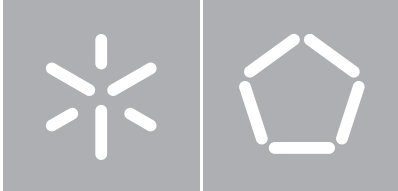


Universidade do Minho
Escola de Engenharia

Ricardo Gomes da Fonseca

Test Automation Framework



Universidade do Minho

Escola de Engenharia

Departamento de Informática

Ricardo Gomes da Fonseca

Test Automation Framework

Dissertação de Mestrado

Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor João Alexandre Saraiva

Acknowledgements

I would like to thank the following people for their help and support over the course of the completion of this thesis.

First of all, to João Saraiva my supervisor, for the continued support and engagement through this master thesis. Furthermore I would like to thank Carlos Argainha, my supervisor from Primavera BSS, for his technical support about software testing and for providing me with so much needed guidance. To both of them I will thank them for helping me grow as an individual and professional.

To Primavera BSS, for providing me the opportunity of working on exciting projects with a team of excellence.

To my co-workers, at Primavera BSS, for all their support, fellowship and sharing of knowledge.

To my parents, António and Maria, who were always there for my best interest even when the path I have chosen was not the one they wished for. I will always be grateful to them.

To my sweet sister, Joana, for her unconditional love and company even when not wanted.

To all my friends, who make life outside of work worth while and with whom I share the good things in life. To my surf friends, a special thanks, for sharing with me the passion for the sea and the waves.

Abstract

Primavera has invested a significant and costly man power in developing business-specific software solutions. Such solutions share a significant part of boilerplate code, namely the user interface. To minimize costs and, thus, improving it's software engineers productivity, Primavera BSS has invested many resources developing a Framework that allows for the next family of Primavera Products to be generated. The developed tool allows the Primavera Software Factory to easily adopt Software Development Processes based on Agile methodologies. The goal of this Dissertation is to add a new software component to this framework, a test automation component, that allows automated execution of tests to be performed on Products modelled on the Framework.

Resumo

A Primavera BSS investiu muitos recursos na criação de uma Framework que permitisse gerar a próxima família de Produtos Primavera. A ferramenta desenvolvida permite à Primavera Software Factory adoptar, mais facilmente, um Processo de Desenvolvimento baseado em metodologias Agile. O objectivo desta Dissertação é o de adicionar uma nova componente a esta Framework, uma componente de Testes Automáticos, que permita executar, de forma automática, testes aos Produtos resultado da modelação efectuada na Framework e se consiga assim que as soluções de teste do Departamento de Qualidade acompanhem Processos de Desenvolvimento ágeis.

Contents

1	Introduction	1
1.1	Overview	1
1.2	Automatic Testing for the Athena Framework	6
1.3	Research Questions	7
1.4	Document Structure	7
2	State of the art	9
2.1	Testing	9
2.1.1	What is Testing	9
2.1.2	Why is Testing Necessary	11
2.1.3	Test Types	11
2.1.4	Test Levels	12
2.1.4.1	Test Levels and the impact of Defects	14
2.1.5	Test Design Techniques	16
2.1.5.1	Black-box	16
2.1.5.2	White-box	17
2.1.6	Test Cases	18
2.1.7	Automated Testing	19
2.1.7.1	Data Driven	19
2.1.7.2	Keyword Driven	20
2.1.7.3	Model-based Testing	22

2.2	Athena Framework	25
2.2.1	Vision	25
2.2.2	Design Principles	26
2.2.3	Architecture	27
2.2.4	Products Structure	28
2.2.5	Developing	30
2.2.6	Domain-specific Languages	39
3	Development Tools	41
3.1	Software	41
3.1.1	Development	41
3.1.2	Testing	42
3.1.3	Management	42
3.2	Infrastructures	43
3.3	Others	43
4	Test Automation Component	45
4.1	Objectives and Motivation	45
4.2	Architecture	47
4.3	Excel Test Specification Generation	54
4.3.1	The Silverlight technology	54
4.3.2	XAML model	57
4.3.3	PRESENTATION model	60
4.3.4	Models Comparison	62
4.3.5	Data generation	63
4.4	Generic Execution Engine	68
4.5	The Validations component	70
4.6	The Reporting component	72
4.7	Excel Data management Tool	74

4.8	Business Scripts Execution	75
5	Case study: Business Suite	79
5.1	Application Description	79
5.2	Designing Tests	79
5.3	Executing Tests	85
5.4	Tests results	88
6	Concluding Remarks	93
6.1	Objectives satisfaction	93
6.2	Research Questions: Answers	95
6.3	Difficulties and challenges	96
6.4	Future work	97
	Bibliography	99
	APPENDIX	101
A	User interface view "Warehouse"	103
A.1	"Warehouse" view	104
B	Test specification excel of the view "Warehouse"	105
B.1	Controls sheet, "Types"	106
B.2	Main sheet, "Warehouse"	107
B.3	Validations sheet, "Validations"	110

List of Figures

1.1	V-model Software Development Process abstract diagram	2
1.2	Athena Framework High Level Architecture	3
1.3	Agile Software Development Process abstract diagram	4
1.4	Athena Framework Product Development abstract diagram	5
2.1	Relative Cost Factors of correcting Failures	15
2.2	Data Driven Script	20
2.3	Keyword driven Table 1	20
2.4	Keyword driven interpreter pseudo-code	21
2.5	keyword driven Table record representation	22
2.6	Model-based Testing workflow diagram	23
2.7	Athena's High Level Architecture	27
2.8	Visual Studio Athenas Framework Application Project	28
2.9	Visual Studio Athena's Framework Module Project	29
2.10	Athena Framework Products Structure	30
2.11	Module Structure	32
2.12	Entities Model Designer	33
2.13	User Interface Model Designer	34
2.14	Services Model Designer	35
2.15	Presentation Model Designer	36
2.16	Lists Model Designer	37
2.17	Product's landing page	38

2.18	Product's home page	39
2.19	Designers DSLs	40
4.1	Test Automation Framework - Components global overview	47
4.2	Test Execution Framework - Components global overview	49
4.3	Test specifications - Components global overview	50
4.4	Test execution environment - Components global overview	51
4.5	Reporting component - Components global overview	53
4.6	Login window of an Athena application(Silverlight technology)	55
4.7	UI Automation tree of figures 4.6 login window	56
4.8	Xaml files for the views of the module Purchases (Business Suite - MainLine)	58
4.9	CreditNote view's xaml specification (Business Suite - MainLine)	59
4.10	CreditNote view (Business Suite - MainLine)	59
4.11	Test specification generation diagram	63
4.12	AthenaTestSuite database (Database diagram)	65
4.13	Excel generation tool (User interface)	66
4.14	Generic execution engine diagram	69
4.15	Reporting component - Html report example	73
4.16	Data management tool (User interface)	74
4.17	Xaml Files for the Views of the module Purchases (Business Suite - MainLine)	77
5.1	Excel generation tool (BusinessSuite case study)	80
5.2	Data management tool (BusinessSuite case study)	81
5.3	BusinessSuite Brand entitie's view (BusinesSuite case study)	83
5.4	Test script for testing a Sale's business process (BusinessSuite case study)	84
5.5	Test execution tool (BusinessSuite case study)	86
5.6	Html script results report (BusinessSuite case study)	87
5.7	Solution coverage of the BusinessSuite application (BusinessSuite test re- sults)	88

5.8	Evolution of the solution's coverage of the BusinessSuite application (BusinessSuite test results)	89
5.9	Evolution of the number of executed test cases (BusinessSuite test results)	90
5.10	Passed and failed tests attending to the Service Releases dates (BusinessSuite test results)	90
5.11	Failed tests and discovered bugs (BusinessSuite test results)	91

List of Acronyms

ERP Enterprise Resource Planning

CMMI Capability Maturity Model Integration

SDP Software Development Process

QA Quality Assurance

ISTQB International Software Testing Qualifications Board

DSL Domain Specific Language

Chapter 1

Introduction

Summary

In this chapter, an overview of the problem covered by this dissertation and the document's structure will be given.

1.1 Overview

PRIMAVERA Business Software Solutions is a, 20 year old, multinational company that develops and commercializes management solutions and platforms for business process integration in a global market. The software developed by Primavera has a well-defined business logic and its most known and recognized product is an Enterprise Resource Planning (ERP) software [1]. An ERP software is an integrated computer-based system used to manage internal and external resources, including tangible assets, financial resources, materials, and human resources [2]. For being generic, Primavera's ERP is able to cover many business areas, such as: Accounting, Retail, Catering industry, Construction and Education, and cover many countries' legislation requirements, such as: Portugal, Spain, Angola and Mozambique.

The development process of Primavera's solutions is carried out in accordance with the Capability Maturity Model Integration (CMMI) Level 2 - Guidelines for Process Integration and Product Improvement [3]. The applications developed by Primavera followed a Software

Development Process (SDP) known as V-model [4].

Figure 1.1 illustrates a conceptual model of the SDP used at Primavera:

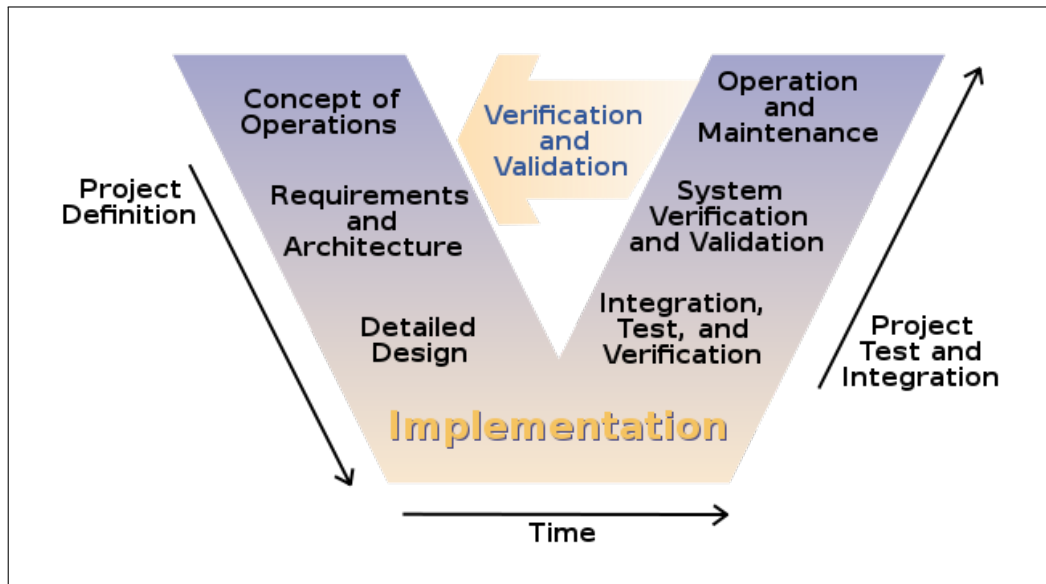


Figure 1.1: V-model Software Development Process abstract diagram

V-model means Verification and Validation model. Just like the waterfall [5] model, the V-Shaped life cycle is a sequential path of execution of processes. Each phase must be completed before the next phase begins. Testing of the product is planned in parallel with a corresponding phase of development [6].

Primavera noticed that the vast majority of its products had, as target, very similar business areas. This meant that much of the business logic was shared by the applications developed, however, the same development process was repeated for the creation of the applications. This meant that much of the SDPs were repeated without any reuse among different applications' with similar business logic.

To address this problem, to increase development's productivity and to improve the quality of applications developed, Primavera's developed the Athena Framework. This framework allows the creation of the next line of Primavera Products and allows the SDP to be more effective. The framework contains much of the business logic's that compose Primavera

1. Introduction

Products and allows a Business Specialist and/or Programmer to specify his application and respective services abstracting him from software architecture design and code details.

Figure 1.2 presents an High-Level architecture of the Athena Framework:

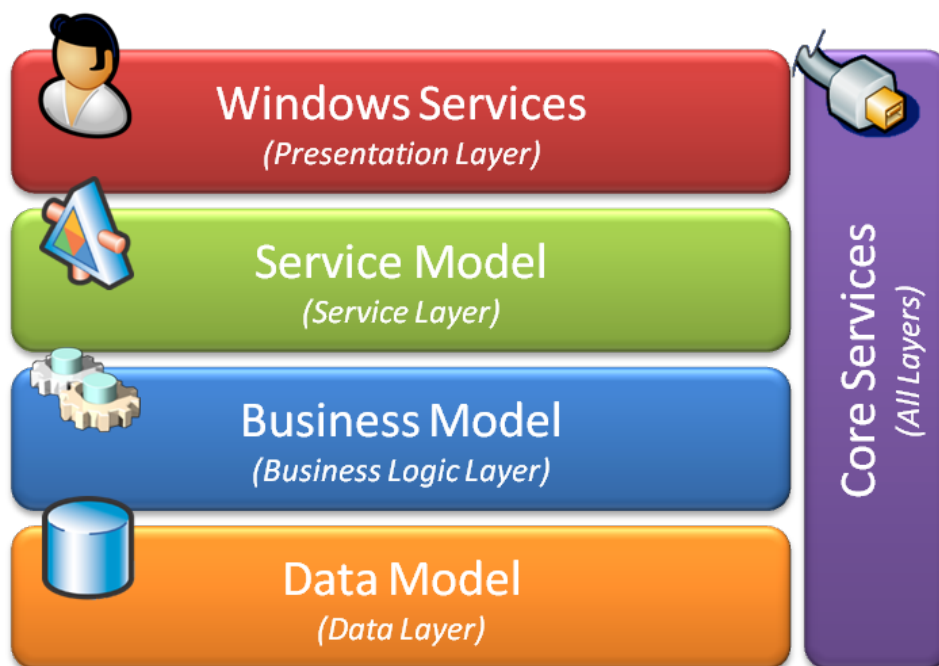


Figure 1.2: Athena Framework High Level architecture

The Athena Framework is a four layer architecture. It generates about 60% of the application's source code, data persistences structures and user interfaces. The framework will be presented in detail later in this document on Chapter 2 (State of the Art), Section 2.2 (Athena Framework).

Primavera has started a process to evolve the SDP to Agile [7] based methodologies. Figure 1.3 shows a conceptual model of an Agile based SDP:

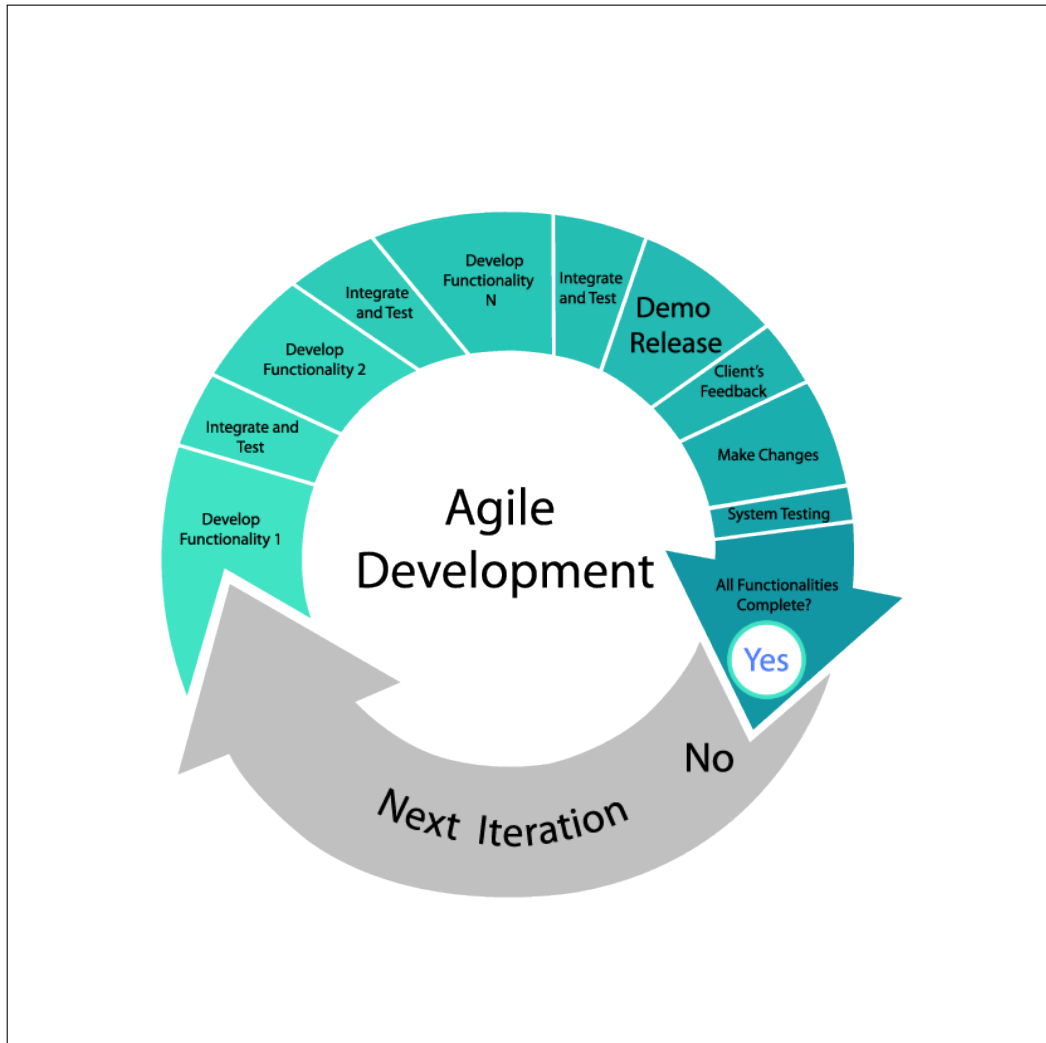


Figure 1.3: Agile Software Development Process abstract diagram

Agile software development is a group of software development methods based on iterative and incremental development. It promotes adaptive planning, evolutionary development and delivery, a time-boxed iterative approach, and encourages rapid and flexible response to change [8].

The Athena Framework allows an Agile development process, since for each Product there are several incremental iterations and, in each iteration, there's no need to start a new development process; the Business Specialist can simply start adding functionalities to the

1. Introduction

previous application iteration which results in an agile software development process.

Figure 1.4 shows an abstract diagram of the development process of a Product using the Athena Framework.

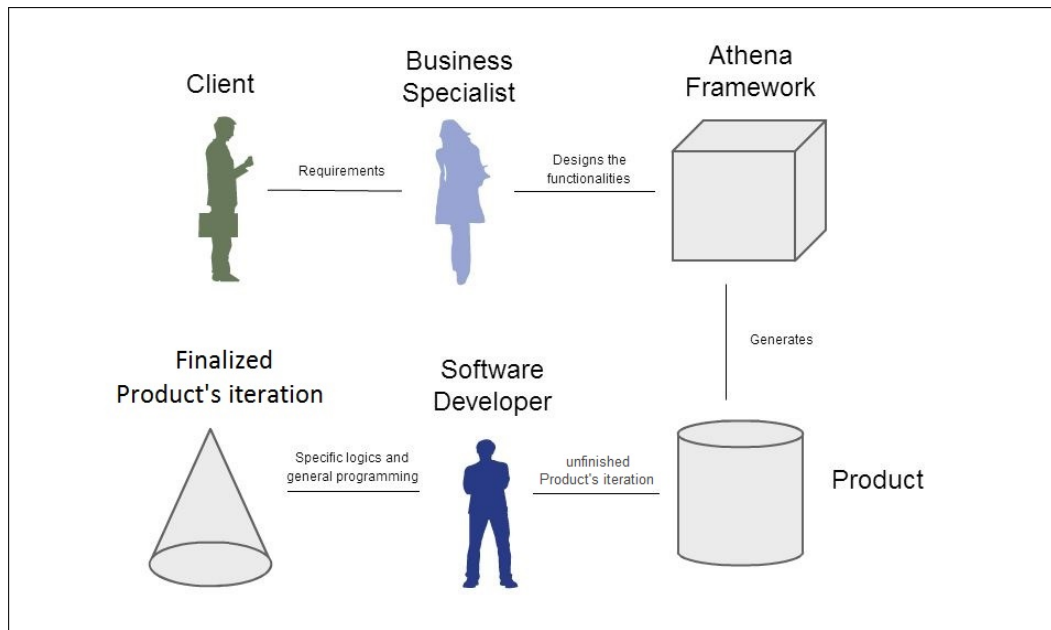


Figure 1.4: Athena Framework Product Development abstract diagram

Firstly, the Client approaches the Business Specialist. Together, they define the requirements which the business specialist will later analyse and record. Secondly, the business specialist implements the functionalities derived from the requirements by designing them on the Athena Framework designers. In most cases, the business specialist is also a software engineer but one of the features and advantages of the Athena Framework is that the person responsible for designing the application only has to be able to understand the business logic of the product that he is developing.

