on same the number - *n_args* - of the arguments defined on the test definition.

```
1  select argument_name, data_type, data_length,  IN_OUT, sequence
2    from all_arguments
3   where data_level = 0
4     and ((owner = ? and package_name is null and object_name= ? )
5        or (owner = user and package_name = ? and object_name = ?))
6     and NVL(overload, 1) = NVL((SELECT overload FROM (
7                             SELECT overload, sum(tr) tr, sum(trf) trf FROM (
8                               SELECT overload, count(*) tr, 0 trf
9                                 FROM all_arguments
10                                WHERE data_level = 0
11                                  AND package_name = ?
12                                  AND object_name  = ?
13                                GROUP BY overload
14                                 UNION ALL
15                                SELECT overload, 0 tr, count(*) trf
16                                  FROM all_arguments
17                                 WHERE data_level = 0
18                                   AND package_name = ?
19                                   AND object_name  = ?
20                                   AND NVL(argument_name, 'RETURN_NAME') IN ( " + params
                                         + ")
21                                GROUP BY overload)
22                          GROUP BY overload)
23                          WHERE tr-trf = 0
24                          AND trf = " + n_args + "), 1);
```

Listing 5.8: Extract of the modified DbFit query to get the arguments of the function

Finally, the retrieved results were the expected ones.

The bulk of the changes were made on the main query, which we discussed several times during this section. In order to provide the necessary information for the query, it was necessary to

	PACKAGE_NAME	OBJECT_NAME	ARGUMENT_NAME	DATA_TYPE	DATA_LENGTH	IN_OUT	SEQUENCE	OVERLOAD
1	TW_EXP	T_DEPARTMENT	DEPARTMENT_ID	NUMBER	22	IN	2	1
2	TW_EXP	T_DEPARTMENT	(null)	VARCHAR2	(null)	OUT	1	1

Figure 5.7: Output for the third approach on changing the original DbFit query

perform changes on the underlying code of the DbFit project. We built a logic structure to support the passage of this data, such as the arguments name and their number. The changes affect Oracle database specific parts. They also spread to the code specific to the other databases supported by DbFit. These further changes are the result of the code refactor on the DbFit core on which the aforementioned parts depend on.

5.5.2 Results

The *Overload* does not occur frequently, however one of the functions on the GPAG module enabled us to identify this problem. Despite it being an isolated example, it called our attention since this can happen in future modules which will be tested using FitNesse. As we can see on figure 5.8, the occurrence of the *Overload* issue can be seen on, approximately 20% of all the Sysnovare functions.

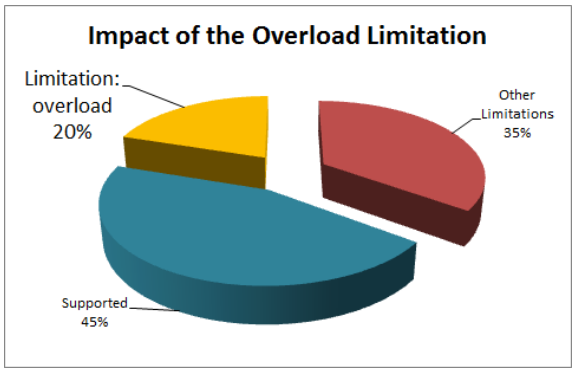

Figure 5.8: The impact of the Overload limitation

Moreover, without solving this constraint the other possible approach would be defining a development rule establishing that functions or procedures with the same name would not be allowed. This alternative seemed to be very restrictive. Therefore, we opt to improve DbFit with major new features.

5.6 The integration of FitNesse

On chapter 3, we mentioned that FIT, together with FitNesse and DbFit, results on a powerful database unit testing tool [Adz08]. To make this tool more attractive we determined the enhancement of some key qualities. Thus, we devoted resources to the development of some features we

found important and improve others that were in some way lacking.

But beyond all these, it was necessary to find a way to easily integrate FitNesse on the already working applications. Part of the success of the integration of this tool relies on the first impression its users have. It is the first impact which leads to a chain reaction on all the team members [dSFS+13]. The more well received, the better for the integration of FitNesse.

After defining a first group of fifty-five tests for the GPAG module, it was notorious how laborious this was. To overcome this drawback of starting using a new tool, we created a PL/SQL function - `create_tests_for_package` - which defines the tables' structure for each procedure or function included on a package. The user can invoke this function for a package to which he wants to define the tests. As an output this function returns a file with the structure of the test definition: (i) the call to the function, (ii) its arguments, (iii) the spaces to fill in the values for the inputs and (iv) the spaces for the outputs.