After the design phase ends, the Athena Framework will, automatically, generate most of the application's source code (about 60%). At this point, however, the application is not yet ready to be delivered to the Client because 40% of the application still has to be developed. The

40% includes specific business processes which, for their complexity or specificity, need to be designed and programmed later; it also includes specific services, deployment questions and data persistence improvements and optimizations. At this phase the person responsible for finishing the application are the programmers but the Business Specialist can also intervene.

After the application source code is completed, an application's iteration has been concluded; what is left is a potentially deployable Product which is ready and awaiting testing. The Quality Assurance (QA) Department of Primavera's Software Factory has the objective of ensuring that all Primavera products comply with the standards of Primavera's quality patterns and that they meet the customers' expectations. To ensure the functional quality of the software, two different but complementary types of testing are conducted: firstly, manual testing that is representative of the large part of the time and effort invested on testing, secondly, a set of very limited automatic tests of about 1.000.000(one Million) tests in each month.

1.2 Automatic Testing for the Athena Framework

The Agile development methodologies have as a main characteristic the quick change and the ability to adapt to that change. In this context, the Athena Framework allows for the SDP to be quicker and facilitates the evolution of the software, which allows for a fast adaptation to change and for it to be done with fewer costs.

However, the current test solutions of the QA Department can not keep up with the ability that the framework has to adapt to changes. It makes absolute sense to add a new test component, that allows code for testing to be automatically generated, to the Athena Framework, as the framework is able to do so for the software.

The work, which will be presented in this dissertation's document, is integrated with the Athena Framework as one of its components, and available to all applications developed

with the framework, since their creation, providing automated test capabilities in the earliest phases of the SDP. It has the capability of running tests on the application's user interface by simulating mouse and keyboard events in the application's views. The solution provides automation for 80% of the Athena applications' views, posing as a reliable tool to conduct Integration and Acceptance testing on the application.

The developed solution is able to keep up with the agile SDP and allows testers to reduce the number of manual tests done through the automation of the process, which results in greater, faster and more extensive testing, thus reducing costs and increasing the software quality.

1.3 Research Questions

In this thesis we aim at answering to the following three research questions:

- 1. Is it possible to automate graphical user interface software tests in Silverlight interfaces?**
- 2. Is it possible to automatically generate excel test specifications from the models of the Athena Framework?**
- 3. Will a model-based test solution present results that will prove it to be a reliable and effective tool, available to the Quality Assurance Department of Primavera Software Factory?**

1.4 Document Structure

This document is organized as follows:

Chapter 2 A state of the art review of some specific areas of software testing, code generation and the Primavera's BSS branch company name Primavera Software Factory is presented.

First the Athena Framework is presented and explained on the context what software testing is and what is it for. Then the Athena Framework is presented and explained on the context of the Primavera Software Factory.

Chapter 3 In this chapter it's given an overview of the different tools that will be available, at the time of this report, for conducting the development and deployment of the project.

Chapter 4 This chapter describes the expected results for the project as well as the main challenges that are expected and the project planning roadmap.

Chapter 5 In chapter 5 it's explained the decision-making process as well as the developed Test automation component for the Athena Framework.

Chapter 6 This chapter presents an overview of the real usage Tests conducted on the developed Solution as well the results of these Tests and the impact of the proposed Solution on the Organization.

Chapter 6 In this chapter the results of the developed work are discussed and the concluding remarks presented. It's also given an overview of what could be the Future Work of this project.

Chapter 2

State of the art

Summary

In this chapter a state of the art of software testing, as well as an overview of the Athena Framework will be presented. A framework developed by the company Technology - Primavera Software Factory, where the work of this dissertation is conducted, which is a subsidiary of the Primavera BSS company.

2.1 Testing

Before we survey the techniques used in software testing, let us clarify the notion of software testing [9] and why it is needed in software engineering [10]. The illustration of the test types and design techniques presented was based on [9]. The International Software Testing Qualifications Board (ISTQB) is a software testing qualification certification organization that operates internationally.

2.1.1 What is Testing

Testing consists on various activities that can take place from the beginning to the end of the software development life-cycle with the purpose of finding defects and identifying failures on the software being tested. Test activities include planning and control, choosing test conditions, designing and executing test cases, evaluating test results, reporting on the tests

2. State of the art

and target system as well as reviewing documents (example: source code) and conducting static analysis.

Static testing includes activities such as reviews, walkthroughs or inspections, that can and are often omitted. In contrast, dynamic testing consists in the actual execution of code that will test the overall software system or a specific module (or functionality) of the same system. In this later case, the testing techniques can be used to test a software sytem that is not fully implemented. In a best case scenario the testing techniques use a set of test cases that should cover (all) the software system. Much of the dynamic testing takes place along side with the development phases so that the problems can be identified as soon as possible.

Testing, however, is not only used to find problems/bugs on the software that is being tested. Testing can be used to give a notion (to the stakeholders) about the quality of the product on a given time, which can be useful, for example, when the software development method is an Agile method [7]. In this case there are multiple cumulative releases of the product, so the tests can give a notion on whether the iteration of the unfinished product has quality to be launched, or not. So, it can be said that the main objectives of testing are finding defects, gaining confidence about the level of quality of the software, providing information for decision-making, and preventing defects.

Some common misconceptions about testing are that testing consists only of running tests cases and that testing and debugging are the same activity. As stated before, testing consists on a large set of well defined activities that can take place along all the development cycle, while debugging is the activity of finding, analysing and removing the cause of software failures. This is a development activity, done by developers.

Testing, and particularly dynamic testing, can show failures that are caused by defects. Dynamic testing is carried out by testers, and a failure is identified by comparing the expected results of test cases to the actual results produced. If expected and actual results differ, then a failure occurred. Then, the developers carry out the debugging activity to find, analyse and resolve the reported defect. After this, the cycle of testing, reporting and debugging restarts.

2.1.2 Why is Testing Necessary

Software applications are changing the way people live. Nowadays there's software in almost every aspect of people's lives, being it communications, business applications, consumer products (i.e. cars, house apparel). The majority of this software is responsible for regular people to interact with cell phones, medical equipment and social networks. It is expectable that sometimes the software doesn't behaves correctly (i.e. as it was expected) which can lead to many problems like: loss of money, productivity, time, business reputation and in same cases even injury or death.

When discussing software defects it is important to distinguish between the root of the defect and the actual consequences of it.

Software is designed by human beings, and human beings can make mistakes that lead to defects (errors, bugs) in software. If a mistake is made when analysing, designing or encoding, then, when the software is executing, it might not behave as it should doing something that was not expected or not doing anything at all, resulting in a failure.

Defects are caused by Human beings, and some of the causes may be time constraints, excessive complexity or the different technologies. Failures can be caused by Environmental conditions like radiation, magnetism or even hardware malfunctioning.

2.1.3 Test Types

As mentioned in the last chapter, there are two types of testing: static testing and dynamic testing.

- **Static Testing:** Refers to the test activities that do not involve the execution of code, parts of the software or the entire software. The main activity are reviews: the purpose is to find defects rather than failures, and the final product of these reviews are report documents. There are four main types of reviews: Informal Reviews , Walkthroughs, Technical Reviews and Inspections.

Static testing is often performed by testing tools. Static testing tools include tools that analyse the program's source code [11], databases, spreadsheet data [12] or program models (like UML) of the design/implementation and generate reports with the intent of finding defects. The use of tools may be useful when the software dimensions are noticeable large and it is composed of a large collection of artifacts that in other way would be extremely complex and time consuming for humans to analyse. Since static testing is not on the scope of this dissertation, it will not be discussed in detail.

- **Dynamic Testing:** It involves the execution of source code, parts of the software or the entire software with the objective of finding failures, i.e. giving inputs to the software and analyse if the outputs were the expected ones.

In dynamic testing the software or some of its parts are actually compiled and executed. Some of the dynamic testing methodologies are Component Testing, Integration Testing, System Testing and Acceptance Testing that will be reviewed in detail in the next section on this document.

Even though dynamic and static testing are performed with complete different approaches, they are complementary and can be carried out in any of the phases of the SDP.

2.1.4 Test Levels

There are four levels of software testing [9]: Component, >> Integration, >> System, >> Acceptance.

- **Component Testing:** Also known as Unit testing. It aims at ensuring that each component/unit of the software being developed is valid, i.e. performs as designed/-expected.

The functionality of the individual software models, programs, objects and classes are tested in order to find defects. This testing method is done by developers, while they are developing these units/components (white-box style). Unit testing, by itself,

cannot validate an entire software but it can ensure that all blocks that compose the software work and perform as individually expected. Despite being a functional oriented test activity, it can also include tests to some non-functional characteristics of the software like robustness testing.

Finding a defect at this level of testing has a low impact on the development process since it can be immediately corrected by the developer and re-tested for ensuring that the same defect doesn't occur again.

- **Integration Testing:** Aims at searching for defects on the interaction of the different components/units of the software.

The interactions between the units, or large parts of the software system, are tested, as well as the environmental components like the operating system, file system and the interfaces between systems. This testing method is conducted by the Testers.

The greater the scope of the integration, the harder it is to isolate defects to a specific component, thus increasing the risk and cost on time for the Developers to find, analyse and fix the defect. In order to avoid this problem, integration testing shall be incremental, i.e. starting by testing interaction between the most atomic units/-components and systematically increase the size and scope, instead of an all-in-one approach.

It is obvious at this point that the system's architecture influences integration testing. In fact, if the designer, developer, and tester fully understand the software architecture, then the development process can be done in such a way that integration test is optimized and consequently more efficient.

- **System Testing:** The purpose of this test is to validate whether the software complies with the specified requirements or not.

A full system integration test is conducted by testers, to ensure that the software complies with the written functional and non-functional requirements, which are both of equal importance. The test environment should correspond to the target execution environment of the software, since any difference may result on defects not being

identified on the tests. The full system integration means that environment variables like the operating system, file system, software interoperability are also tested, as long as they are referenced on the Master Test Plan.

For this level of testing there must be well defined documentation on the Master Test Plan.

- **Acceptance Testing:** Acceptance testing establishes the system's compliance with the business requirements and whether it is acceptable for delivery or not.

The aim of this test is not to find defects on the functionalities of the software but rather to establish confidence, test specific non-functional characteristics (such as usability) and assess if the software can be deployed. It is conducted by stakeholders, mainly by the customers and the users of the system.

2.1.4.1 Test Levels and the impact of Defects

A study [13], conducted on 2002, showed that Software bugs or errors cost the U.S. economy around \$59.5 billion (USD) each year.

The study also found that, although all errors cannot be removed, more than a third of these costs, or an estimated \$22.2 billion (USD), could be eliminated by an improved testing infrastructure that enables earlier and more effective identification and removal of software defects.

In that study, a table shows the relative cost factors of correcting errors taking in consideration only where those errors are introduced and found in the SDP. Figure 2.1 shows that regardless of when an error is introduced it is always more expensive to fix it downstream in the development process.

2. State of the art

Where Errors are Introduced	Where Errors are Found				
	Requirements Gathering and Analysis/ Architectural Design	Coding/ Unit Test	Integration and Component/ RAISE System Test	Early Customer Feedback/Beta Test Programs	Post-product Release
Requirements Gathering and Analysis/ Architectural Design	1.0	5.0	10.0	15.0	30.0
Coding/Unit Test		1.0	10.0	20.0	30.0
Integration and Component/ RAISE System Test			1.0	10.0	20.0

Figure 2.1: Relative Cost Factors of correcting Failures, as a relation between in what phase the Failures were introduced and the estimated cost of correction depending on which phase they were found.

Each column on the table of the previous figure, represents a stage in the SDP, left to right from early to advanced stages. Each line also represents stages in the SDP, top to bottom from early to advanced stages.

If an error is introduced in the earliest of stages (Requirements stage) and found on that same stage, then the cost of correcting it is the factor one(1) and represents the smallest possible cost; this is, the best-case cenario.

On the other hand, if an error is introduced in the earliest of stages (Requirements stage) but is only found in the Post-product Release, then, the cost of correcting it is 30x (30 times) higher than the cost of finding and correcting it at the earliest stage (previous scenario). Finding an error that is introduced in the earliest of the stages only in the latest stages means that, possibly, all the stages of the SDP will need to be reviewed and corrected.

By analysing the figure, we conclude that testing needs to be conducted as soon as possible in all the stages of the SDP.

2.1.5 Test Design Techniques

The purpose of test design techniques is to identify test conditions and test scenarios through which effective and efficient test cases can be written.

2.1.5.1 Black-box

Black-box testing is a method of software testing whose goal is to analyse the functionalities of a software, regardless of what the software does internally.

This method assumes that the software (or parts of it) being tested is a black-box that receives an input and returns an output. It can be said that black-box testing evaluates the behaviour, i.e. output of the software being tested rather than its internal functions, procedures and interactions.

This test method is usually applied at every level of software testing, though it is more predominant in high-level testing.

Next, an overview of some of the main black-box design techniques is given:

- **Decision Table Testing**

A decision table as the purpose of modelling the logic of the software being tested. Tests done with decision table techniques may be more rigorous because those tables enforce logical rigour. The tables describe multiple scenarios, i.e., multiple combinations of actions that can be taken under multiple conditions.

- **All-Pairs Testing**

Also known as Pairwise Testing in Computer Science. It's a method of software testing that, for each pair of input parameters, tests all possible discrete combinations of those parameters. This method is used to generate the smallest possible vectors of input values that allow for a set of tests to be made to the software (or a part of it) and to ensure that it's enough for the test.

- Equivalence Partitioning

Equivalence partitioning is a technique for dividing test input data into partitions from which test cases can be derived.

It is the process of taking all possible test cases and placing them into classes. One test value is picked from each class while testing. Each equivalence data class is a partition of the original input data.

By using this technique it is expected that the test cases be designed to cover every partition at least once. This technique can also be used with the output values of the tests, i.e., defining equivalence sets of output data values from which the tester can infer the actual result of the test.

- Boundary Value Analysis

This technique aims at defining values for the tests' input variables that include representative boundary values.

It is usually used in conjunction with the equivalence partitioning technique since the boundary values are defined for each one of the equivalence sets of data defined.

This allows a reduction in the number of tests that are needed to test the software (or parts of it) and thus improve the testing activity.

2.1.5.2 White-box

White-box testing, also known as structural testing, is a method that aims at testing the internal functions, procedures, interactions of usually small parts of the software.

The scope of this method is not what the software does but how it does it instead. As a consequence, these methods can be better used, for example, in component testing, in which the developer tests the source code being written and can immediately correct errors in the source code implementation.

Despite being a method that aims at software implementation correctness, it cannot test unimplemented parts of it and it cannot test for defects on the requirements.

2.1.6 Test Cases

Test Cases are a set of conditions and/or variables that allow to test whether a software or parts of it are working correctly, or not.

There are two main types of test cases:

- **Formal Test Cases**

Formal test cases are those that are written against a given requirement. In [10] it is advocated that a requirement must always allow for a test to be written against it, otherwise the requirement wouldn't be verifiable.

So by writing test cases against the requirements, one may ensure whether the application does or does not what it is supposed to do, that is, what is defined in the requirements. Despite being truth, it does not ensure that the software is free of defects since it only validates the functionality defined by the requirement and not the requirement itself.

Formal test cases characteristics [9]:

- Test cases should be written to test only one thing at a time. They should not overlap or be complicated. Test cases should be 'atomic'.
- Test cases should cover both positive and negative scenarios.
- Accurate: Should have a well defined purpose.
- Traceable: Capable of being traced to requirements.
- Repeatable: Can be performed multiple times.
- Reusable: Should be able to be reused.

- **Informal Test Cases**

Informal test cases are written when a software, for some reason, does not have written requirements.

In this case, the testers write the test cases based on experience and/or look and

feel and the scope of the test cases are the accepted normal operations of software of similar classes.

2.1.7 Automated Testing

Automation helps to implement and verify best practices and organizational standards; it improves people's productivity and facilitates control of the software processes by collecting measurement data [14].

2.1.7.1 Data Driven

Data driven testing is a methodology where the data used for tests, i.e, the input values and conditions, the output values and conditions and the test environment variables and conditions can be changed by storing the variable values on tables, databases or excel files and executing the test with the data from these sources.

Data driven scripts are those application-specific scripts captured or manually coded in the automation' tools proprietary language and then modified to accommodate variable data [15].

In fact, these test scripts are nothing more then source code or specific application domain code that will actually perform the test, and that will get the data for the test to an external source, which means that the test data is not hard coded on the test script, code or program. This approach allows the reuse of the test scripts and programs.

Data Driven Features:

The data scripts contain hard coded string tokens that are used by the application that runs the script to execute the test. When this occurs, the scripts are easily broken when an application evolves [16]. The author [15] also gives an example of a data-driven script as follows:

2. State of the art

```
Image Click "DocumentTitle=Welcome;\;ImageIndex=1" "Coords=25,20"
```

Figure 2.2: Example of activating a server-side image map-link in a web application automation tool scripting language.

In Figure 2.2 the scenario of clicking on the image identified by his 'ImageIndex' on the page identified by the 'DocumentTitle' can be identified.

The image information is hard coded on the test script this means that this script may be broken if the title of the document or the index of the image change.

2.1.7.2 Keyword Driven

Keyword Driven testing, also known as Table Driven, is a technique to develop tests as data tables using a keyword vocabulary that is independent of the test automation tool used to execute them.

This technique allows the development of an application-independent framework but should also be suitable for manual testing.

Figure(2.3) shows a test case record defined according to this technique. The test case aims to verify the value of a user ID textbox on a login page.

WINDOW	COMPONENT	ACTION	EXPECTED VALUE
LoginPage	UserIDTextbox	VerifyValue	"MyUserID"

Figure 2.3: Keyword driven Table 1

The 3 main characteristics [15] that define a keyword-driven Automation Framework are:

- Reusable Code

2. State of the art

- Error Correction
- Synchronization

Figure(2.4) shows the pseudo-code that can be used to interpret the data record from the table of Figure 2.3.

Framework Pseudo-Code
<p>Primary Record Processor Module:</p> <pre>Verify "LoginPage" Exists. (Attempt recovery if not) Set focus to "LoginPage". Verify "UserIDTextbox" Exists. (Attempt recovery if not) Find "Type" of component "UserIDTextbox". (It is a Textbox) Call the module that processes ALL Textbox components.</pre> <p>Textbox Component Module:</p> <pre>Validate the action keyword "VerifyValue". Call the Textbox.VerifyValue function.</pre> <p>Textbox.VerifyValue Function:</p> <pre>Get the text stored in the "UserIDTextbox" Textbox. Compare the retrieved text to "MyUserID". Record our success or failure.</pre>

Figure 2.4: Keyword driven interpreter pseudo-code

The pseudo-code presented in Figure 2.4 executes as follows: The primary loop reads a record from the data table, performs some high-level validation on it, sets focus on the proper object for the instruction, and then routes the complete record to the appropriate component function for full processing.

Keyword driven design allows a vocabulary that can be processed by both man and machine.

In Figure 2.5 is presented a representation of the test case record from the table of

2. State of the art

Figure 2.3, that a human tester can have and allows the tester to design tests without knowing a thing about the automation tool used to execute them.