The monotonous task of defining the tests' structure is therefore more expedite. The definition of the data to use on the tests is left for the user. This is in fact a very time consuming activity. It requires commitment from all the elements taking part on it. Therefore, the fact of removing the laborious task of defining the tests' structure, aims at sparing the concentration to perform the other part of the tests' definition task which highly influences the quality of the testing results.

An important piece on this process of integrating FitNesse at Sysnovare includes introducing this component on the GP tool. The test files are generated for packages of business logic which belong to project's modules. Therefore, the same relation is made for the testing files: each of these belong to a project module. Including the testing files on GP allows controlling their versions.

The version control is crucial. It allows keeping an history of the changes, but it also enables having versions of the tests related with projects' versions. This relation between test versions and project versions is useful in order to know which tests were implemented at the moment a project's version was released.

However, this functionality is beyond the scope of this work and needs to be further explored. The final goal would be: at the same time a version of a project is installed, the automatic tests should run. Achieving this goal means having a complete automatic installation process. Nowadays, installing code is already a prompt one-click activity. We want to expand the concept of installing a version without reverting the current automated process. To achieve this it is still necessary to work on the automation of running the automatic API tests.

Finally, every tool comes with a user manual. FitNesse is not an exception. Therefore, we have prepared thorough documentation so that the FitNesse user can be as independent and well succeeded as possible. The documentation describes the necessary steps to achieve a final goal: run tests for a module. It includes initial tasks of setting up the tool, generating the testing file and how to run the tests.

5.7 Conclusion

During the last five sections we described the work developed to successfully achieve one of the objectives of this project: integrate a testing tool on the testing process which is mandatory to effectively test the API layer.

To achieve this goal, it was necessary to explore DbFit to be aware of its constraints. On a further moment, it was also mandatory to know DbFit in detail so we could develop new features for this open source tool. Not only have we contributed for a tool which can be used by other Oracle users, but we have also enriched DbFit. We can now we can successfully put in practice the testing process activity of using FitNesse to test Sysnovare applications. We have been describing the results obtained from the improvements we have applied. Now we have to expand the FitNesse usage to other modules.

Despite we were able to solve some of DbFit constraints, there is still work being developed and work planned for future improvements to the tool. DbFit still has some limitations, but as they have a minor impact (approximately 20%, as it is possible to see on figure 5.9) on testing Synovare applications we have prioritized them to be implemented after this first group of solved constraints.

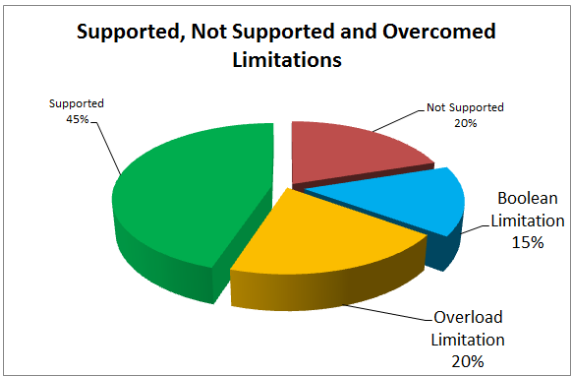

Figure 5.9: Supported, not supported and solved limitations

As an example, we plan on deeply exploring the wrong results we are getting while testing PL/SQL functions which receive parameters of type DATE. We also plan on analysing the behaviour of DbFit when business error messages are raised by the functions being tested. Hopefully, these new improvements will be launched on the following release of DbFit as it happened with some of the constraints we have solved which already integrated the last version of DbFit, 2.0.0 RC5, released on the 25*th* of May 2013 (announcement available at http://blog.quickpeople.co.uk/2013/05/25/release-of-dbfit-2-0-0-rc5/).

Chapter 6

GUI Test Automation

In this chapter we describe the work developed at the GUI test level. In a context of web applications, testing user interfaces is as important as testing the business logic. As a matter of fact, normally, the defects of applications are first identified on the interfaces. In fact, it is at this level that the typical end user, the non-technical user, mainly interacts with the application. Furthermore, the acceptance of a release is, in the majority of the times, decided based on a few clicks on the application's interface.

However, as it has already been discussed throughout this document, this test strategy is not the most effective nor reliable. It is hard to assure that a user who tests an application, of extensive dimension, by manually covering the interfaces will not miss a button, a menu or any other feature allowed by the application. This can have an undesirable impact when the decision of launching the application to the public is made based on the performance of these tests. It can happen that, in the middle of countless interfaces of the application, a faulty interface may suffice to result on a bug reported by the client.