On the LoginPage, in the UserIDTextbox, Verify the Value is "MyUserID".
--

Figure 2.5: keyword driven Table record representation

2.1.7.3 Model-based Testing

Model-based testing [17] is the automatic generation of software test procedures, using models of the system requirements and behavior. It can also include executing artifacts to effectively perform software testing.

Some of the benefits of model-based testing are [18]:

- Shorter schedules, lower costs, and better quality;
- Capability to automatically generate many non-repetitive and useful tests;
- Test harness to automatically run generated tests;
- Eases the updating of test suites after requirements change;
- Capability to evaluate regression test suites.

There are several steps required to successfully introduce Model Based Testing into a project [19], and Figure 2.6 presents the workflow diagram for it:

2. State of the art

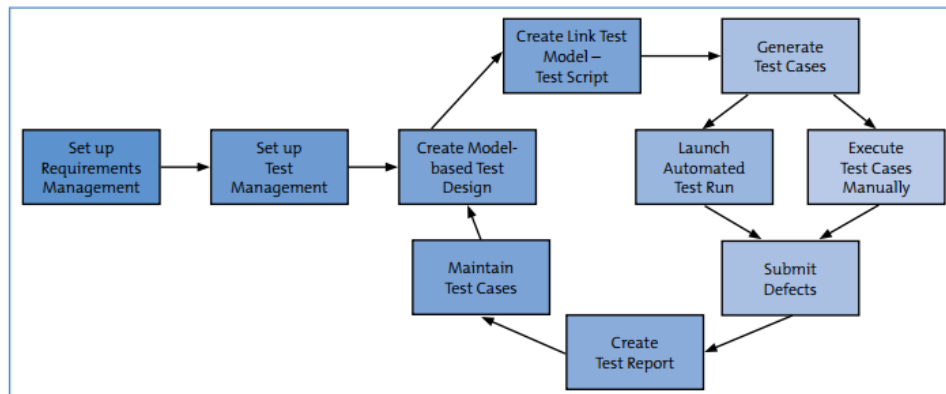


Figure 2.6: Model-based Testing workflow diagram.

[Set up requirements management] - A pre-requisite for model-based testing is that the requirements are detailed and clear enough so that a formal model can be derived from them. This phase also involves text experts in early review to avoid problems on the test process later.

[Set up test management] - Based on the requirements, a test structure is created consisting of test packages and test cases. Model-based testing may not be applicable to all requirements.

[Create model-based test design] - In this step it is important to have a proper integration with the test management tool for importing information about requirements and test packages. It is important to avoid reusing models, i.e. if the code is generated from a model then there will be no changes on the generated code if the model is reused.

It is important that the designed models cover both test data and test sequence.

[Create links between the test model and executable test script] - The generated tests must be linked to the model by assigning steps to GUI elements on the test definition or even creating scripts for each keyword of the test definition.

[Execute Test Cases] - The test execution tool should be fully integrated into the test

2. State of the art

management framework. It should be possible to start test runs and store the results automatically from within the test management tool.

In this step, corrections to the model are made by verifying that the tests run as expected and changing the model until the tests run smoothly.

[Submit Defects] - In cases of deviation from the expected behaviour, it has to be determined whether the problem was really caused by a bug in the software or conversely by a defect in the test model.

[Create test report] - The execution logs created during the test run have to be transformed into a concise test report that informs the responsible managers about the current project status. In this step, it is important to have a good integration with other tools.

[Maintain test cases] - One of the goals of model-based testing is reducing the maintainability cost of existing tests for new versions of the same application. For each new application's version that has to be tested, the tests have to be adapted and repeated. If the models are correctly defined, then a new application's version will only imply slight changes to the models and the re-generated test Cases will maintain their validity.

A variety of techniques/methods exists for expressing models of user/system behaviour (These techniques will not be detailed in this thesis since that is not the goal of the project):

- Decision Tables

Tables used to show sets of conditions and the actions resulting of them.

- Finite State Machines

A computational model consisting of a finite number of states and transitions between those same states, possibly with accompanying actions. An example of this technique is used by the GuiSurfer tool. It extracts finite state machines, describing the GUI, from source code [11].

2. State of the art

- Grammars
Describe the syntax of programming and other input languages.
- Markov Chains (Markov process)
A discrete, stochastic process in which the probability on where the process is in a given state at a certain time depends only on the value of the immediately preceding state
- Statecharts
Behaviour diagrams specified as part of the Unified Modelling Language (UML). A statechart depicts the states that a system or component can assume, and shows the events or circumstances that cause or result from a change from one state to another.
- ClassSheets
ClassSheets [20] are a formal model to define the business logic of spreadsheet data. In [20] techniques to infer such models from spreadsheet data is presented. Such models are then used in the MDSheet(FALTA CITAR) tool.

2.2 Athena Framework

PrimaveraBSS invested many resources in creating a framework that allows the creation of the next line of Primavera products and allows the SDP to be more effective. This resulted in the creation of the Athena Framework. It allows the programmer to specify his application and respective services. The Framework generates some of the application code as well as data persistence structures and user interfaces.

2.2.1 Vision

The Main Goals of the framework are:

- Support the Development of the next generation of Primavera's Products
- Increase Development Productivity
- Increase Product Quality

2.2.2 Design Principles

- Declarative Programming
Most of the Framework behaviour can be configured, customized and modified through XML-based configuration files, which allows the customization of these behaviors without recompiling the software components.
- Design Patterns
The architecture implements known design patterns and best practices.
- Code reuse
The re-utilization of third-party libraries to implement parts of the architecture design can boost productivity.
- Service orientation
This design orientation promotes independence between components and reduces the communication's verbosity.
- Process orientation
The Framework's core is designed to support operations "guided" by business processes and workflows.
- Extensibility
All parts of the Framework are designed with the goal of being easily extended and customized.
- Technology independence
The framework's components must be technology independent as much as possible.

Such feature will allow the replacement of part of the architecture with minimal or no impact on the overall design.

2.2.3 Architecture

The High-level Architecture of the Athena Framework is divided according to the following logic layers:

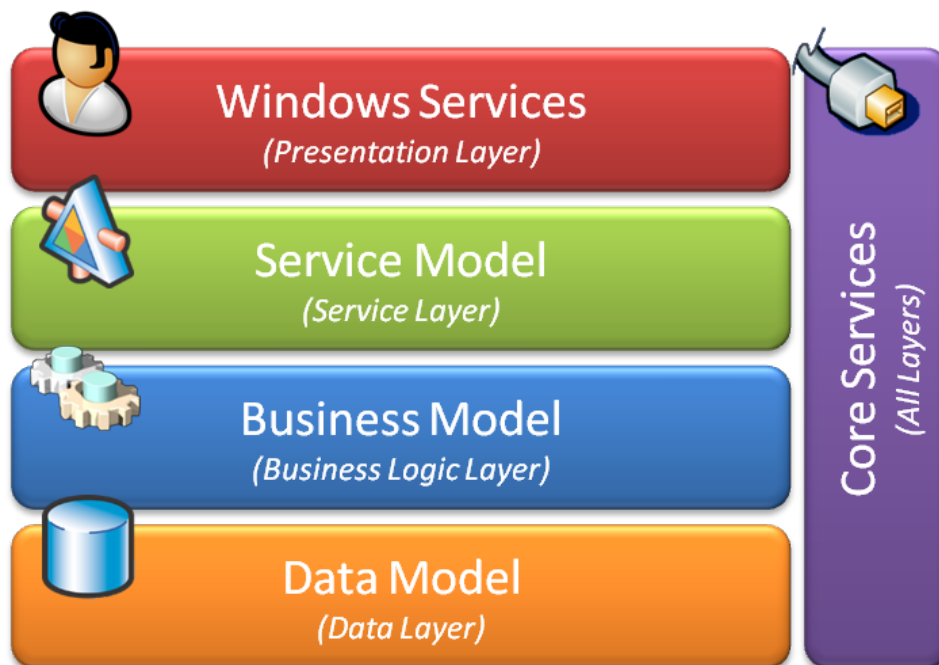


Figure 2.7: Logic Layers that compose the Athena Framework High Level architecture

Thus, the Athena Framework is a four layer architecture, where the Presentation layer deals with the user interface and user experience. The Service layer deals with services that are used for the presentation layer(client) to communicate with the lower layers(server). Underneath the presentation layer, there's the Business Logic layer, that handles processes that compose the logic of the application. Finally, the Data layer deals with data representation and persistence.

2.2.4 Products Structure

Structure of a Product Developed with the Athena Framework.

The development environment for generating a Product with this framework is the Visual Studio IDE Tool. Athena Applications Structure:

The Products developed with the Athena Framework are composed of two components: Application and Modules as it can be seen on figure 2.8 and 2.9.

Application is the project that aggregates all Modules Projects. Modules are projects that represent logical/business components of the product being created as shown on figure 2.10.

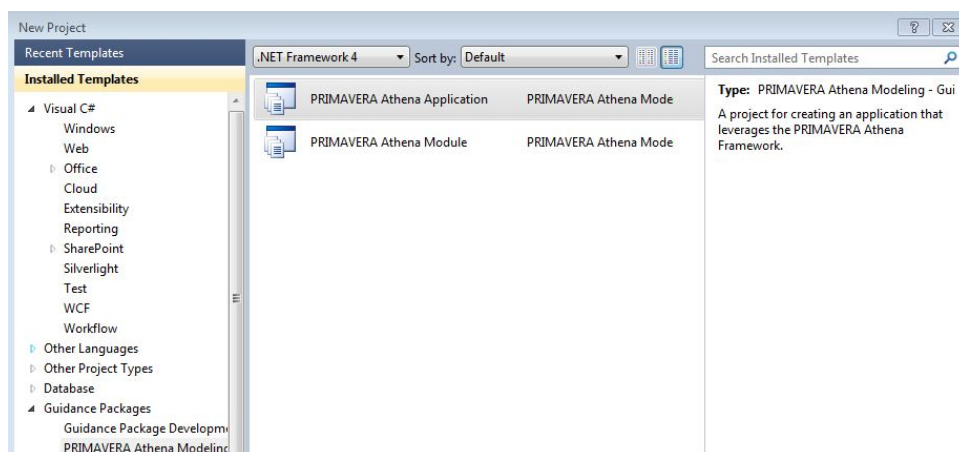


Figure 2.8: Project type Application, available with the Athena Framework integrated with the Visual Studio IDE(version 10.0.40219.1 SP1 Rel)

2. State of the art

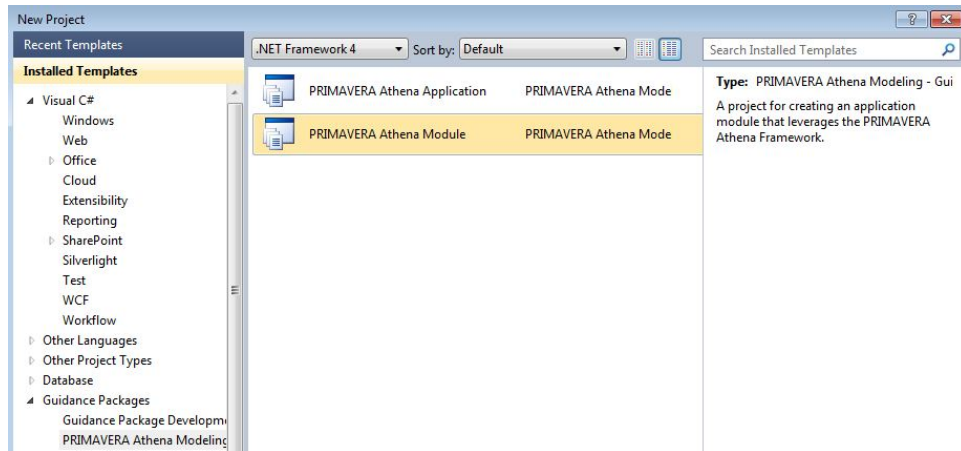


Figure 2.9: Project type Module, available with the Athena Framework integrated with the Visual Studio IDE(version 10.0.40219.1 SP1 Rel)

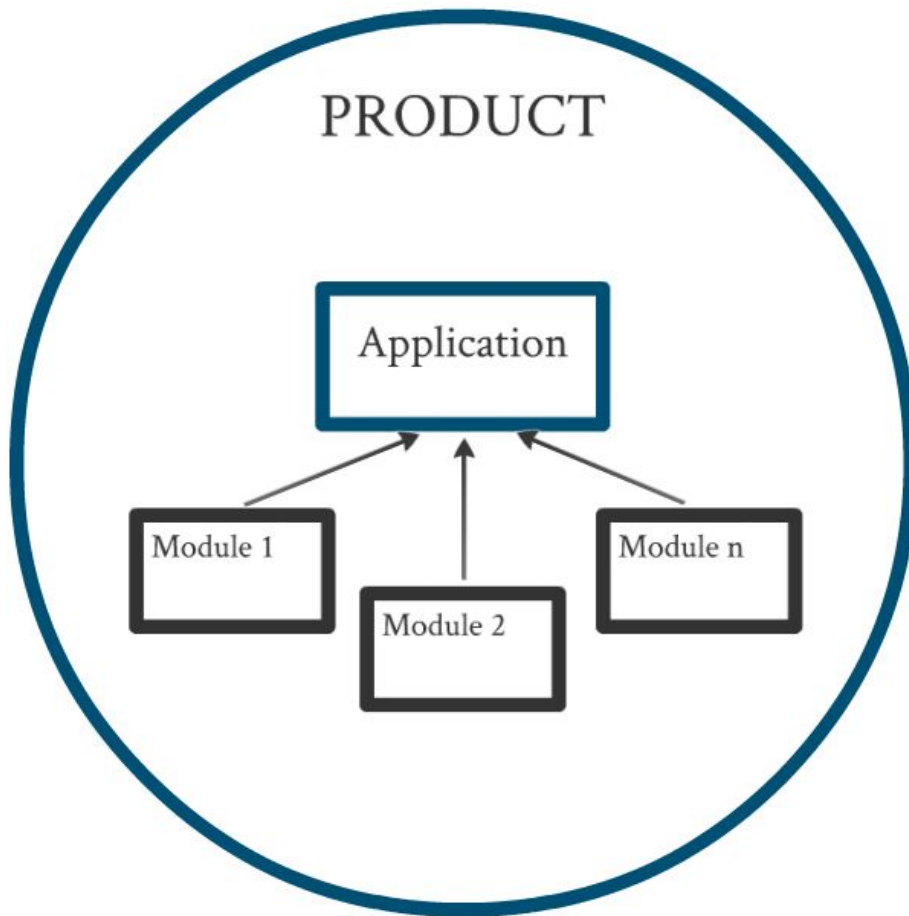


Figure 2.10: Diagram of the structure of a Product developed on the Athena Framework

2.2.5 Developing

Developing a Product with the Athena Framework.

Developing a Product with the Athena Framework is easy. After the Application's project has been created, Module's projects that express business components can be created or added to the Application's project, since the framework allows a modular and incremental component reuse as a software development approach. On the next example the designs that can be used to modulate the business logic of the created Module named "Materials" will be shown.

1. Module Structure

2. State of the art

A Module is composed by several Visual Studio projects that were automatically generated on the Module creation.

These projects are aggregated by folders(Fig. 2.11):

- 'Host'
It aggregates the project with the configuration for running the module.
- 'Client'
Holds the projects that will allow the application's presentation to be displayed on the client side.
- 'Models'
It's composed of two projects: the project that generates all database structures needed to support the business logic of the module, and the project with the designers.
- 'Server'
A set of projects that hold the configurations to treat the server side of this particular module.
- 'Tests'
The projects used to test the Module at a variety of levels.

2. State of the art

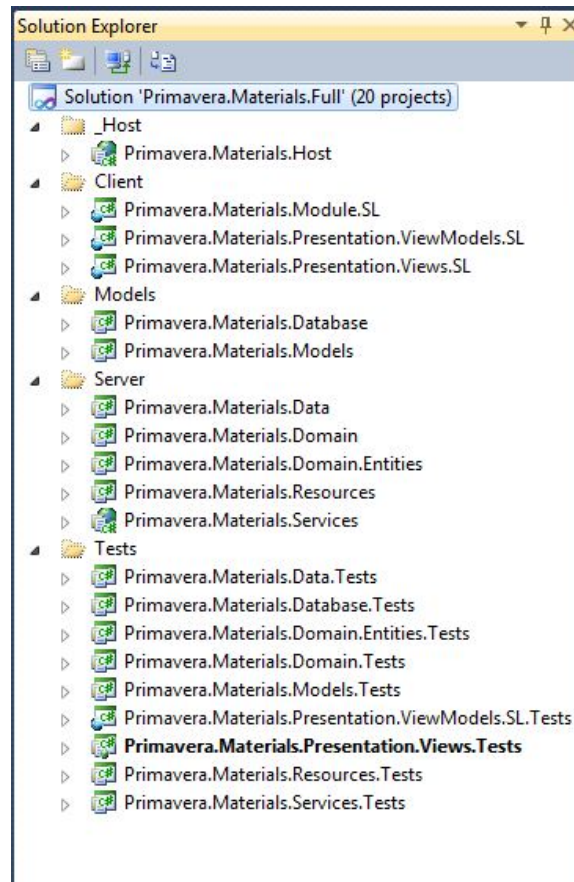


Figure 2.11: Structure of a Module 'Solution' on Visual Studio IDE.

2. Entities Model Designer

The Entities Model designer allows the programmer to design the entities of this module, that is the business entities.

2. State of the art

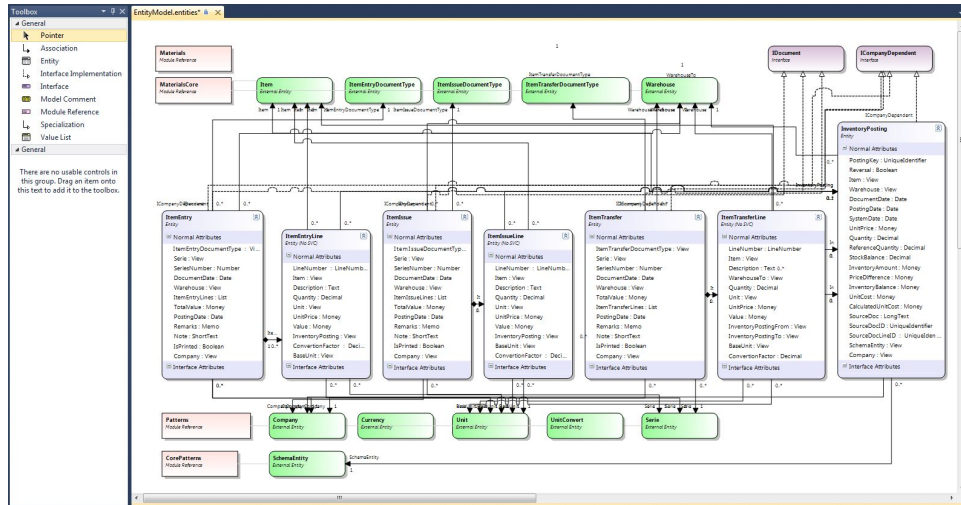


Figure 2.12: Entities Model Designer and ToolBox of the Module 'Materials' on the Visual Studio IDE(version 10.0.40219.1 SP1 Rel).

The most important artifacts are described as follows:

The light-brown boxes(Module Reference) allow the referentiation of other modules of the product.

The green boxes(External Entity) are references to entities of other modules in the product, models that have the corresponding module reference box created.

The blue boxes(Entity) are the actual entities of this module that can be created, as well as their attributes, and can have relations between the other entities (whether they are referenced or not).

3. User Interface Model Designer

The Interface Model designer allows the programmer to create views over the entities that he desires and also allows the default view to be edited, i.e. allows the programmer to choose what's the view structure, attributes fields, and properties about them.

2. State of the art

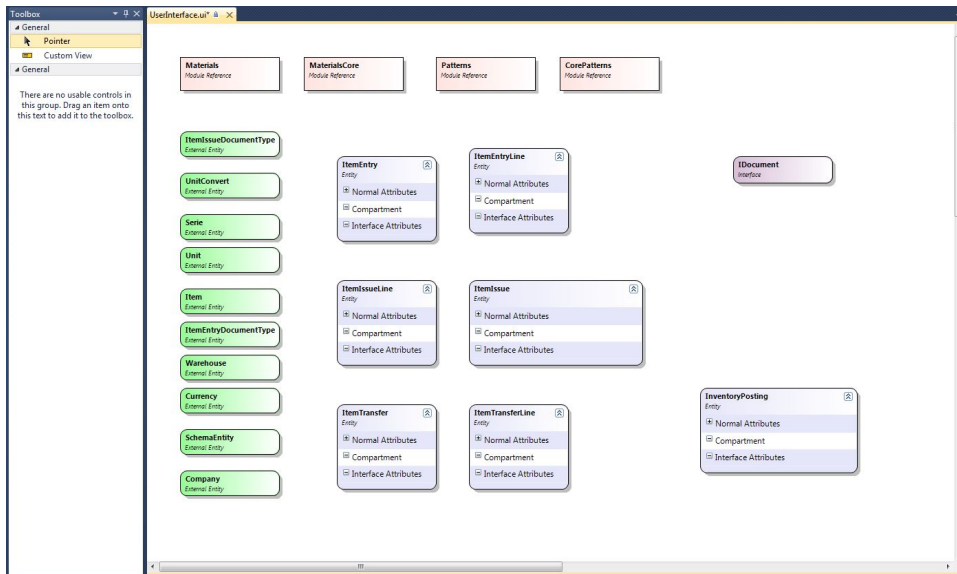


Figure 2.13: User Interface Model Designer and ToolBox of the Module 'Materials' on the Visual Studio IDE(version 10.0.40219.1 SP1 Rel).

The yellow box (Custom View) gives the possibility to define custom views for any entity if the programmer desires to do so.

4. Services Model Designer

The Service Model designer is the tool used to define the services available for any entity.

These Services that can be defined are the create, update and delete operations, so it allows for the CRUD services.

2. State of the art

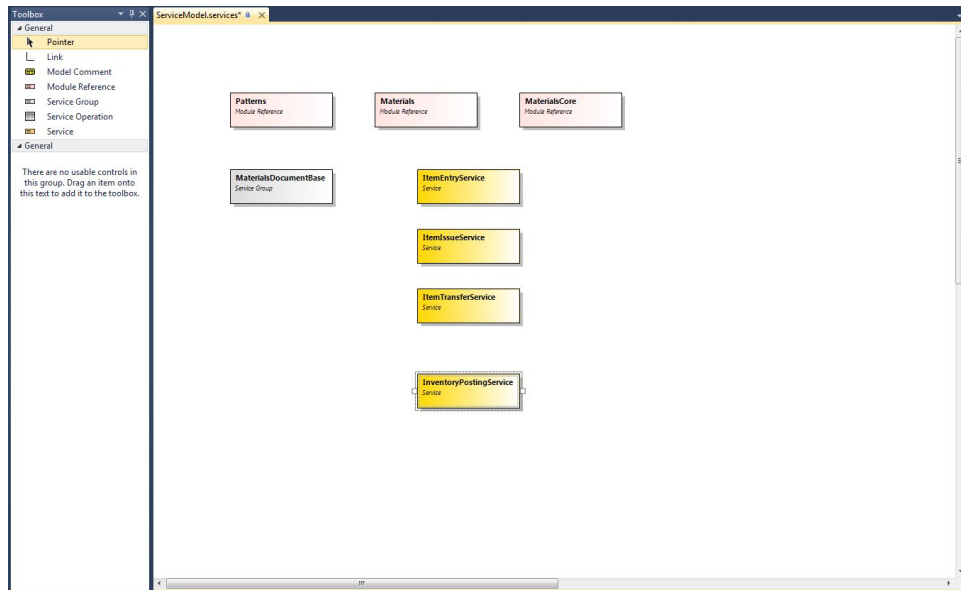


Figure 2.14: Services Model Designer and ToolBox of the Module 'Materials' on the Visual Studio IDE(version 10.0.40219.1 SP1 Rel).

5. Presentation Model Designer

In this designer the programmer defines Presentations or Reporting Views for the desired Views of the Entities.

2. State of the art

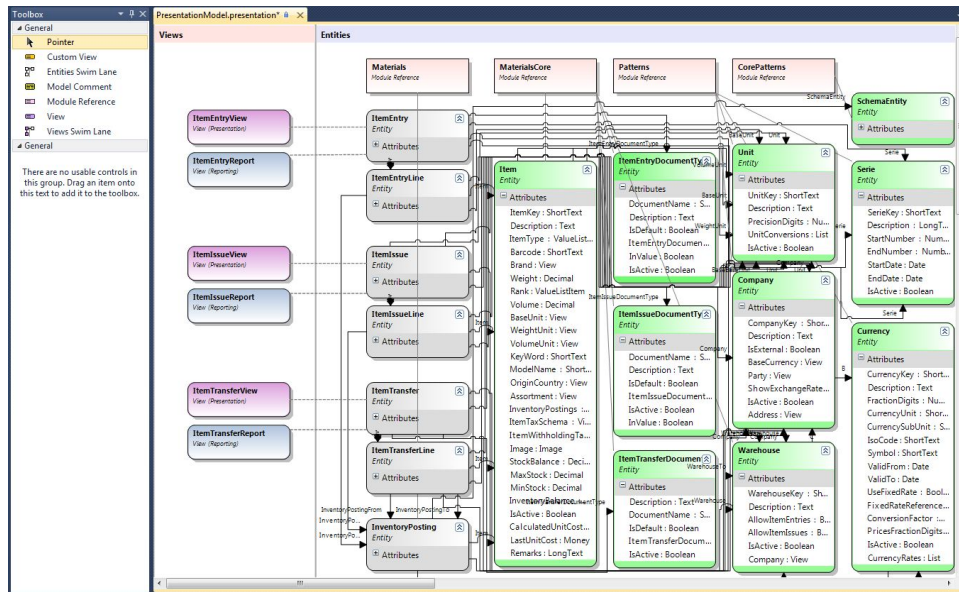


Figure 2.15: Presentation Model Designer and ToolBox of the Module 'Materials' on the Visual Studio IDE(version 10.0.40219.1 SP1 Rel).

6. Lists Model Designer

It allows the programmer to create perspectives over the entities, i.e. the definition of specific views where only the selected attributes of the entity are shown. It can be seen as the definition of a preview view over the list of registers of the entity on the database.

2. State of the art

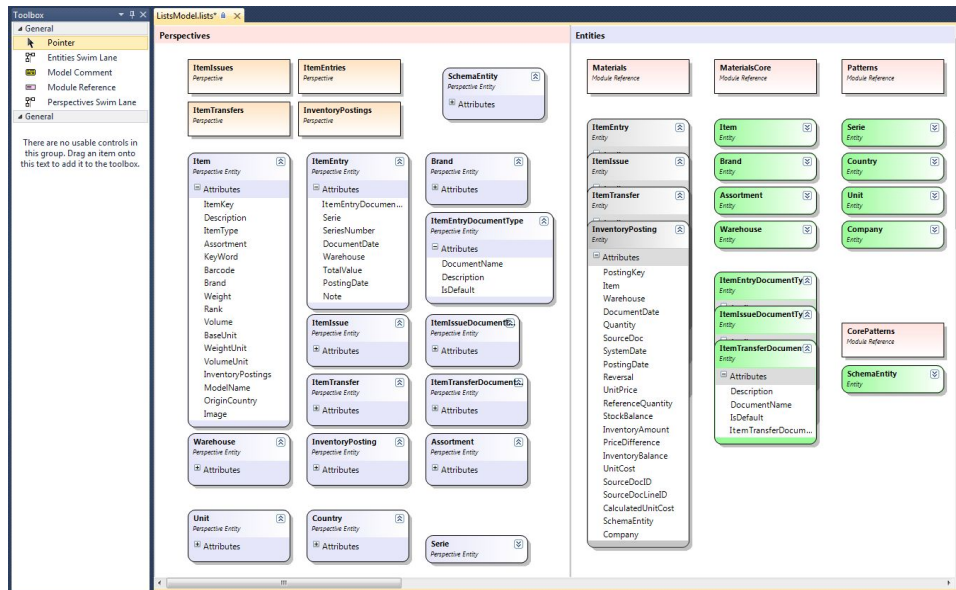


Figure 2.16: Lists Model Designer and ToolBox of the Module 'Materials' on the Visual Studio IDE(version 10.0.40219.1 SP1 Rel).

7. Result Product

After the modulation as been done the modules can be compiled, as well as the application.

The product is finally created and when it's compiled and executed, the framework, with the modulation and models of the designers, generates all the host and server services and configurations, all the business logic for the modelled entities, as well as the presentation layer for them and the data persistence.

The application can, then, be accessed through the use of a web browser and a preview of the final product can be seen in the figures 2.17 and 2.18.

2. State of the art

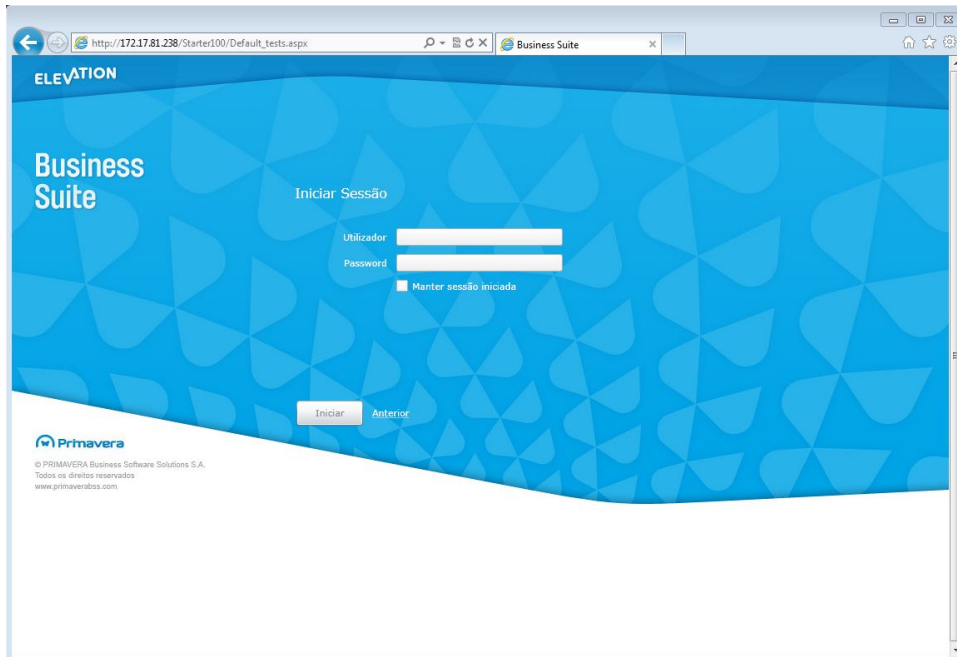


Figure 2.17: Developed Product landing page in Internet Explorer(version 9.0.8112.16421) and Silverlight plugin (version 5.1.10411.0).

2. State of the art

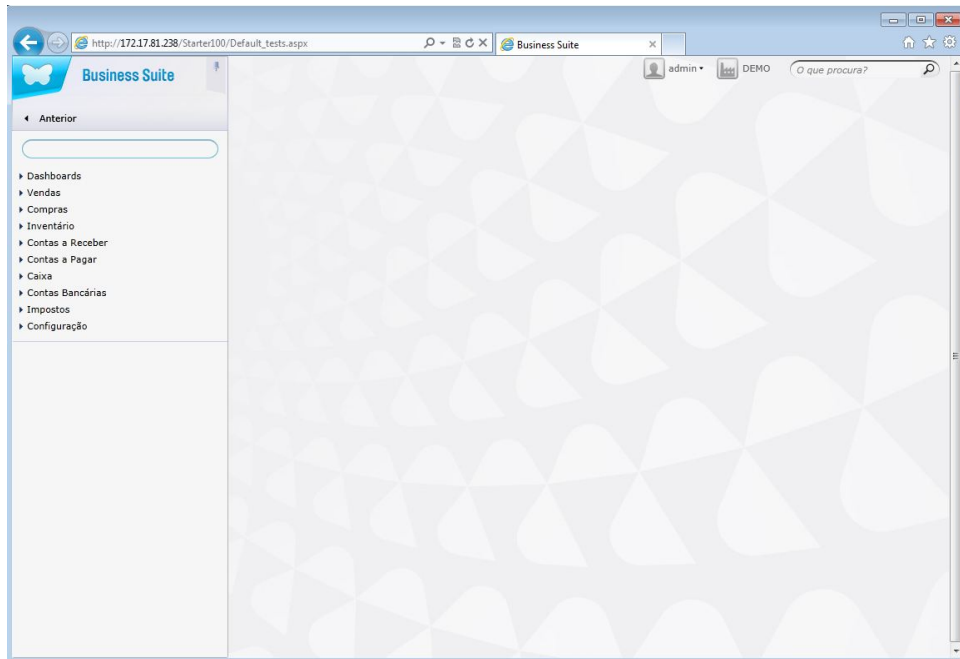


Figure 2.18: Developed Product home page in Internet Explorer(version 9.0.8112.16421) and Silverlight plugin (version 5.1.10411.0).

2.2.6 Domain-specific Languages

The basic idea of a Domain Specific Language (DSL) [21] is that it is a programming language that is targeted to a particular kind of problem, rather than a general purpose language that's aimed at any kind of software issue.

DSLs are very common in computing: examples include CSS, regular expressions [22], make, rake, ant and SQL.

The Athena Framework focuses on creating and exploiting domain models with abstract representations of the knowledge about the business logic that are part of the domain of Primavera's Products.

The model designers presented on the previous chapter are no more than DSLs that were developed to take advantage of the business domain and for each a designer was built. As so, it was possible to give an abstraction about implementation details to the programmer.

Chapter 3

Development Tools

Summary

This section presents the tools that were used for the development of this project. We survey tools for software development, software testing, software management and infrastructures.

3.1 Software

Overview of the available software tools.

Some tools were already in use by Primavera: Visual Studio and .Net platform, White Plugin, Team Foundation Server and VMWare Server Infrastructure.

3.1.1 Development

- Microsoft Visual Studio

Microsoft Visual Studio is an Integrated Development Environment (IDE) from Microsoft. With this tool it is possible to develop Windows applications, Web applications, Web services and Console applications for all platforms supported by Microsoft Windows, Windows Mobile, Windows CE, .NET Framework, .NET Compact Framework and Microsoft Silverlight.

3. Development Tools

It supports many programming languages like C/C++, C#, F#, Python, Ruby, HTML/XHTML, Javascript and CSS. In addition to all the native features, Visual Studio allows the developer to write extensions in the form of plugins that extend the IDE capabilities.

3.1.2 Testing

- White

White is a framework for automating rich client applications based on Win32, WinForms, WPF, Silverlight and SWT (Java) platforms.

This framework is based on the .NET framework and it's Open Source licensed. The Automation programs that use the framework can be written in any .NET supported language, IDE and tools. White provides a rich and easy tool to test automation since it hides Microsoft's UIAutomation library, on which White is based.

- SilverlightSpy

SilverlightSpy is a visual runtime inspector for Silverlight based applications.

The application has a proprietary web browser. When loaded with a silverlight application it uses the built-in XAML explorer to give an overview of the application GUI controls and properties.

- Ulspy

It is a tool that analyses, at runtime, the hierarchical structure, properties values and raise events of user interfaces developed with Microsoft's UIAutomation Library.

It also allows the user to interact with the user interface by raising events.

3.1.3 Management

- TFS

It is a Microsoft product intended for collaborative software development projects that offers source control, data collection, reporting and project tracking.

3. Development Tools

It is available as a stand-alone software or integrated with the Visual Studio Team System.

3.2 Infrastructures

Overview of the infrastructure tools available for supporting the development.

- Win2008R2-Athena

It is a VMware virtual machine. A simulation of a machine running Microsoft's Windows Server 2008 (64-bit) with the following emulated hardware specifications: 2 Processors, 2GB of RAM Memory, 24.00GB of Hard Disk Storage and a bridged network connection.

- VMWare Server Infrastructure

It is a server suite, developed and supplied by VMware, Inc, that allows to create, edit and play virtual machines. It allows remote access to the virtual machines since it uses a client-server model.

3.3 Others

Overview of some support tools.

- Microsoft Excel

It is a Spreadsheet application owned and developed by Microsoft.

Microsoft Excel provides calculation, graphing tools and pivot tables functionalities. It takes part of the software bundle also developed by Microsoft named Microsoft Office.

Chapter 4

Test Automation Component

Summary

This chapter presents the adopted strategy for the automation of testing Athena specified and generated applications. It presents the decision making process and developed artifacts of the Test Automation Component for the Athena Framework.

4.1 Objectives and Motivation

The Development Process of Primavera Software Products always followed a methodology based on Waterfall methods.

A major objective of the Athena Framework was to "agilize" the SDP of Primavera Products. As shown in Section 2.2, Chapter 2, this was achieved by abstracting the programmer from the source code complexity. Any change that the developer would make to the product using the designers would be converted by the Athena Framework in a new iteration of the product.

Since the creation of the Athena Framework, a solution of integrated Automatic Testing was always a concern and a goal for the creators of the Framework. As such, since the beginning, a very limited test automation component was developed and integrated within the Athena Framework. Even though this component existed previously, it consisted on very limited Visual Studio unit tests that were only able to perform basic interface operations on

4. Test Automation Component

a very limited set of views. The component was never a viable solution for the Quality Assurance Department and his development was halted very early.

Analysing the goals that were set by the team that created the Athena Framework, the test automation component must be able to fulfill the following requirements:

- Perform integration/acceptance testing;
- Use a black-box technique, interface tests, to perform automated tests;
- Be a generic solution for any product developed with the Athena Framework;
- Be a solution which adapts the tests fast and automatically for each new version of the product being tested.

The structure of a product developed in the Athena Framework is the following: An application is composed of several modules and each module is composed of several views. For each application there are several lines of development (i.e. latest development version, the latest released version, etc.)

There is also a Nightly Builds mechanism implemented. This means that every day, at night, all versions of all projects are compiled, which means that every day there is a new version of each project.

One of the objectives of the Athena Framework was to "agilize" software development by using designers in which the developer could very quickly and easily make changes to each product module.

This poses a serious problem because it means that, every day, there are (potentially) different application versions from the previous day (eg: new views and forms fields, field names, etc.).

These changes in the Framework are extremely difficult to manage if the test specifications are managed manually, so an alternative to manage them automatically had to be found.

4.2 Architecture

In this section an overview of the solution's global architecture will be presented. On the next sections, each of the solution components will be presented in detail.

The architecture and components will be presented using various diagrams that abstract the many solution dimensions, which isolates the various components and interactions between them. The solution will be presented on a top-down approach.