As a solution to enhance the performance and quality of the interface tests, we have ventured on the integration of a web testing tool: Selenium.

This tool, introduced on chapter 3, offers automating web applications execution for testing purposes [Adz09]. Defining a test using Selenium requires recording a simulation of an interaction of a user with the application. The person who simulates it, has to perform the actions of a every day user of the application. At the same time, the group of actions being executed are registered by Selenium in the form of commands, as it was shown on chapter 3.

This group of commands is later used by Selenium to autonomously replay the interaction simulated during the test definition.

Unfortunately, there are commands, registered by Selenium, that during the replay of the tests are not correctly represented, unless they are edited by the user. This scenario is not suitable when we want non-technical users defining the tests. Therefore, our work at the GUI level focused on

the removal of these obstacles which negatively impact on the autonomy of a non-technical user defining the tests.

6.1 Wait for element command

After defining an experimental test suite, with the aim of becoming acquainted with Selenium, the result was a series of time-out errors. The test started with a simple click on a button. When replaying the test, immediately after the `click` command, there was the time-out error. This error was related with the html element on which the `click` command should be executed.

6.1.1 Solution

A first approach to this problem was finding, on the available Selenium commands, one suitable to indicate the javascript concept of waiting for an element. In fact, it was possible to add a `waitForElementPresent` command prior to the `click` command.

However, this solution had two drawbacks. (i) The insert of a `waitForElementPresent` command, prior to each `click` potentially doubles the number of commands on a test suite. This, results on a hard to read test suite, where the task of finding a command easily results on a complex task. (ii) There was also the possibility of setting a rule that prior to a `click` command, it was mandatory to insert a `waitForElementPresent` command. Even though this could be instituted, it would be an unfortunate solution regarding the Selenium's philosophy. Its main appeal is being able to define a test just by interacting with interfaces. Manually adding a command on the Selenium IDE test suite is the opposite idea. Moreover, this operation is not doable for a non-technical user, someone who is not familiar with javascript concepts. On the context of this work, where both the tester, who has no technical knowledge, and, in a forthcoming future, the client, would not be able to autonomously define the interface tests.

Not fully satisfied with this solution we looked for another alternative. After some research we found a plugin worth trying: the *ImplicitWait*. This is a plugin already suggested by Selenium which aims at automatically waiting until an element is found, before executing the following commands [Sel04]. It is available as a firefox plugin and, visually, results on the addition of a hourglass icon on the Selenium IDE interface. This icon allows activating the *ImplicitWait* option.

6.1.2 Results

With the integration of the *ImplicitWait* plugin, it was possible to reduce the number of commands from 621 to 503. These results are from a test suite with eighteen test cases which totals 621 commands. Notably, this test suite is now easier to read and, most important, it is easier to update.

This decrease of 118 commands, which had to be manually introduced, also has a high correlation with the time necessary to define a test suite. Since there are no pauses on the automatic test capture, the test definition requires less time effort when using the *ImplicitWait* plugin. Reducing

the time necessary to define a test is one of the objectives of this project, since we aim at achieving a testing process which does not require a lot of time from the participants involved on it.

6.2 Images with mouse over

Throughout the user interfaces of Sysnovare applications there is a considerable number of actions available through the *click* on images only visible with the *over* action. While trying to define tests which included clickable elements with a previous action of *mouseOver*, when replaying the test it would fail. The test was failing because the *mouseOver* commands were not being included on the test case actions, as it is illustrated on figure 6.1. Without the sequel that *mouseOver* has on the element properties, Selenium was not able to identify the element, using the same locator it was used during the definition of the test. In fact, the element locator is only valid after the outcome of *mouseOver* action.