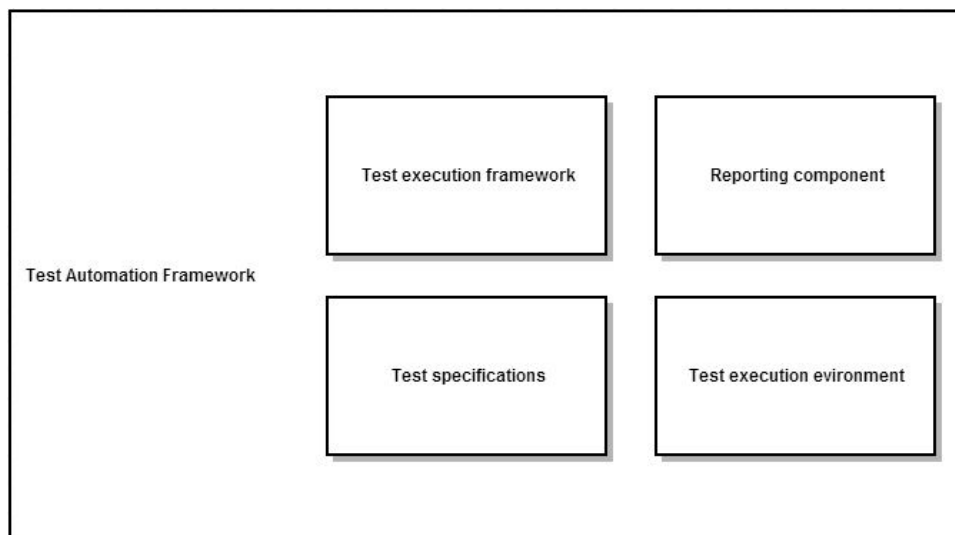


Figure 4.1: Test Automation Framework - Components global overview

The previous diagram presents a global overview of the most important components of the solution. The solution consists on 4(four) main components:

- Test execution framework

The test execution framework corresponds to all the test projects and code for test execution on graphical user interfaces that use the Silverlight technology. The execution of tests depends(necessarily) on the data sources(data-driven) within the test specifications;

- Test specifications

4. Test Automation Component

The Quality Assurance Department test automation solutions already used Spreadsheets as a support for test specifications. The test cases are defined on Spreadsheets and can then be read by the existent test solutions. The Spreadsheet technology used at Primavera is Microsoft's Excel and it will also be used in this project as a constraint. It corresponds to the set of excel files which contain the data for the tests(data-driven) and also for the test execution scripts that are no more than sequences of test cases from the excel files arranged on a business logic process sequence. Included in this component there's also the tools for automatic generation and manipulation of the excel test specifications;

- Test execution environment

There are many test execution environments. The test execution framework allows the tests to be executed in almost any computer. The requirements assuming that such computer has access to the application being tested(the application can be installed locally on the machine or through the internet network), access to the test reporting structures(email service and results database service);

- Result report

The test results are treated in two separated ways:

- Email

The test results are sent, by email, at the end of the test script execution. The results are presented to the user through a user-friendly html file with the information about the expected and actual test result for each of the test cases from the test script. The html also contains the execution and error log, as well as the context informations;

- Database

The results of the test script are stored on a central database.

Test execution framework

4. Test Automation Component

A central component of the test solution is the Test Execution Framework. This is the data-driven framework that will receive test specifications as an input, execute the test cases defined on the test specifications and output the results.

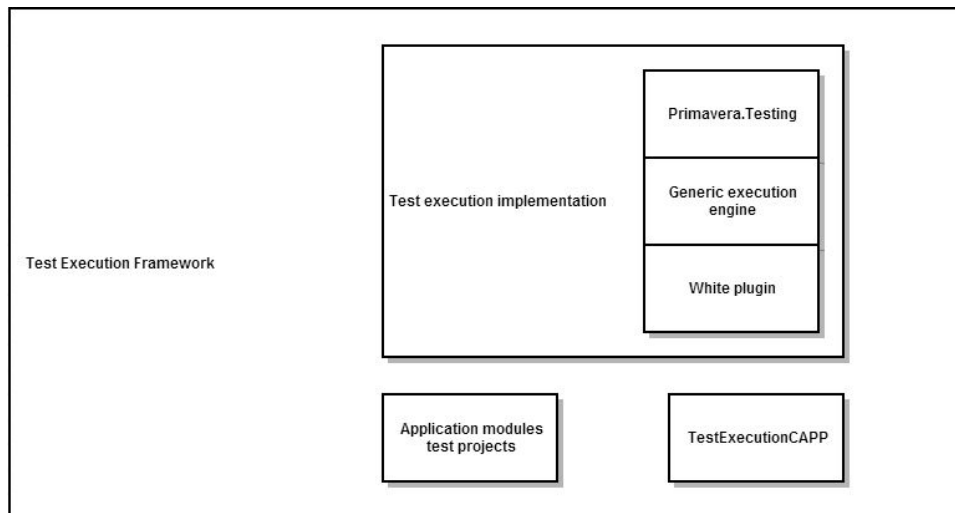


Figure 4.2: Test Execution Framework - Components global overview

- Test execution implementation
It corresponds to source code artifacts, methods and algorithms that allow the execution of tests to be conducted on Silverlight interfaces;
- Application modules' test projects
Each module of an Athena application has a test project that allows access to the designer's dsls which allows for the test specifications to be generated. Each test project also allows automated testing to be conducted on the module views;
- TestexecutionCAPP
The test execution console application is a program developed for the execution of test scripts. Test scripts are logic sequences of test cases that can or cannot represent business processes.

Test specifications

The Test specifications component aggregates all the input data for executing tests. It also includes data management tools, which allows the designed test cases to evolve along with the application's iterations.

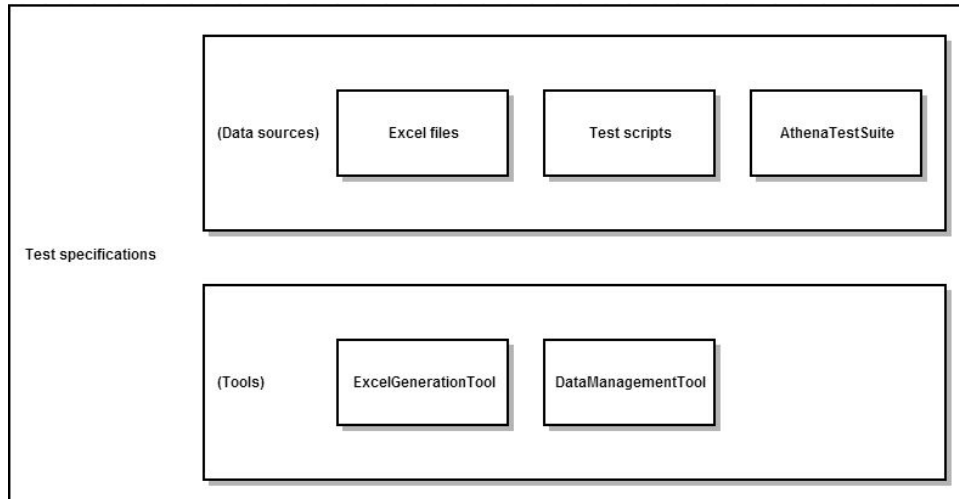


Figure 4.3: Test specifications - Components global overview

- Excel files
For each automatable view of each Athena application module, an excel file, with data for the tests of the view, exists;
- Test scripts
It corresponds to sequences of tests from the excel file that, when logically sequenced, correspond to complex business processes;
- AthenaTestSuite
It is a database which holds the information about each view's graphical user interface specification(controls). This database is always up to date with the most recent versions of the Athena applications;
- Excel generation tool
The tool was developed to allow the test specifications files(excel files) to be generated

4. Test Automation Component

without any human intervention. It generates each excel file for each module's view, based on the view's graphical user interface specification, present on the AthenaTest-Suite database;

- Data management tool
It provides "find and replace" and "copy" operations between the views test specification excel files.

Test execution environment

In order to ensure testing correctness, validity and effectiveness, the environments where the tests are executed are of extreme importance. At Primavera there was already a virtualization environment implemented. This environment separates the developer's/tester's local environment from a staged "real cenario" environment, thus assuring testing correctness. Test validaty is also assured because we have complete control over the test cenarios, which means, the tester can setup a virtual machine on the network with whatever software characteristics he wants. Accordingly, testing can be effective because it is possible to cover the needed test scenarios with precision.

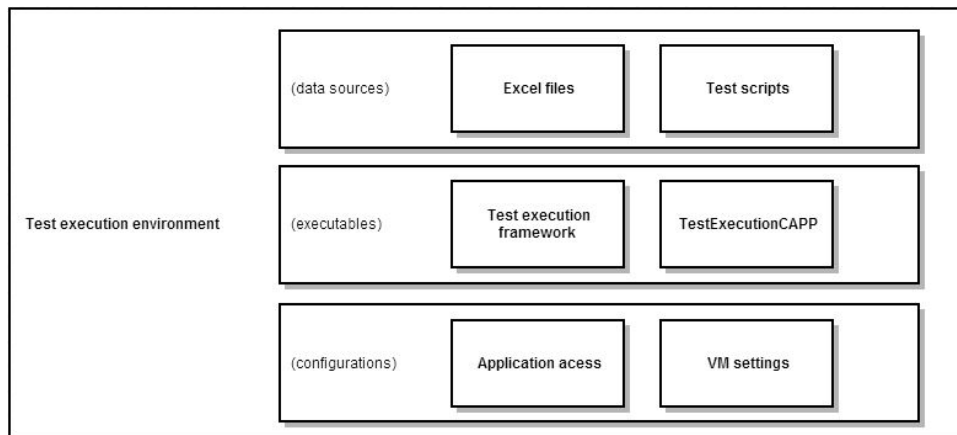


Figure 4.4: Test execution environment - Components global overview

4. Test Automation Component

- Excel files
The files that contain the test cases specifications for each view;
- Test scripts
The script(s) that will be executed;
- Test execution framework
The binaries(dlls) of each of the application module's test projects and those of the test implementation projects;
- TestExecutionCAPP
The tool that will automatically execute the test script;
- Application access
The test environment must provide access to the version of the application being tested. The application may have been previously installed on the machine or it can be accessed through the network, which requires either local access to the machine where it is installed or internet access.
- VM settings
It corresponds to the operating system of the VM as well as the versions of the software installed.

Reporting component

4. Test Automation Component

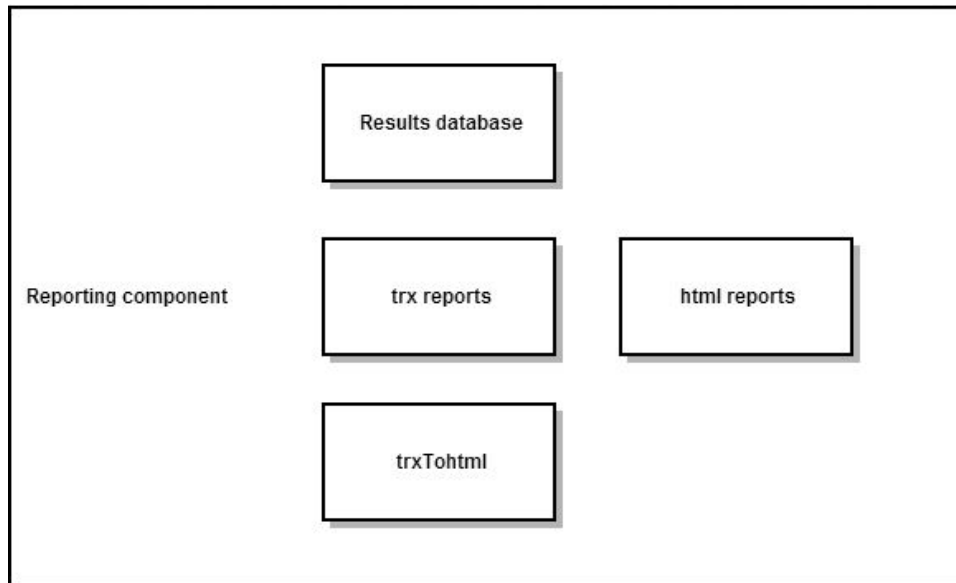


Figure 4.5: Reporting component - Components global overview

- results database
The results of each test script executed are stored on a database, as well as the information about the test environment;
- trx reports
After each test case(each step of the test script) has been executed, a trx extension file is produced and stored;
- html reports
After a test script has been executed, an html file is generated automatically with the most relevant information about the test results. This file is then sent by email to the test responsible;
- trxTOhtml
This tool was developed to read and compile the information of each trx file in order to generate an html file that aggregates the most relevant information about the results of a test script execution.

4.3 Excel Test Specification Generation

As mentioned before, the need to automate the generation of the test specifications arises from the fact that the Athena development environment potentially produces a new application from day to day (derived from the changes made).

The maintenance cost of the test specifications (data-driven) would be great and it may even be infeasible for temporal matters (the limit would be a day to update the specifications and automatically run tests).

Thus, the generation of specifications for data-driven testing follows a model-driven implementation.

To perform graphical user interface automated testing in Athena applications it is only required to know which controls are part of each view of each module of each application.

4.3.1 The Silverlight technology

Microsoft Silverlight is a powerful tool to create and deliver rich Internet applications and media experiences on the Web.

Silverlight 5 builds on the foundation of Silverlight 4 for building business applications and premium media experiences. Silverlight 5 introduces more than 40 new features, including dramatic video quality and performance improvements as well as features that improve developer productivity. [23]

The interface layer of the Athena framework generates the graphical user interfaces with the user on the Silverlight technology.

From the point of view of test automation, it is important to understand how the elements of the graphical user interface of a silverlight application are structured.

Using a tool like Microsoft Inspect.exe [24], which makes use of the Microsoft UIAutomation api, the visual UIAutomation tree and information about the controls of the application can

4. Test Automation Component

be visualized.

The following figures show the information of a Ui Automation control tree for the Login window of an application developed in Athena with a Silverlight interface.

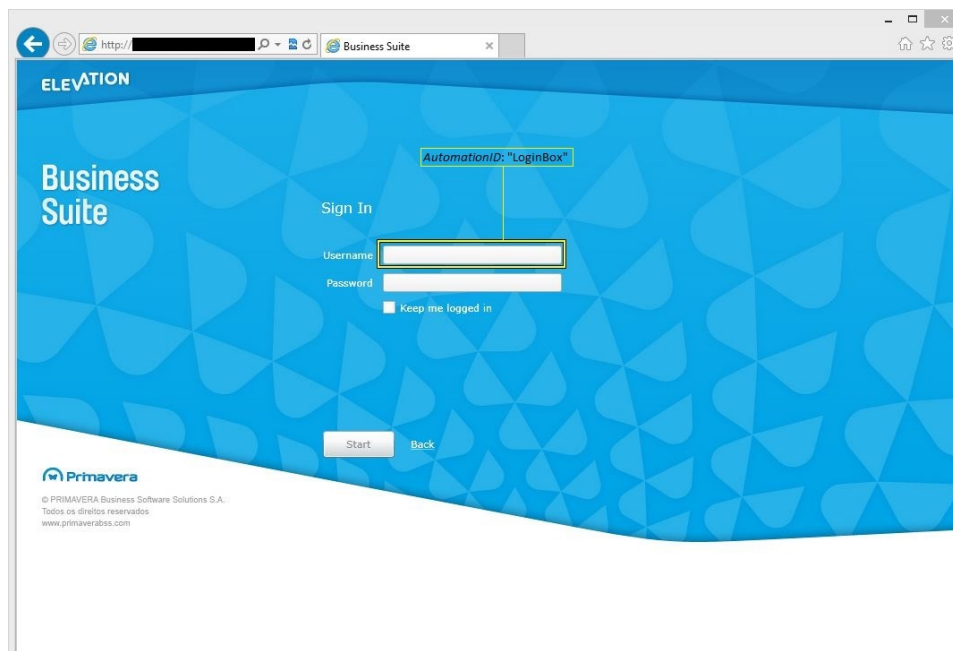


Figure 4.6: Login window of an Athena application(Silverlight technology)

Are context variables, which can be useful for automation (ie, if a check box is not visible it makes no sense to try to trigger a click event on the same).

4.3.2 XAML model

The first model studied and used for the generation of the test specifications was the XAML model [25].

XAML is a declarative markup language. As applied to the .NET Framework programming model, XAML simplifies creating a UI for a .NET Framework application. We can create visible UI elements in the declarative XAML markup, and then separate the UI definition from the run-time logic by using code-behind files joined to the markup through partial class definitions. XAML directly represents the instantiation of objects in a specific set of backing types defined in assemblies [25].

Figure 4.8 shows the xaml files for the views of the module "Purchases" of the Business-Suite application(MainLine line):

4. Test Automation Component

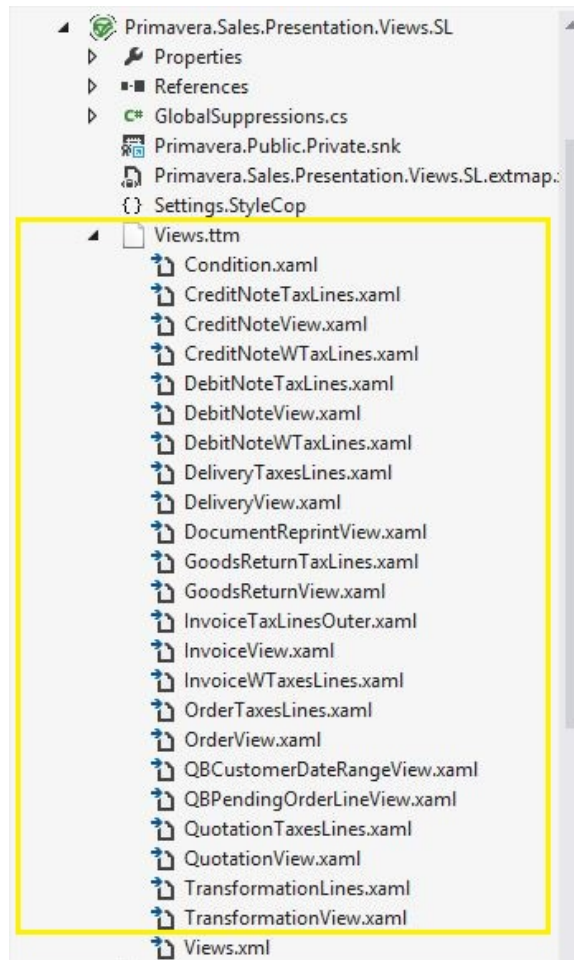


Figure 4.8: Xaml files for the views of the module Purchases (Business Suite - MainLine)

Each Xaml specification file corresponds to a view of the respective module.

The files are automatically generated using a T4 Text file template [26] Views.ttm which is transformed upon the module's build.

The template uses the Presentation Model to generate the code for each xaml file. This way it is guaranteed that the .Xaml contains the most recent specifications of the view.

A parser was then developed to extract, from the model's code, the most relevant information about the controls of each view, which resulted in the creation of a class "XamlView-Parser.cs" that is nothing more than an XML parser.

The following figures show an excerpt of the code of the file CreditNoteView.xaml, which iden-

4. Test Automation Component

tifies a control of the view Credit Note and the identification of the control on the interface:

```
<!-- BuyerCustomerParty Field -->
<ctl:ItemsFormField
  Caption="{Binding Source={StaticResource resources}, Path=RES_LBL_Field_CreditNoteView_BuyerCustomerParty}"
  LabelWidth="140"
  Width="Auto"
  VerticalAlignment="Top"
  HorizontalAlignment="Stretch"
  FieldType="SearchText"
  EnableSearch="True"
  FieldName="BuyerCustomerParty"
  Value="{Binding Source={StaticResource viewModelFinder}, Path=ViewModel.BuyerCustomerParty.Value, Mode=TwoWay}"
  IsStopField="True"
  DrillDownEnabled="True"
  IsMandatory="{Binding Source={StaticResource viewModelFinder}, Path=ViewModel.BuyerCustomerParty.IsRequired, Mode=
  IsVisible="{Binding Source={StaticResource viewModelFinder}, Path=ViewModel.BuyerCustomerParty.IsVisible, Mode=
  IsEnabled="{Binding Source={StaticResource viewModelFinder}, Path=ViewModel.BuyerCustomerParty.IsEnabled, Mode=
  IsReadOnly="{Binding Source={StaticResource viewModelFinder}, Path=ViewModel.BuyerCustomerParty.IsReadOnly, Mode=
  ItemsSource="{Binding Source={StaticResource viewModelFinder}, Path=ViewModel.BuyerCustomerParty.ItemsSource, Mode=
  DisplayMemberPath="NaturalKey"
  >
<ctl:ItemsFormField.Columns>
  <ctl:ColumnBase UniqueName="NaturalKey" Caption="{Binding Source={StaticResource resources}, Path=RES_Reference
  <ctl:ColumnBase UniqueName="Description" Caption="{Binding Source={StaticResource resources}, Path=RES_Referenc
</ctl:ItemsFormField.Columns>
</ctl:ItemsFormField>
```

Figure 4.9: CreditNote view's xaml specification (Business Suite - MainLine)

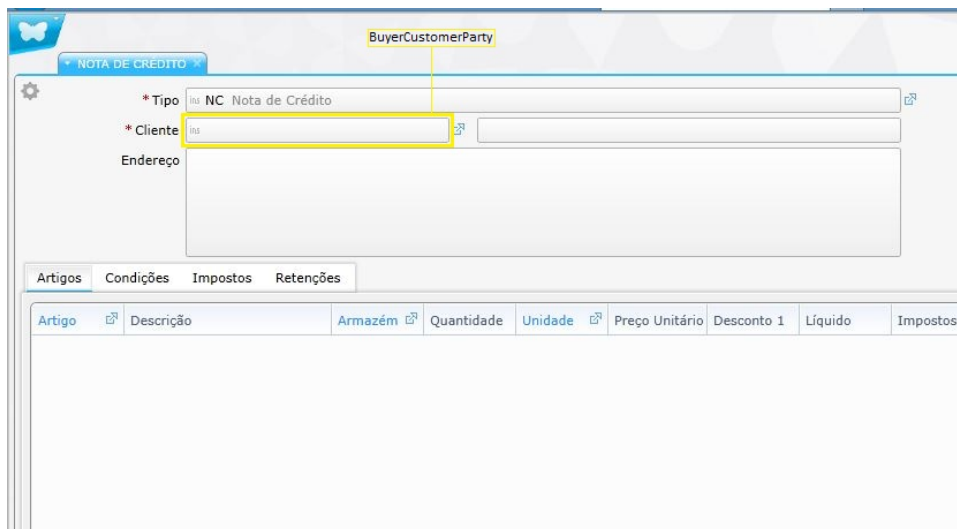


Figure 4.10: CreditNote view (Business Suite - MainLine)

At the end of the execution of the developed parser, an intermediate representation of each view is generated in a structure of Containers and Controls. Subsequently, it was used for the generation of excel specifications.

4.3.3 PRESENTATION model

The Presentation Model is one of the DSLs of the Athena framework designers.

This model also contains all the information of the structure (containers and controls) for every view since it is on the Presentation Designer that the programmer defines the structure (s) of the view (s) for each entity of the module.

This model is stored as a file in XML notation for each module of the Product. No example will be presented of the code of this file as it is a specific DSL. The model is accessed through the object representation using an existing Serializer. The access to the Presentation Model is made using a T4 text file. All relevant information of the views is read from the objects and stored locally and temporarily with the purpose of generating the excel specifications.

The meta-code below shows the strategy used to read information from the model, as well as the intermediate structure in which the information read is stored for later use.

```
1  struct ViewElement
2  {
3      string automationId
4      string name
5      string caption
6      string type
7      string isVisible
8      string isReadOnly
9      string isRequired
10     string parentContainer
11
12     ADD() {}
13 }
```

fontcode/presentationmodelintermediaterepresentationmetacode.txt

4. Test Automation Component

```
1 List<ViewElement> viewElements
2
3 // Process each Entity view.
4 foreach (View view in PresentationModel.Views)
5 {
6     viewElements = new()
7
8     foreach (Container container in View.Containers)
9     {
10        ProcessContainer(container)
11    }
12
13    ProcessViewElements(viewElements)
14 }
15
16 // Process each View Container.
17 ProcessContainer(Container container)
18 {
19     foreach (Container childContainer in Container.Containers)
20     {
21        ProcessContainer(childContainer)
22    }
23
24    // Process each Container Field.
25    foreach (Field field in Container.Fields)
26    {
27        // Add this field to this View Elements.
28        viewElements.ADD(field , container)
29    }
30 }
```

fontcode/presentationmodelparsermetacode.txt

4.3.4 Models Comparison

Initially, it was decided to use the XAML model to automatically generate the test specifications. This decision was made taking into consideration that the XAML model is an interface language from Microsoft and, as such, has a well defined syntax and semantics.

However, when using this model some concerns arised, namely:

- Second-level model

XAML files are generated automatically from the model the Presentation Model's model and so we can question the use of a second-level model when the first-level model is accessible and can be read without a higher cost.

- Model's information

The information contained in the XAML model is less expressive than the one contained in the Presentation Model Dsl. Some of this information(the properties of the field: mandatory, be visible and be read-only) can be used during the execution of automatic tests and is only present in the Presentation Model.

The Presentation Model is a DSL developed for the Athena Framework and is subject to change. In fact, during the various versions of the Athena Framework there were several changes to DSLs and some were even abandoned.

However the level of maturity of the Athena Framework at the time of writing this thesis, and the guarantees given by the team responsible for his development, makes this model a better alternative to the XAML model. Therefore it was decided to use the Presentation Model as a model for generating test specifications and, thus allowing for a model-driven test automation framework to be developed. It has the same "cost" as the XAML model with the advantages that have been mentioned.

4.3.5 Data generation

So the tests performed by the Test Automation Framework are considered "data-driven", the data from the test specifications (environment variables, data input, etc. ..) is stored in structures external to the code that executes the test itself.

So the tests performed by the Test Automation Framework are considered "model-driven", the test specifications are generated from the Presentation Model.

This section will present how, by using the Presentation Model, the test specifications are generated and the excel files automatically created.

The following figure shows the diagram of interaction between the various components of automatic generation of test specifications for excel files using the Presentation Model.

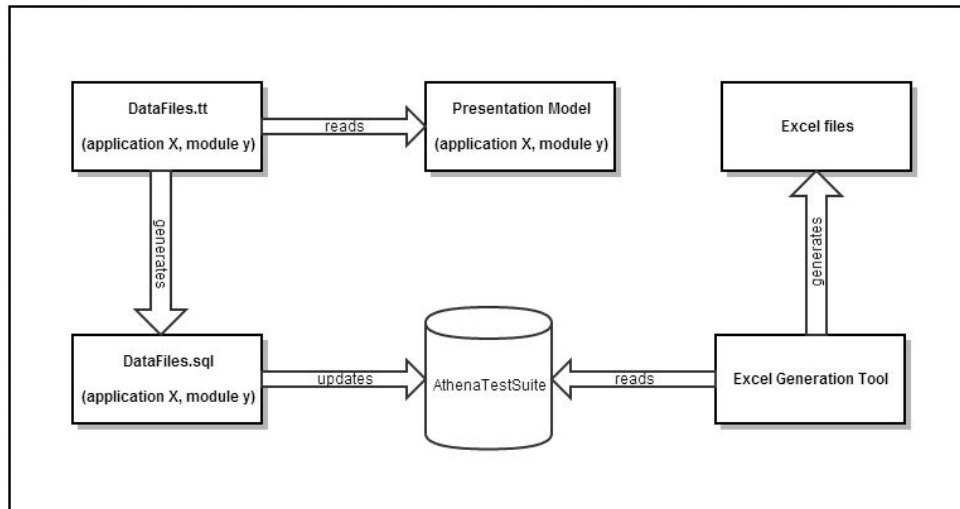


Figure 4.11: Test specification generation diagram

Presentation Model

The model from which the information of each view controls will be extracted;

DataFiles T4 Text Template, DataFiles sql file

The code responsible for accessing the Presentation Model and generate a file with sql queries;

AthenaTestSuite Database

The database used to store the most up to date information concerning each view controls for each module of each application and line. It is updated by the queries in the file DataFiles.sql. The database structure follows the structure of the application developed with the Athena Framework. It is designed to be used with any Athena application.

4. Test Automation Component

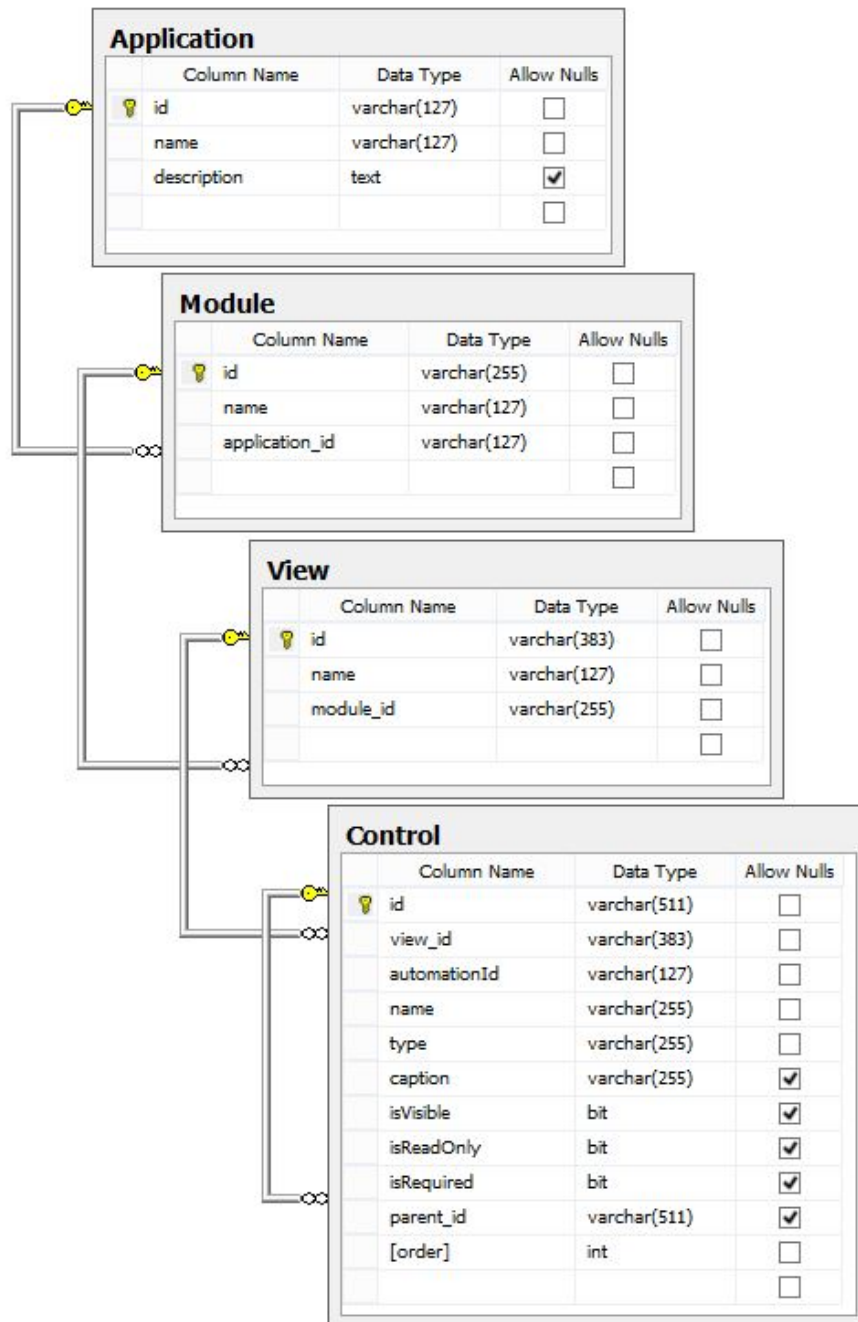


Figure 4.12: AthenaTestSuite database (Database diagram)

Generation Tool

A tool developed to generate excel files with the latest specifications for testing using the information contained in the database AthenaTestSuite;

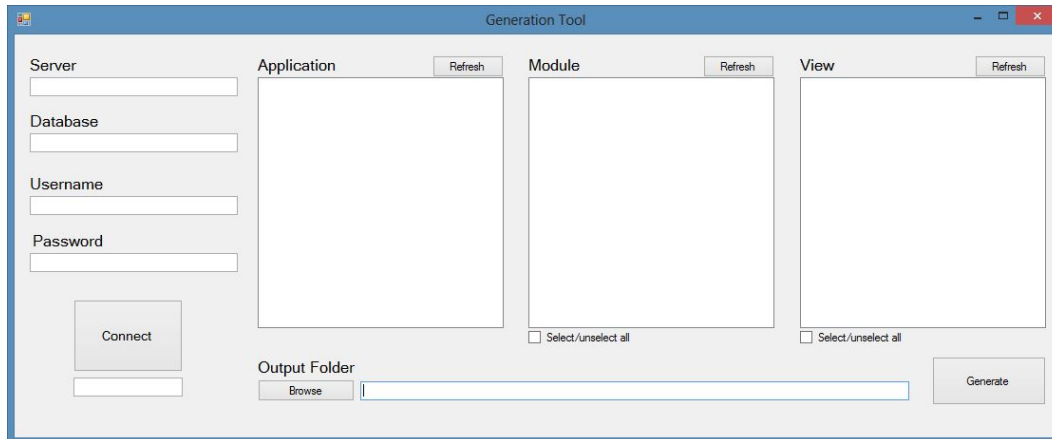


Figure 4.13: Excel generation tool (User interface)

Excel test specifications

The excel files generated by the tool presented above follow the same logic of the Athena applications. Each file contains the test specifications for a single view of a particular module of a particular Athena application.

Each excel file is generated automatically from the (latest) information on the controls of the View that is in the database AthenaTestSuite. Thus, the excel specifications are generated from the Presentation Model and we can say that the specifications for testing are, in fact, model-driven. On the other hand, it is possible to design as many test cases as wanted in each excel file by setting different values for each test case environment variables and also data input for the controls. As such, it can be stated that the tests performed by this Framework are indeed data-driven.

Each excel file consists of multiple excel sheets, each with its own logic:

4. Test Automation Component

- Controls sheet

The controls sheet named "Types" has the AutomationId and Type information of all the controls of this test specification file view.

- Main sheet

The columns represent test variables and the header of each column is the variable name.

Each line(except the header) represents a definition of a test case, in which the tester defines the value to be attributed at each variable. These variables include test environment variables, as well as the input for each control in the view.

- Validations sheet

A sheet in which the testers define sql validations (sql queries) and the expected result of each validation. The validations are executed after the test case execution.

In appendix A.1 is presented an example of the user interface view "Warehouse", from the module "MaterialsCore", of the application "BusinessSuite". In appendix B, the test specification excel for the view "Warehouse is presented. "Appendix B.1, B2 and B.3 presents the Controls, Main and Validations sheet, respectively.

4.4 Generic Execution Engine

Before presenting the generic execution engine it is important to remember the White Plugin, presented on subsection 3.1.2 (Testing) of Chapter 3 (Development Tools).

The White plugin is an open-source, written in C#, and it supports all rich client applications, which are Win32, WinForm, WPF and SWT (Java). It is .NET based and does not require the use of any proprietary scripting languages. It provides a consistent object oriented API and it hides all the complexity of Microsoft's UI Automation library and Win32 Windows messages.[27].

The White plugin interacts with the application through the UI Automation elements, allowing the triggering of events in the controls.

The generic execution engine is the main component of the test execution framework. Its aim is to perform operations in a particular GUI using the controls data input present in the excel files. This engine has the ability to adapt to the interface's technology because it uses an Interface (programming) implementation; this is, an intermediate representation of the user's interface underlying technology. For this project the Interface methods were implemented using the White plugin.

The following diagram shows the flowchart for implementing the generic execution engine.

4. Test Automation Component

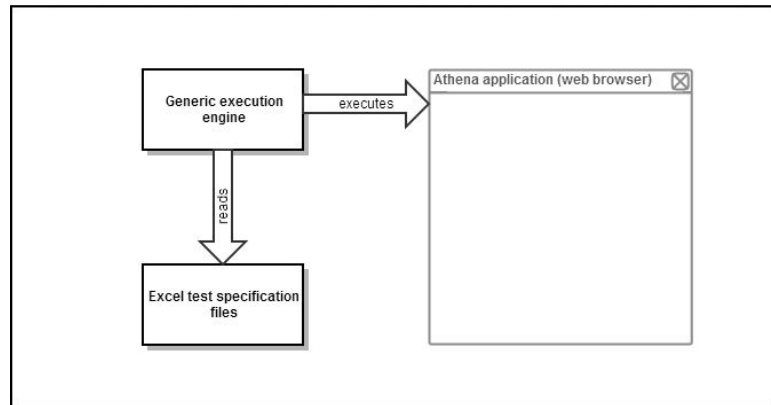


Figure 4.14: Generic execution engine diagram

4.5 The Validations component

The test validation component is one of the most important of the Framework.

This component consists of a collection of processes that evaluate the actual result of the test execution compared with the expected result, and "computes" the end result of each test.

At the end of the execution of each test case, two types of validations are made to assess the final outcome of the test case:

- Graphical User Interface Validations

Validations that are made once the test run is completed. It consists of seeking errors in the main application's GUI that may have occurred due to the test execution.

Some examples of validations may be: Searching for a window with an error message after the test as been made, searching for a window with a warning message after or during the test execution and validating if the application is in a valid state, that is, if the windows that are being displayed on the application's interface correspond to a valid state of the application. The result of this validation can be that no error window was found or that one or more error windows were found.

- Database Validations

It corresponds to the values validated on the application's database. Those values may have been created, deleted and updated during the execution of the test because of the business process being tested.

This validations are made by running SQL queries against the application's database and comparing the queries' results with the expected result. Both the sql queries and expected results are defined for each test case on the test specification spreadsheets. Each sql validation has an effective result of the executed query and an expected result set by the test case designer.

4. Test Automation Component

When the two main validation processes are concluded and the results of each are evaluated there's a third process that evaluates the final test result by comparing the validations' results with the expected outcome of the test.

4.6 The Reporting component

A very important piece of a Testing Framework is the results reporting component.

Running tests is not worth while if the test results are not clear and do not quickly reach the right people.

The test reporting component has two similarly important dimensions: Reporting by email and reporting to the database.

Email reporting

At the end of each test case execution a VisualStudio trx file is produced. This file contains, in XML notation, the information about the test and its implementation and can be read using the Visual Studio.

In order to make the results more accessible and to distribute them in a user-friendly format, a "trxTOhtml.exe" utility program was developed. The tool collects data from the trx files and creates an html file with the same data. The html report can be accessed on any browser that supports JavaScript.

When the execution of a test reaches the end, the various trx files are transformed into a single html file with the results information. The file is then emailed to a number of key people within the organization.

Sending the test results on html format allows:

- Immediate access to test results;
- There is no need of specific tools (eg Visual Studio);
- Reading and understanding the results regardless of the level of expertise of those

4. Test Automation Component

who read them;

- Increased awareness of the organization regarding the tests being executed.

The html file is organized as follows:

- General summary of the tests performed (number of failed test cases, past, test duration, etc.);
- Summary of the result of the execution of each test set;
- Summary of the test cases of each test set;
- Detailed information (for the application log, error log, print screen of the error) of each test case.

Figure 4.15 shows an example of an Html test results report file.

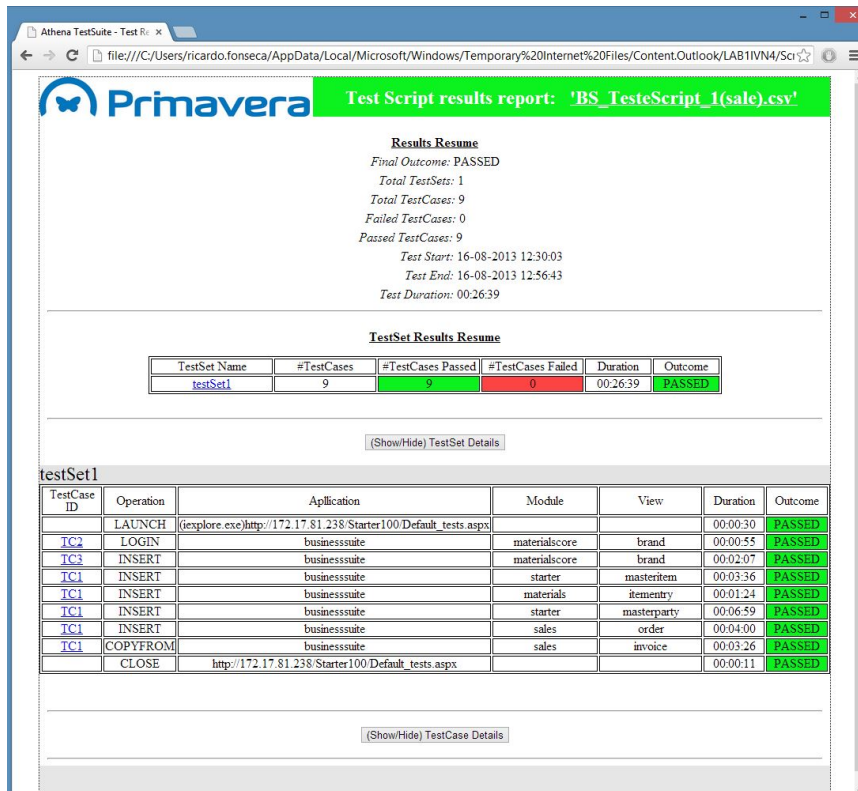


Figure 4.15: Reporting component - Html report example

4.7 Excel Data management Tool

To maintain the test specifications valid for each iteration of the product, the specifications are automatically generated using the Presentation Model.