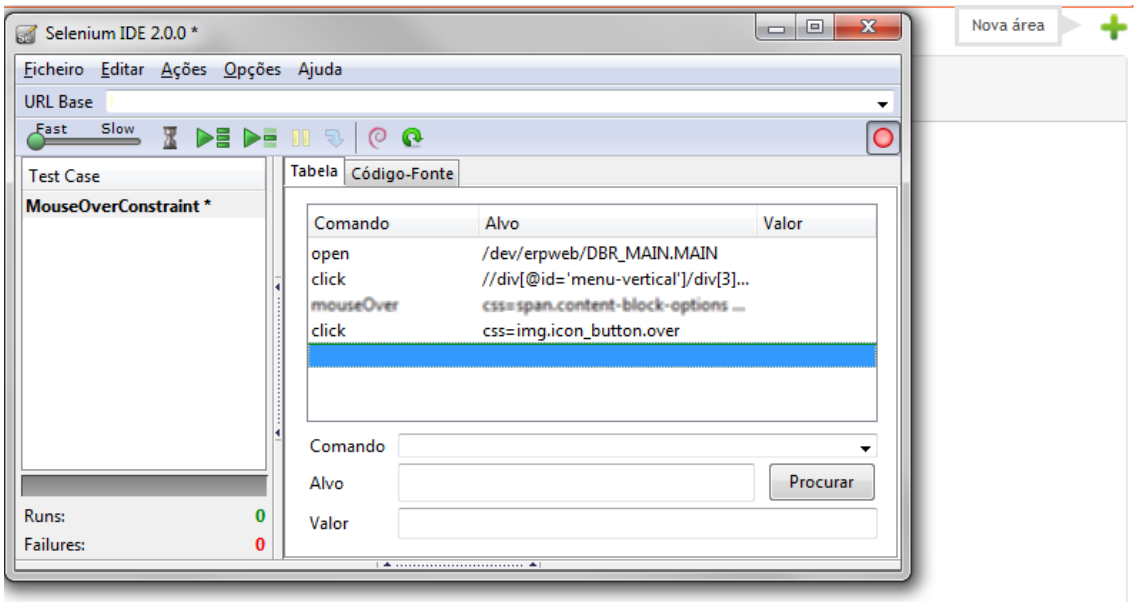

Figure 6.1: Selenium clickable images with *mouseOver*

Manually, it was possible to avoid this issue: the person defining the test would remove the *.over* from the *Locator css=img.icon_button.over*. However, besides only a user with know-how on this area being capable of performing this solution, this is not the purpose of Selenium-IDE. Furthermore, this patching up task is actually rather specific and different from case to case, thus it being impractical.

6.2.1 Solution

The solution for this constraint was to add the *mouseOver* command prior to the *click* command. To implement this solution we changed the original Selenium handler for the *click* command and also created a new handler for the *mouseOver* command.

Previously, all the *mouseOver* actions were being captured but ignored and did not figure on the list of the registered commands. So, exploring this opportunity, we changed the *click* handler adding a verification to decide if the *mouseOver* command should be included on the list of the test case commands. As it is shown in listing 6.1 we verify whether the *Locator* for the *click* command is the same used on the previous *mouseOver* command on that same element. When the *Locators* are different, it indicates that the *mouseOver* command is needed to carry out some modification to the HTML element, so it will have other attributes to be used to identify it on the *click* command.

```
1  Recorder.addEventHandler('clickLocator', 'click', function(event) {
2      if (true){
3        var clickable = this.findClickableElement(event.target);
4        if (clickable) {
5                  if (this.mouseoverLocator &&  (this.mouseoverLocator.toString() !=
                       this.findLocators(event.target).toString())) {
6                      this.record('mouseOver', this.mouseoverLocator, '');
7                      delete this.mouseoverLocator;
8                  }
9                  this.record("click", this.findLocators(event.target), '');
10             } else {
11                 var target = event.target;
12                 this.callIfMeaningfulEvent(function() {
13                     });
14             }
15     }
16   }, { capture: true });
```

Listing 6.1: Selenium click handler modified

6.2.2 Results

The need for this development resulted on a feedback given by a user who, each time he tried to perform the action *Nova área* displayed on figure 6.1, while replaying the test, it would fail.

On the module where this issue was detected, there were another four similar situations. Without the sequel the *mouseOver* has on the element properties it was not possible to successfully replay the tests. Even though, with this improvement we only affect five test cases, in a total of eighteen, avoiding manually changing five commands, it is important to mention that this is an issue no user will be confronted with again. Therefore, the user will not exponentially increase the time of defining the tests, since he will not need to search for a reason why the error is happening, nor trying to find a solution for it.

Being so, even though we do not present a significant number showing the improvements brought by this solution, solving the mouse over issue has side effects on the forthcoming test definitions.

6.3 Html autocomplete elements

HTML autocomplete are elements vastly used on Sysnovare interfaces especially in forms. They aim at helping users inserting data and also reducing the number of errors due to incorrectly inserted data. These elements are simple inputs allowing the type of text and at the same time offering suggestions to the user.

The constraint around this element is the *Locator* used by Selenium to identify the element and register it for a future test replay. Selenium, by default, identifies the autocomplete using its *id* attribute, as the last line of the exemplification of a test definition on figure 6.2 shows. Unfortunately, its *id*, as it is suggested by the HTML5 specification, is unique among all the IDs in the element's home subtree [(W312]. To assure the compliance of this rule, the *id* of the autocomplete elements include a randomly generated number. This results on the same element with different *ids* every time a page is loaded. Therefore, Selenium is not capable of finding the autocomplete element when replaying the test.