However, it is still necessary to find a way to migrate the oldest test data specifications(definition of test cases) to the latest specifications.

The need to copy the test cases definitions from older test specification to the latest ones comes from the need to adapt the tests to potential changes in the product (eg, existence of new fields and elimination of others). Without automating this process, the effort would be pointless because the tester would take the trouble to manually copy the data to the new specifications. This task, in addition to having a huge cost, would be very susceptible to errors by the Tester.

Therefore an utility, with the following features, was developed:

- Data Copy
It allows the Test specifications of older versions to be reused;
- Data manipulation through Find And Replace
It allows the manipulation of specific values on the excel file and will allow a quick adaptation of the test specification to the test scenarios.

The following figure presents the graphical user interface of the developed tool.

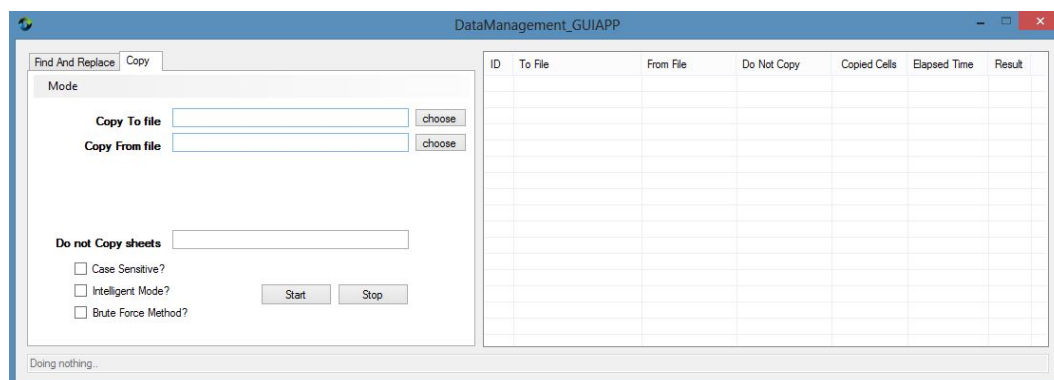


Figure 4.16: Data management tool (User interface)

4.8 Business Scripts Execution

Executing Tests is only useful when the tests itself are useful. The Athena applications developed by Primavera - Software Factory are applications with very specific and well defined business logic. As such, the Test Automation Framework should allow the automations of these business processes. For this, the framework has the ability to perform eight types of test cases:

1. **Open Application**

It corresponds to the action of opening a desired application(url) on a desired browser;

2. **Login**

It automates the process of Login in a Athena developed application. The login process is standard for all Athena applications since it's provided by the Athena Framework;

3. **Create**

The Create test allows the Tester to Create a document of any view of any module. It corresponds to an Insert operation on the application database;

4. **Update**

It corresponds to updating the fields or state of a document already existent on the application;

5. **Delete**

This tests allows for the deletion of documents already existent in the application. The Tester must be aware that some documents might remain stored in the application's database due to the business logic of the application;

6. **Copy To (Transformation)**

It automates a very specific business process of the domain of the application. It corresponds to the creation of a document by using the data of another document;

7. **Copy From (Transformation)**

Similarly to the Copy To test, it allows the creation of a document by copying the data of another existent document;

8. **Compare**

This test allows a quick comparison test between the values that appear on the form of an existent document and the expected values. This test should not be seen as a test case validation, even though the purpose of this test might be, after all, a validation of the values on the gui form of a document. The need for this test-case was raised by the Quality assurance department of Primavera- Software Factory.

The tester has the ability to define the amount of tests as he wants on each excel test specifications file. However, the value of testing appears only when the test cases can be sequenced on business a logic.

The test execution component has the ability to sequentially execute test cases; however, it has no awareness of the business value of their sequencing. For this, a tool that allows running test cases on a pre-defined order was developed and it allows complex business processes to be simulated and provides business value to the tests performed by the Test Automation Framework.

The tool allows the test designer to define the sequencing of test cases execution on the following logic:

- Test script

A test script contains the definition of tests for an Athena application. It contains the definition of a set of test sets with a proper sequence and logical order;

- Test set

A test set consists of a set of test cases. The test cases may be of any view of any application module to be tested. The test cases are ordered in a logical sequence that allows testing business processes.

4. Test Automation Component

- Test case

Meets the definition of an entry (row) in the test specification excel of a view, which was created by the test designer. Each entry defines a test case type of the 8 types presented previously and their environment data.

The following Figure shows the csv file that defines a Test script execution for an Athena application. The file is then interpreted and executed by the tool running in the test environment.

	A	B	C	D	E	F	G
1	testeSet	operation	application	module url	view	tcid	moduledllfullpath
2	testSet1	OPENAPPLICATION	iexplore.exe	http://172.17.81.238			
3	testSet8	LOGIN	businesssuite	taxscore	exemptionreasoncode	TC2	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
4	testSet8	INSERT	businesssuite	taxscore	exemptionreasoncode	TC3	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
5	testSet8	UPDATE	businesssuite	taxscore	exemptionreasoncode	TC4	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
6	testSet8	DELETE	businesssuite	taxscore	exemptionreasoncode	TC5	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
7	testSet8	INSERT	businesssuite	taxscore	fiscaldocumenttype	TC1	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
8	testSet8	UPDATE	businesssuite	taxscore	fiscaldocumenttype	TC2	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
9	testSet8	DELETE	businesssuite	taxscore	fiscaldocumenttype	TC3	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
10	testSet8	INSERT	businesssuite	taxscore	itemtaxschema	TC1	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
11	testSet8	UPDATE	businesssuite	taxscore	itemtaxschema	TC2	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
12	testSet8	DELETE	businesssuite	taxscore	itemtaxschema	TC3	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
13	testSet8	INSERT	businesssuite	taxscore	itemwithholdingtaxschema	TC1	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
14	testSet8	UPDATE	businesssuite	taxscore	itemwithholdingtaxschema	TC2	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
15	testSet8	DELETE	businesssuite	taxscore	itemwithholdingtaxschema	TC3	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
16	testSet8	INSERT	businesssuite	taxscore	PartyTaxSchema	TC1	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
17	testSet8	UPDATE	businesssuite	taxscore	PartyTaxSchema	TC2	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
18	testSet8	DELETE	businesssuite	taxscore	PartyTaxSchema	TC3	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
19	testSet8	INSERT	businesssuite	taxscore	PartyWithholdingTaxSchema	TC1	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
20	testSet8	UPDATE	businesssuite	taxscore	PartyWithholdingTaxSchema	TC2	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
21	testSet8	DELETE	businesssuite	taxscore	PartyWithholdingTaxSchema	TC3	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
22	testSet8	INSERT	businesssuite	taxscore	TaxTypeCode	TC1	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
23	testSet8	UPDATE	businesssuite	taxscore	TaxTypeCode	TC2	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
24	testSet8	DELETE	businesssuite	taxscore	TaxTypeCode	TC3	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
25	testSet8	INSERT	businesssuite	taxscore	WithholdingTaxCode	TC1	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
26	testSet8	UPDATE	businesssuite	taxscore	WithholdingTaxCode	TC2	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
27	testSet8	DELETE	businesssuite	taxscore	WithholdingTaxCode	TC3	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
28	testSet8	INSERT	businesssuite	taxscore	WithholdingTaxType	TC1	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
29	testSet8	UPDATE	businesssuite	taxscore	WithholdingTaxType	TC2	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
30	testSet8	DELETE	businesssuite	taxscore	WithholdingTaxType	TC3	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
31	testSet9	INSERT	businesssuite	financialcore	accounttransfertype	TC3	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
32	testSet9	UPDATE	businesssuite	financialcore	accounttransfertype	TC4	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
33	testSet9	DELETE	businesssuite	financialcore	accounttransfertype	TC5	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
34	testSet9	INSERT	businesssuite	financialcore	CashFlow	TC1	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
35	testSet9	UPDATE	businesssuite	financialcore	CashFlow	TC2	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
36	testSet9	DELETE	businesssuite	financialcore	CashFlow	TC3	C:\Program Files (x86)\PRIMAVERA\Athena210\Star
37	testSet9	INSERT	businesssuite	financialcore	CashTransferType	TC1	C:\Program Files (x86)\PRIMAVERA\Athena210\Star

Figure 4.17: Purchases - BusinessSuite(MainLine)

Each line(except the header) defines one step of the test script. The script is structured as it follows:

- **"testSet"** column

It identifies the test set to which this test case belongs;

4. Test Automation Component

- **"operation"** column
It identifies the type of test case from the 8 types presented previously;
- **"application"** column
If the type of the test case is "LAUNCH"(Open Application) it defines in which browser the application will be launched. Otherwise it defines what application this test case belongs;
- **"module || url"** column
If the type of test case is "LAUNCH" (Open Application) it defines the url of the application to be launched. Otherwise it defines the module of the application to which this test case belongs to;
- **"view"** column
It defines the view to which the test case belongs;
- **"tcid"** column
It defines the unique identifier test case to be executed. The combination of the application, module, view and TestCaseId identifies as unequivocal the test case to be executed;
- **"moduleDllfullpath"** column
It defines where the test project module dll is and to which test case belongs to. It is used on the machine to execute this test case. Bear in min that only the test project of a target module can execute the test cases of the views of that same module.

Chapter 5

Case study: Business Suite

Summary

This chapter presents the BusinessSuite Athena application case study. Firstly, a brief description of the BusinessSuite application, secondly, all steps performed to design and execute tests on the BusinessSuite application with the developed solution, and lastly the results of the performed tests with graphics and their analysis.

5.1 Application Description

The ELEVATION Business Suite is a solution developed exclusively to be used in a web environment, constituting the first version of the new Primavera ERP, based on a technology platform developed internally (Athena Framework) and on the more advanced concepts of software engineering.

5.2 Designing Tests

The Test Automation Framework is to be used by automation Testers. The next steps describe the process of tests carried out with the tool, which include the design of test cases, their execution and the review of the tests' results.

5. Case study: Business Suite

It is important to note that the modules of the Athena product BusinessSuite already contain the test projects. During the last nightly builds, the latests view specifications were read from the Presentation Model and the AthenaTestSuite database was updated. All these processes ran automatically without any human action.

To begin the process of testing, the tester uses the generation tool to generate the latest excel specifications for testing:

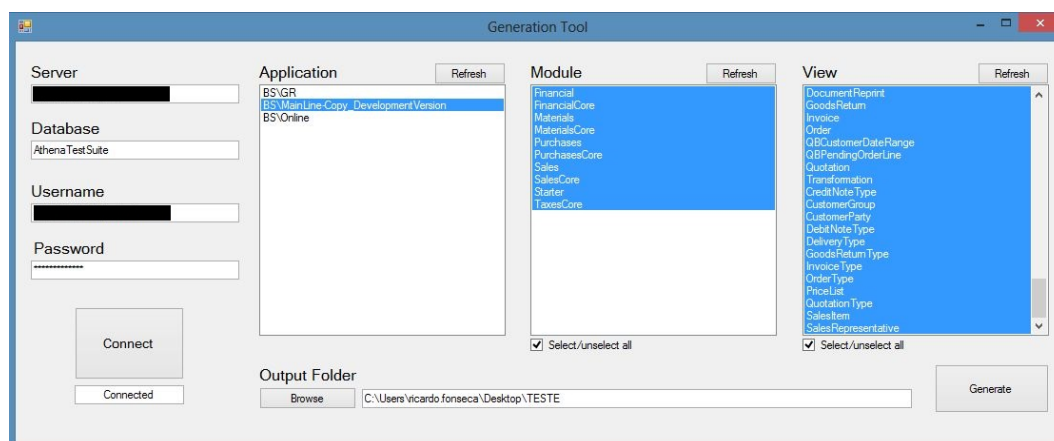


Figure 5.1: Excel generation tool (BusinessSuite case study)

Once the specifications are generated, the tester has access to the most recent specifications for testing. If the tester had already designed test cases in older test specifications versions he may decide to re-use them, thus using the tool "datamanagement_GUIAPP" to copy the old test cases from the older tests specifications to the latest:

5. Case study: Business Suite

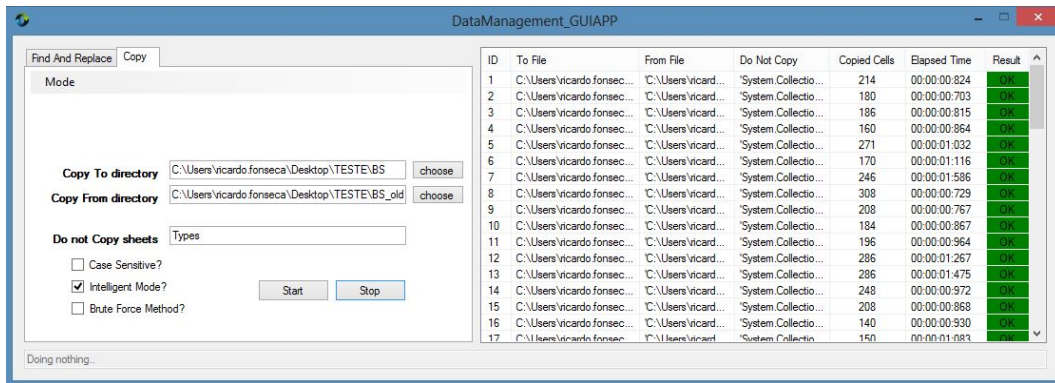


Figure 5.2: Data management tool (BusinessSuite case study)

After that, the tester evaluates the design of test cases to verify that they are valid. If they are not valid/not compatible for the newer version then it updates them in order to make them valid/compatible. Example: If there is a new field on a view, the most recent test specification excel have a new column which corresponds to the new field. Because on the older test specifications this field/column on the excel did not exist, then for each copied test case this field is blank, so, the tester may need to fill the excel with values for this field in each test case.

The tester may also wish to design new test cases and so, further bellow, a definition of a new test case of the type INSERT for the "Brand" view, from the MaterialsCore module is presented:

Main sheet("Brand")

A new line is added to the main sheet, which corresponds to a new test case, and the fields are filled with the environment variables and the control's input data for the test case.

- (COLUMN: VALUE)
- TestCaselid: TC4
- Enabled: TRUE

5. Case study: Business Suite

- TestType: Insert
- TaskFriendlyName: Marcas
- TaskId: RES_MenuCaption_Task_Brands
- Polarity: Positive
- LoadInitialValues: FALSE
- ListForEdition: FALSE
- OpenFromList: TRUE
- FileName: iexplore.exe
- Arguments: <http://172.17.91.338/BusinessSuite.aspx>
- Name: Business Suite
- Type: Silverlight
- BrowserType: IE
- BrandKeyTextFieldId: TESTAUTOMATION
- DescriptionTextFieldId: no description

The columns "BrandKeyTextFieldId" and "DescriptionTextFieldId" correspond to controls of the view "Brand"; all other columns correspond to environment variables for this test case.

5. Case study: Business Suite

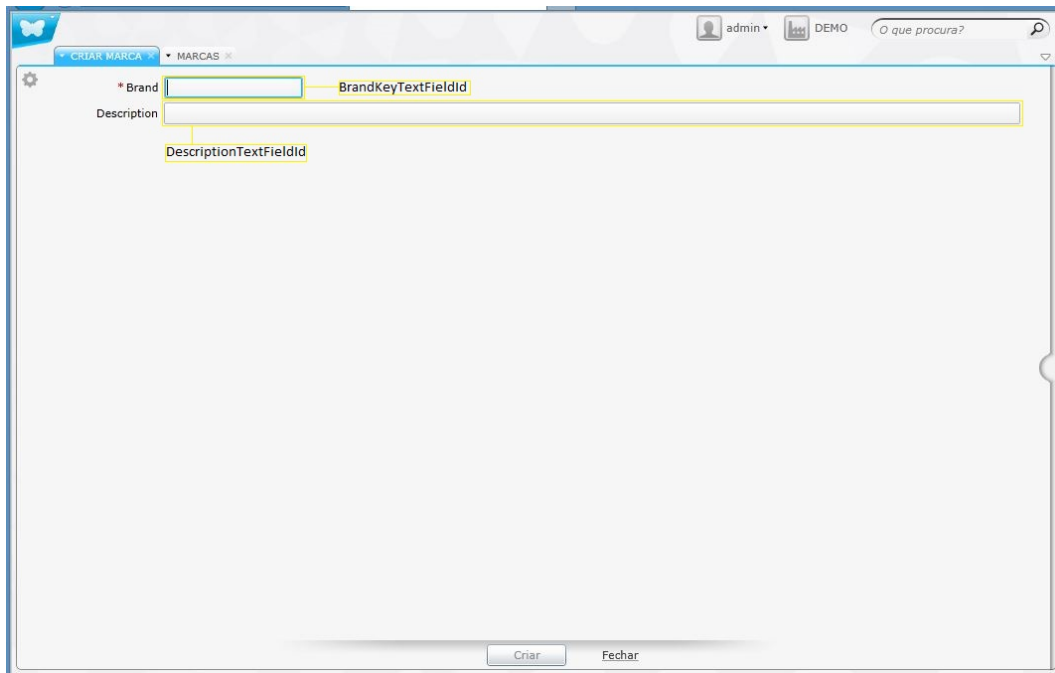


Figure 5.3: BusinessSuite Brand entitie's view (BusinesSuite case study)

Validations sheet("Validations")

In the validations sheet a new line is added which defines an sql validation for the test case.

- (COLUMN: VALUE)
- TestCaseld: TC4
- ValidationType: SQLValidation
- Server: 172.17.91.338
Primavera
- User: user
- Password: password

5. Case study: Business Suite

- QueryStatement: select naturalkey as result from bsdatabase.materialscore.brands where naturalkey = 'TESTAUTOMATION'
- ExpectedResult: TESTAUTOMATION

Once the test cases have been designed, the tester defines a test script for the application BusinessSuite. It organizes the defined test cases sequentially for each view of each module of the BusinessSuite application. Thereby designing the testing of complex business processes.

The following script sets up a test to a business process on selling.