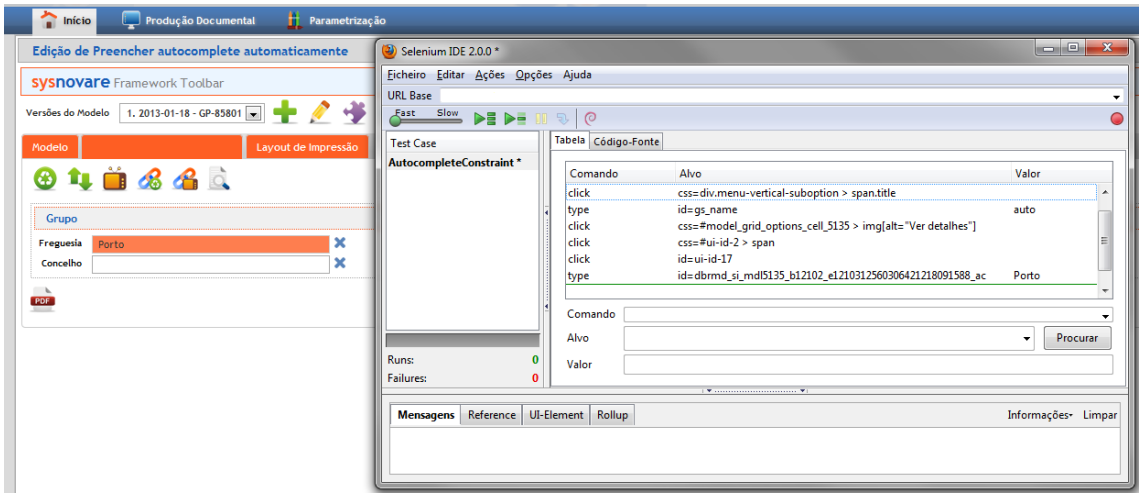

Figure 6.2: Selenium and the HTML autocomplete element

The *id* is not the only available *Locator* on Selenium. However, as it is shown on figure 6.3, Sysnovare autocomplete elements only have the *id* attribute. It was by identifying this constraint and analysing the autocomplete element that we choose the solution described next.

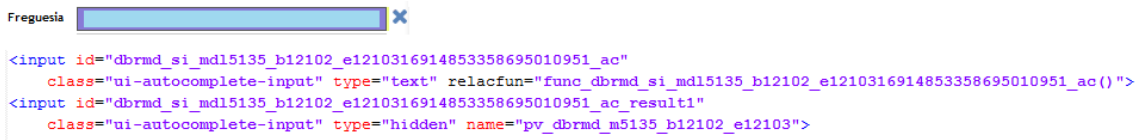

Figure 6.3: HTML autocomplete attributes

6.3.1 Solution

As the autocomplete element has no other attributes besides the *id*, we had to look for an alternative.

Choosing the *xpath Locator* was an alternative. However, we believe this was not the best alternative considering that this is an attribute which varies with the element's position on the page. Even though changing an element's position is not very frequent, we wanted a more robust solution.

Therefore, we analysed the autocomplete element in more detail and we found another approach based on another element related to the autocomplete. It is an *input* of type *hidden*. Besides the *id*, which is the same as the autocomplete element, this hidden element also has the *name* attribute (figure 6.3).

This element was the solution to identify the autocomplete element. To implement this solution, we had to edit the procedure used to get a *Locator* for an element. As the 6.2 code snippet shows, we check if the element is an autocomplete and we indicate that the *name* attribute of the hidden element with the same *parent* as the autocomplete, are the element and the *Locator* to use to identify the autocomplete.

```
1  if (e.hasAttribute("class")) {
2    e_class = e.getAttribute("class");
3    //Checking if the element is an Autocomplete
4    if (e_class.search("autocomplete") != -1) {
5      //Setting the Locator as the Name
6      i = 3;
7      //Getting all the siblings of the Autocomplete Element
8      var childNodes = e.parentNode.childNodes;
9      //For all the siblings of the Autocomplete Element
10     for (var j = 0; j < childNodes.length; j++) {
11         //Checking if its attribute type is of type hidden
12         if (childNodes[j].type == "hidden") {
13         //If so, it is the element we want to use and to apply the locator defined
               by setting i to 3
14             e = childNodes[j];
15             break;
16         }
17     }
18   }
19   }
```

Listing 6.2: Extract from the procedure which chooses the Locator

With this approach, we assure that any autocomplete element will not be identified by its *id* attribute or any other *Locator*. They will be identified by the *name* attribute of the *hidden* element which id is the same every time a test is replayed.

6.3.2 Results

The module we used to start integrating Selenium-IDE on Sysnovare, is a module which enables creating as many layout documents as the user wants. Normally, these documents are forms, which include various types of input elements. One of the inputs extensively used is the autocomplete element. It is by knowing how these elements are extensively used that we focused our work on solving this issue.

Even though, at this moment, the test suite only includes a document layout with two autocomplete elements, the need of configuring more documents with autocomplete elements is certainly going to increase. Later, creating and, specially, replaying the tests for these documents will not be a problem again.

6.4 Mouse right click

While defining the interface tests we faced another issue which required our attention: not registering the right hand click. On the applications being tested with the aid of Selenium-IDE, there are various interfaces where, to access certain functionalities, it is necessary to use the right hand click. This is a possible control over the user interfaces widely used on the tested applications. As an example, we do have a few trees which include context menus to access a group of options, as we can see on figure 6.4. Therefore, it was urgent to overcome this issue.

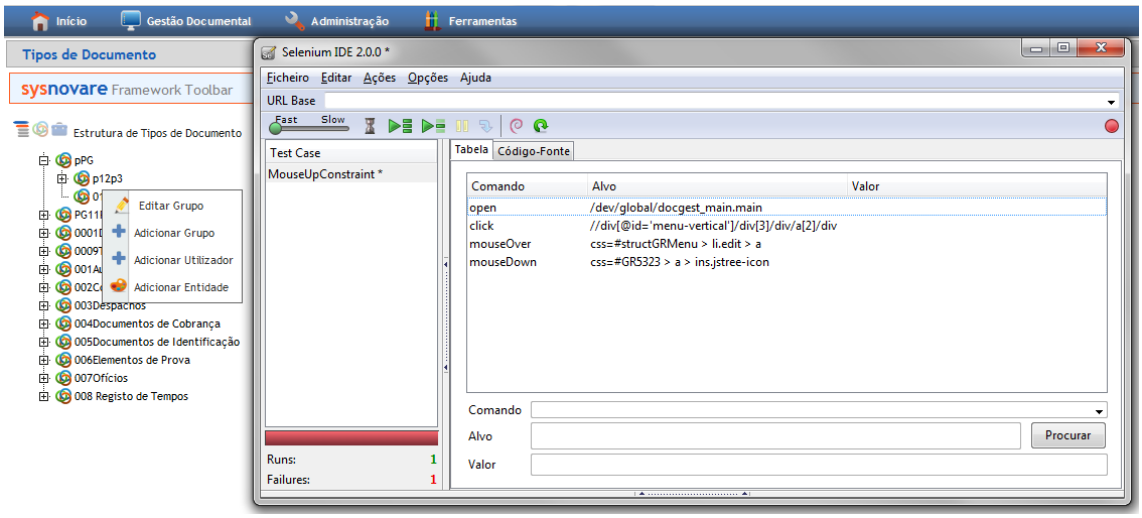

Figure 6.4: Mouse right click actions and test capture

The right hand click is composed of two actions: mouse down and mouse up. For the test replay to work, it was necessary to manually add a `mouse_up` command. Only the `mouse_down` command was being captured during the test definition.

6.4.1 Solution

The desired solution should also be capable of adding the, until then manually inserted, command `mouse_up`.

Once again, we started by searching for an existing solution to overcome this issue. We found a plugin, despite not being suggested on the Selenium project, which aims at recording all the mouse events: `RecordMouseDownUp.js`. This plugin registers all the mouse clicks: `up` and `down`. It registers both for the right and left buttons.

However, we slightly changed the plugin since there is no reason for registering the `mouse_up` for the left button, since this action is included on the `click` command. So, we changed the plugin to answer our needs of registering the right `mouse_up` action (listing 6.3).

```
1 Recorder.addEventHandler('mouseUps', 'mouseup', function (event) {
2   if (event.button == 2) {
3     this.record("mouseUpRight", this.findLocators(event.target), '');
4   }
5 }
```

Listing 6.3: Mouse Up handler

6.4.2 Results

This improvement on Selenium-IDE is specially noticed when accessing the context menus. Throughout Sysnovare applications these type of menus are widely used, thus this improvement being very useful. However, in what concerns the modules already being tested with Selenium and the interfaces which already have defined tests, this feature, of the right mouse click, was only used on the example given on figure 6.4.

Even thought, right now, it is only an interface using the right mouse click, this improvement brings benefits for future test definitions. The user who defined the test case for the interface on figure 6.4 spent more six hours from what he had planned trying to define the test case. This loss of time, which does not positively contribute in favour of using Selenium-IDE, will not be repeated, at least in what concerns the mouse right click.

6.5 Conclusion

This chapter described the work developed at the GUI testing layer with the aim of fulfilling the objective of having non technical users defining the GUI tests.

It was by receiving feedback from the users who started using Selenium that we worked on the four issues described above, which were not randomly chosen. At Sysnovare we had people responsible for defining the tests at the GUI layer, for two modules of a Sysnovare product. These two modules were chosen due to the high number of interfaces they have. They also include a huge

variety of interface elements, which allowed verifying the behaviour of Selenium in a significant variety of contexts. Therefore, based on the difficulties the users were finding, we developed our work. We were given specific examples of the problems they were facing with and our goal was finding a workaround for each problem. This resulted on the work described during this chapter: regarding the problems reported we were able to find a solution.

We aim at continuing the work at this level, according to the feedback we continue receiving. There are still a high number of interfaces without defined tests, a high number of projects also with no defined tests, therefore, we are expecting to continue having inputs that will require more work to be done on Selenium-IDE.

Chapter 7

Conclusions and Future Work

Now that we finished exposing how this project started, which were the first steps taken to approach the project and how the project was developed we end with a reflection on the work developed, how we achieved the objectives proposed at the beginning of the project and we also mention which are going to be the future developments for this project.

7.1 Overall Reflection

This project started with an initiative launched by Sysnovare with the important aim of improving its testing process so that they can achieve better results in this area.

Accepting the challenge, we started with a raw testing process to which we were able to positively contribute and improve it. The testing subject is itself a challenge. It is an extensive area, with various different aspects regarding a large set of topics. Being able to participate on the integration of a testing process on an enterprise with already a considerable number of habits and maturity, increases the level of the challenge.

Therefore, at the beginning of the project, we set an objective, split into smaller achievements. We stated that we proposed ourselves to systematize and partially automate Sysnovare testing process, with the aim of improving its effectiveness, efficiency and accessibility. The effectiveness was possible to achieve by having a testing process less dependable on manually performed tasks. The testing process is more efficient in what concerns the time which is required to test a product. This achievement was only possible thanks to the improvements made on the tools which now take part on the testing process. Finally, the accessibility goal was achieved by building a testing process where the non-technical participants have instruments which help them in performing their activities. So, we consider that this objective was ultimately achieved and we have done it by conquering the smaller objectives which we showed throughout the chapters which described the results of our work.

First of all, we defined a coherent and detailed testing process. This process has clarified activities and rules. A process ready to be put into practice, since, besides its rules, it is capable of being adjusted to the forthcoming necessities.

Secondly, we worked in two directions to automate the testing process and one of them was testing the API layer of the applications. At this level, we were able to integrate a testing tool, FitNesse, which enables running previously defined tests for the business rules without requiring programming effort. To integrate this tool, and since it is an open source tool, we were able to develop new features and improve the existing ones. With this work we improved FitNesse features in what concerns the Oracle support. Besides benefiting the worldwide community using it, this also benefits Sysnovare, since now it is possible to test more of the existing business rules with FitNesse.

Lastly, and since our work has a strong web context, we have also integrated an automated component to test the user interfaces. This activity is performed with the help of Selenium-IDE. To achieve the objective of having a non-technical user defining the tests, we had to improve some of the Selenium features to adapt to Sysnovare needs and to take the most advantage from its capture and replay concept. This consisted mainly on improving support for complex and dynamic web pages typically found on modern web sites.

7.2 Future Work

One of the ideas mentioned on this document to characterize the software testing area is that enterprises are eager to implement a fruitful testing process, that is, with reliable results and without consuming a lot of development time.

Despite the results achieved, which contribute to the desired characteristic of a testing process, there is still a long way to go to achieve a level of excellence in what concerns the automated testing process. The path forward includes work at the two layers of testing: API and GUI.

In what concerns the API layer, it is still necessary to further contribute to the DbFit project: (i) data types are still not possible to include on the tests and it is an issue which still has to be analysed; (ii) FitNesse behaviour with business error messages has to be discussed; (iii) it is not possible to test functions returning table types; (iv) functions using business types are also not possible to test. At least, there are these four constraints of FitNesse which require investigation and development work if we want this tool to be an indispensable tool to test any Oracle application.

Regarding the GUI layer, similar to what was done with FitNesse, the improvements on Selenium, an open source tool, ideally, should also be added to its project. Until now, managing the merge of our modifications and new releases of Selenium versions has not been a problem. But this is a task which can easily escalate to something difficult to manage.

It is still necessary to take the most advantage of the work developed during this project. This means that we would like to have more modules using FitNesse. The same applies to the user interface tests which still have a significant number of interfaces without defined tests.

In what concerns integrating FitNesse, until now, as mentioned before, we have done it with two modules. Using FitNesse does not involve solely generating the tests' structure. The activity which requires more work is the data generation. The two modules used and described on chapter 5 were very independent: were almost completely isolated from other modules. They did not require data from other modules. However, now we are working on integrating three more modules with FitNesse. Unfortunately, one of them is not so independent, requiring data from other two modules still not integrated with FitNesse.

By sharing these two thoughts, we aim at demonstrating that the testing process still requires study on how to manage the test data: how to introduce automatism into this activity and how to overcome the dependencies issue. When a solution for these two challenges is found, we believe it will constitute an excellent contribution to the automation of the testing process.

References

[Adz08] Gojko Adzic. Dbfit: Test-driven database development, October 2008. Available at `http://gojko.net/fitnesse/dbfit/`.

[Adz09] Gojko Adzic. *Test Driven . NET Development with FitNesse*. Neuri Limited, Second edition, 2009.

[Agr05] C. Agrawal, S. Barclay. Database pl/sql user's guide and reference 10g. Technical report, ORACLE, June 2005.

[BCH+11] Bob Brumfield, Geoff Cox, David Hill, Brian Noyes, and Michael Puleio. *Larger Cover Developer's Guide to Microsoft Prism 4: Building Modular MVVM Applications with Windows Presentation Foundation and Microsoft Silverlight*. Microsoft Press, 1st edition, 2011.

[dSFS+13] Fabio Q.B. da Silva, A. César C. França, Marcos Suassuna, Leila M.R. de Sousa Mariz, Isabella Rossiley, Regina C.G. de Miranda, Tatiana B. Gouveia, Cleviton V.F. Monteiro, Evisson Lucena, Elisa S.F. Cardozo, and Edval Espindola. Team building criteria in software projects: A mix-method replicated study. *Information and Software Technology*, 55(7):1316 – 1340, 2013.

[EMT05] H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(3):226–237, 2005.

[Feu96] Steven Feuerstein. *Advanced Oracle PL/SQL Programming with Packages*. O'Reilly, 1st edition, 1996.

[Feu13] S. Feuerstein. Wrap your code in a neat package. *Oracle Magazine*, 2013.

[Fla04] J Flack. MVC Development in PL/SQL, 2004.

[IEE04] IEEE Computer Society. *Software Engineering Body of Knowledge (SWEBOK)*. Angela Burgess, 2004.

[Itk12] J. Itkonen. Test-driven lecturing. In *Proceedings - 12th Koli Calling International Conference on Computing Education Research, Koli Calling 2012*, pages 141–142, 2012.

[LJ04] L. Lindstrom and R. Jeffries. Extreme programming and agile software development methodologies. *Information Systems Management*, 21(3):41–52, 2004.

[MH02] M. M. Müller and O. Hagner. Experiment about test-first programming. *IEE Proceedings: Software*, 149(5):131–136, 2002.

REFERENCES

[Nor06] Dan North. Dan north associates, 2006. Available at `http://dannorth.net/introducing-bdd/`, last access on February 2013.

[ORA11] ORACLE. Database jdbc developer's guide 11g. Technical report, ORACLE, 2011.

[Sei09] Peter Seibel. *Coders at Work: Reflections on the Craft of Programming*. Apress, First edition, 2009.

[Sel04] Selenium. Seleniumhq, 2004. Available at `http://seleniumhq.org`, last access on February 2013.

[SW11] C. Solís and X. Wang. A study of the characteristics of behaviour driven development. In *Proceedings - 37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011*, pages 383–387, 2011.

[SWD12] Mathias Soeken, Robert Wille, and Rolf Drechsler. Assisted Behavior Driven Development Using Natural Language Processing. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 269–287, 2012.

[VBCJ09] K Vlaanderen, S Brinkkemper, T Cheng, and S Jansen. Case Study Report : Agile Product Management at Planon, 2009.

[Vla03] K. Vlad. Applying MVC to Web-Based Applications with Generic Views, 2003.

[(W312] World Wide Web Consortium (W3C). Html5 a vocabulary and associated apis for html and xhtml, 2012. Available at `http://www.w3.org/TR/html5/`, last access on June 2013.

[WMT12] S. Wood, G. Michaelides, and C. Thomson. Successful extreme programming: Fidelity to the methodology or good teamworking? *Information and Software Technology*, 2012.