	A	B	C	D	E	F	G
1	testeSet	operation	application	module url	view	tcid	moduledllfullpath
2	testSet1	LAUNCH	iexplore.exe	http://172.17.81.238/Starter100/Default_tests.aspx			
3	testSet1	LOGIN	businesssuite	materialscore	brand	TC2	C:\Program Files (x86)\PRIMA\
4	testSet1	INSERT	businesssuite	materialscore	brand	TC3	C:\Program Files (x86)\PRIMA\
5	testSet1	INSERT	businesssuite	starter	masteritem	TC1	C:\Program Files (x86)\PRIMA\
6	testSet1	INSERT	businesssuite	materials	itementry	TC1	C:\Program Files (x86)\PRIMA\
7	testSet1	INSERT	businesssuite	starter	masterparty	TC1	C:\Program Files (x86)\PRIMA\
8	testSet1	INSERT	businesssuite	sales	order	TC1	C:\Program Files (x86)\PRIMA\
9	testSet1	COPYFROM	businesssuite	sales	invoice	TC1	C:\Program Files (x86)\PRIMA\
10	testSet1	CLOSE	iexplore.exe	http://172.17.81.238/Starter100/Default_tests.aspx			

Figure 5.4: Test script for testing a Sale's business process (BusinessSuite case study)

The created script, defines the following business process sequence:

1. (Line 2) Open the BusinessSuite application on the Internet Explorer browser;
2. (Line 3) Login in the application;
3. (Line 4) Create a new Brand;
4. (Line 5) Create a sale's item of the created brand;
5. (Line 6) Create a stock entry for the created sale's item;
6. (Line 7) Create a Customer;
7. (Line 8) Create an Order for the created customer and the created sale's item;
8. (Line 9) Transform the created Order into an Invoice;

9. (Line 10) Close the BusinessSuite application.

5.3 Executing Tests

To run the test script, the tester has to copy the excel specifications files and the test script to the testing environment.

The test execution environment had been previously prepared by one of the people responsible for the automation. The execution machine, in this case, is a virtual machine hosted on a virtual machine server and has the following features:

Hardware

- Processor: 2 x 4.556 GHz;
- Ram: 2048 MB;
- Hard Disk: 24.00 GB;
- Network: Bridged connection.

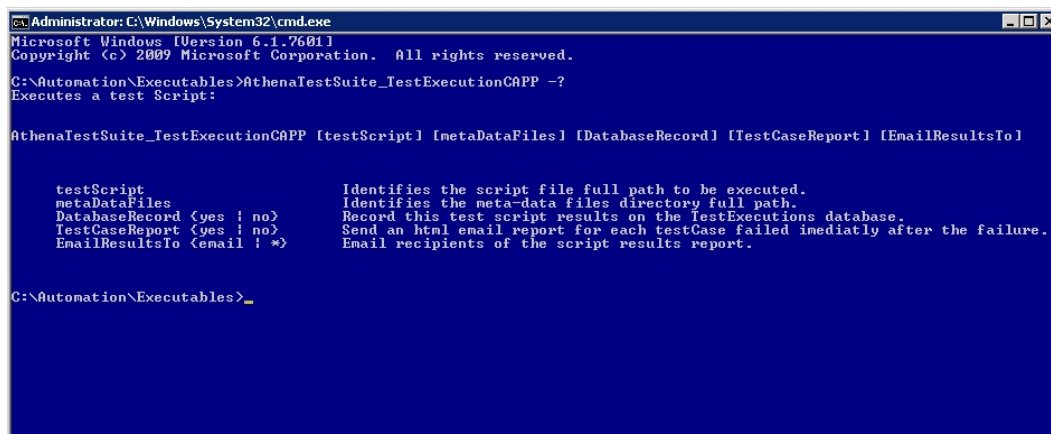
Software

- OS: Microsoft Windows Server 2008 (64-bit);
- Target version of the BusinessSuite application and a DEMO database;
- Test scripts execution tool, "AthenaTestSuite_TestExecutionCAPP";
- Html reports generation tool, "trxTOhtml".

5. Case study: Business Suite

In the context of this case study and for this particular test, the BusinessSuite application is installed on the same machine where the tests will be performed. However, for the tests execution it is only required to have access to the application in order to test it, whether this access is made locally or via the Internet.

The tester accesses the virtual machine and copies the test specifications and test scripts to the root of the hard disk. Then he uses the "AthenaTestSuite_TestExecutionCAPP" tool to launch the process that will execute the test script:



```
Administrator: C:\Windows\System32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Automation\Executables>AthenaTestSuite_TestExecutionCAPP -?
Executes a test Script:

AthenaTestSuite_TestExecutionCAPP [testScript] [metaDataFiles] [DatabaseRecord] [TestCaseReport] [EmailResultsTo]

testScript          Identifies the script file full path to be executed.
metaDataFiles       Identifies the meta-data files directory full path.
DatabaseRecord <yes | no> Record this test script results on the TestExecutions database.
TestCaseReport <yes | no> Send an html email report for each testCase failed imediatly after the failure.
EmailResultsTo <email | *> Email recipients of the script results report.

C:\Automation\Executables>_
```

Figure 5.5: Test execution tool (BusinessSuite case study)

The execution tool is responsible for sequentially executing the steps of the script, calling up each test case and invoking the respective module's test project binary of the view. To conclude, a test script results report is generated and sent to the e-mail recipients. Recipients then receive one email with the html file which can be opened in any browser that supports Javascript.

5. Case study: Business Suite

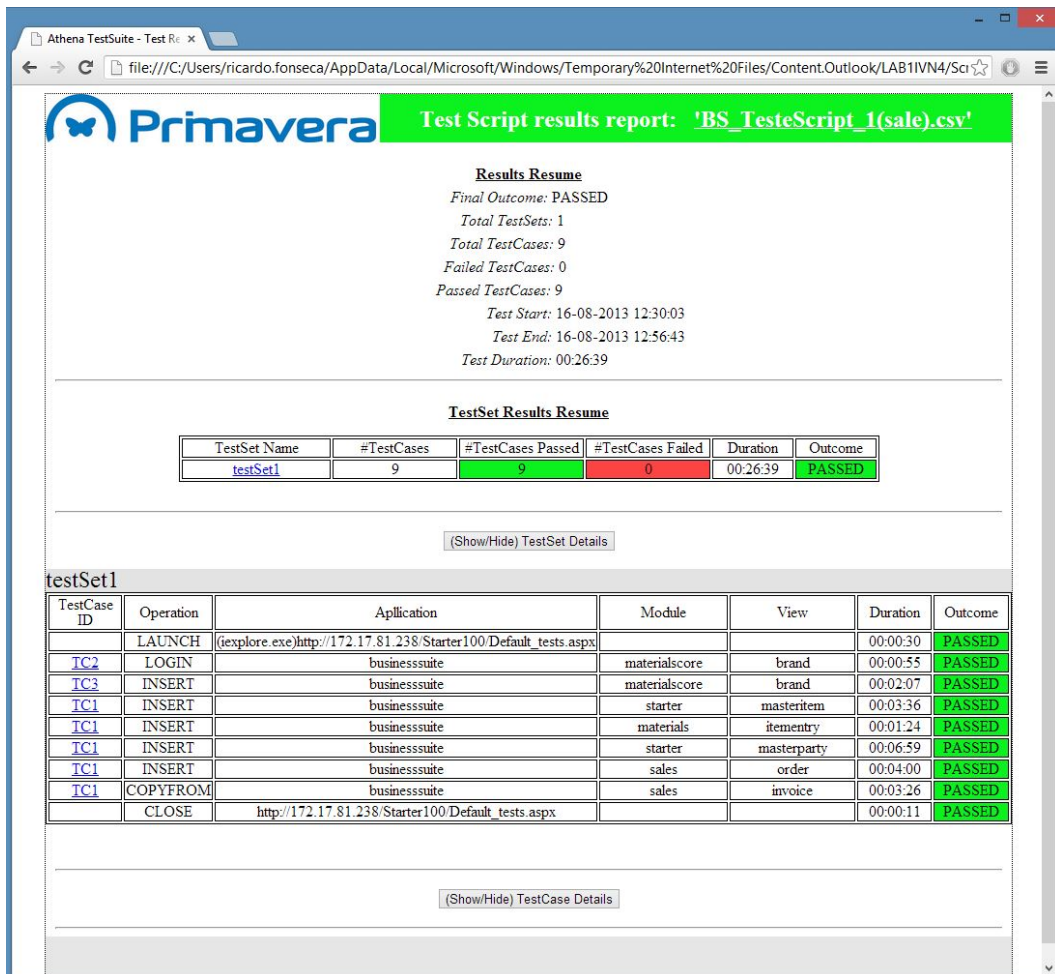


Figure 5.6: Html script results report (BusinessSuite case study)

The results report aggregates all the relevant information about the test script execution and the test case results. The end result of running the script was successful. All test sets and their test cases were evaluated to pass.

Among the most relevant information it remarks the 9 test cases evaluated with "PASSED" and 0 with "FAILED".

The execution of the script took approximately 30 minutes. For each test case, the execution time can be consulted and in the section "TestCaseDetails" the execution log, with all actions performed by the test execution framework can also be seen.

5.4 Tests results

The Test Automation Framework followed an iterative development model and, at an early stage, the solution made possible to achieve automated testing on Athena applications, mainly in the BusinessSuite application. The development process undertaken then allowed to add features, improve the architecture and add components to the Test Automation Framework as it was being developed.

Therefore, and to better understand the impact of the tests made to the BusinessSuite application, some stats and some relevant results of the tests performed by the Test Automation Framework in the period between January 2013 and July 2013 will be presented.

Solution coverage of the BusinessSuite application

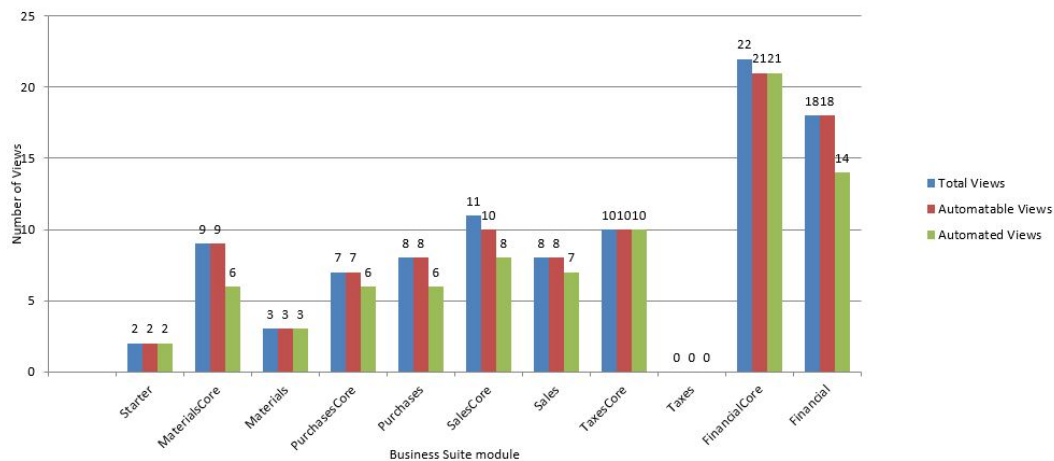


Figure 5.7: Solution coverage of the BusinessSuite application (BusinessSuite test results)

The test solution coverage for the BusinessSuite has an average of approximately 80%. The only module for which there are no automated views is the Taxes module. In this particular module there are no transaction views because the taxes are created by the country's

5. Case study: Business Suite

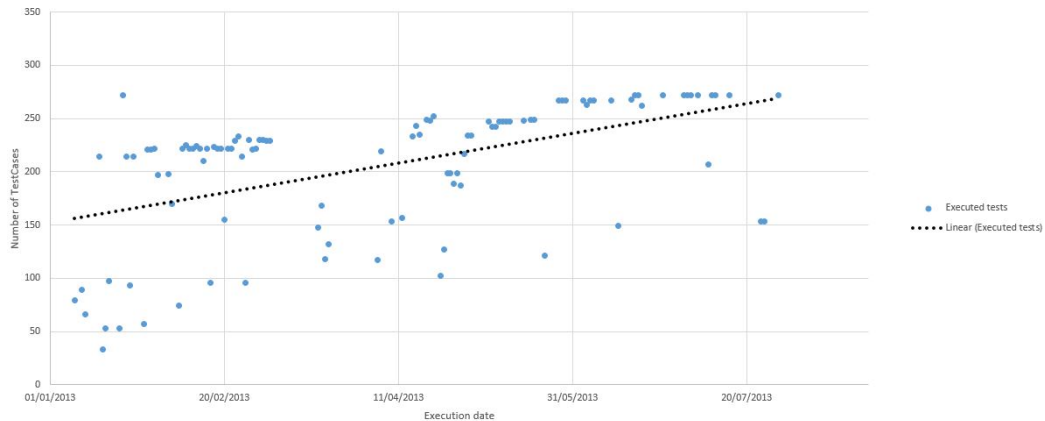


Figure 5.9: Evolution of the number of executed test cases(BusinessSuite test results)

The increase on executed test cases is a direct consequence of the increase on the covered views of the application. The features that most contributed to this increase were the addition of the two test case types, "COPYTO" and "COPYFROM", which allowed for more complex business processes to be designed and executed.

Passed and failed tests attending to the Service Releases dates

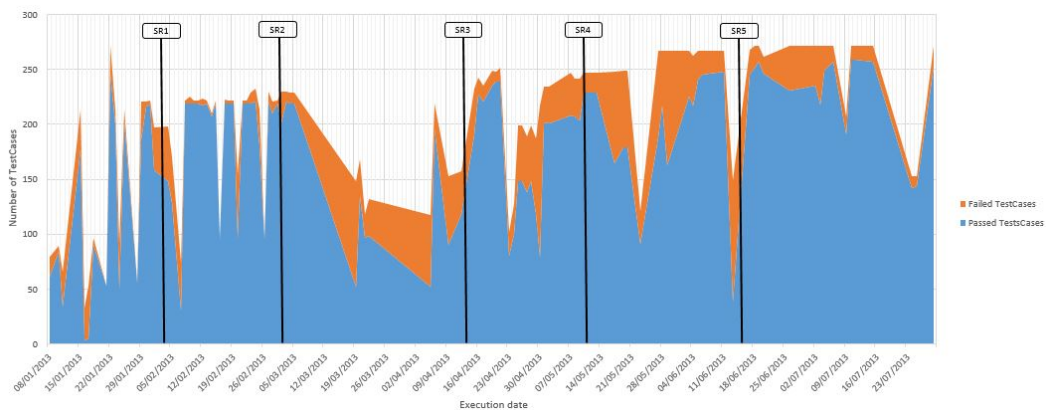


Figure 5.10: Passed and failed tests attending to the Service Releases dates(BusinessSuite test results)

The graphic shows the number of passed and failed tests for each day in which the

5. Case study: Business Suite

automated tests were executed. The Service Releases dates(SR1, SR2, SR3, SR4, SR5) are also highlighted, and they correspond to updates to the application that was made available to the clients. It can be seen that between Service Release dates the number of failed test cases decreases with the approach of each service release date. This is rather logic because in each service release new features and bugs are corrected and so it is expectable that those changes cause early, when introduced, inconstancy and instability on the application. The fluctuation on the number of executed test cases is due to the fact that some bugs would make some test cases fail and these fails would result in some other test cases not to be executed.

Failed tests and discovered bugs

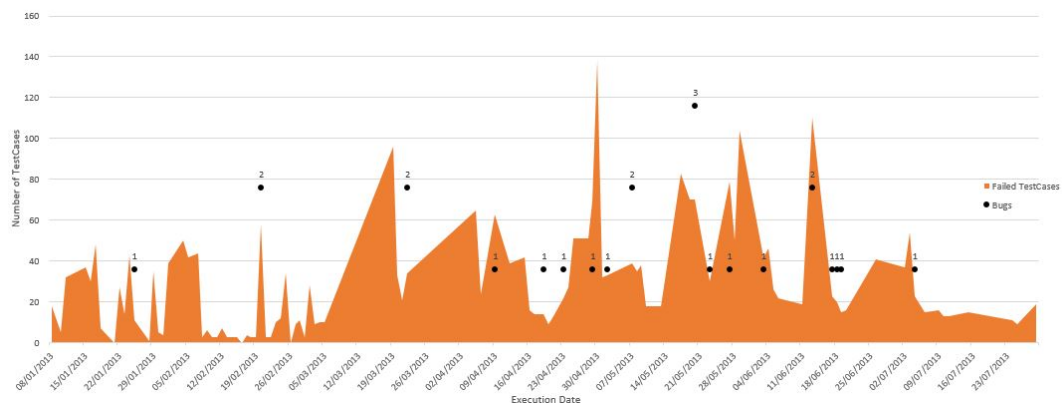


Figure 5.11: Failed tests and discovered bugs(BusinessSuite test results)

The Test Automation Framework proved to be successful in his higher mission to encounter bugs on the application, and so 24 bugs were found and reported exclusively with the results of the tests executed by the Test Automation Framework.

The first thing that stands out from an analysis to the graphic is that a higher number of failed test cases doesn't necessarily mean more bugs on the application. This is due to the fact that the tests were automatically executed day-by-day and the bugs were not corrected from one day to the other. Another explanation is that a single bug could be common to

5. Case study: Business Suite

many of the views and modules, which lead to a higher number of failed test cases.

Chapter 6

Concluding Remarks

Summary

This chapter presents the main conclusions of the work presented in the previous chapters. The goals satisfaction, answers to the research questions, difficulties and proposed future work are also discussed.

This document presented the Test Automation Framework developed to allow automated testing to be conducted in applications developed with the Athena Framework. The Business-Suite application case-study shows not only that the developed solution works as expected, but also that it proves to be a reliable solution available to the Primavera's Quality Assurance department. The solution's impact exceeds in a high extent the tests performed and reported bugs, drawing the attention of the organization for the existence of automated testing and its importance on the software development process.

6.1 Objectives satisfaction

The main goal for this project was to create a solution that would allow automated integration/acceptance testing to be performed to the products developed with the Athena Framework studying a model-driven automation framework approach and model alternatives.

To achieve the proposed goals, the following steps were followed:

6. Concluding Remarks

- A study on the Software Testing process. The test design techniques were also studied the test design techniques and automated testing approaches that would allow for a better understanding of the problem were also studied. For this task the International International Software Testing Qualifications Board (ISTQB) documents of the "Foundation level" were also used, as well as practical and theoretical examples of test automation frameworks.
- Familiarization with the Athena Framework and creation of a prototype application. This allowed a better understanding on how the framework worked and contact with the source-code. It would prove to be very valuable along the solution's development, since the Test Automation Framework was developed has a component of the Athena Framework itself.
- Study of the Athena Framework models that would allow the implementation of a model-driven Automation Framework. At this point, two different models were studied, the XAML model and Presentation Model.
- Implementation of a component and a tool that would allow the automatic generation of excel test specifications with the use of a model. At first, this component used the XAML model, but with the solution's development it was decided to use the Presentation Model.
- Creation of a "DataManagementTool" tool to allow the test designer to re-use test cases from older test specifications and to also allow an easy adaptation of the test cases to the test scenarios.
- Study of the White plugin. A plugin that would allow actions to be performed on the controls of Silverlight applications.
- Creation of a Generic execution engine, a single component that, by using the White plugin, would allow to perform the test actions on whichever view of whichever module of any Athena application. It allowed the reduction of duplicated code.

- Build a case study based on the Athena application BusinessSuite.

6.2 Research Questions: Answers

In this thesis we aimed at answering to three research questions. Now, we answer these questions:

1. **Is it possible to automate graphical user interface software tests in Silverlight interfaces?**

Yes, it is possible. To automate applications with Silverlight user interfaces, two requirements have to be met: Firstly it must be possible to have access to the Silverlight objects that compose the user interface of the application. Secondly, we need to be able to raise events on those same objects. Events that include mouse and keyboard inputs. In this thesis, the White plugin allowed for these two requirements to be met. However, there are some questions that have to be taken in consideration: The existence of the AutomationID property value for each of the Silverlight objects allow for an easy implementation of automation. Without this property value, it is possible to implement automation but it will come with a great cost. Another aspect to be taken in consideration is the existence of third-party Silverlight controls which, the majority, does not comply with the first requirement. If these third-party controls are heavily used, then, the automation coverage may be significantly reduced and automating the simplest test procedure impossible.

2. **Is it possible to automatically generate excel test specifications from the models of the Athena Framework?**

Yes, it is possible to automatically generate excel test specification from the models of the Athena Framework. This was achieved using a model-driven approach that uses the framework's Presentation model to generate up-to-date excel test specifications. For each of the views of the Athena application being tested, a tool executes an auto-

matic procedure which updates a database with the view's controls latest information. After, another automatic procedure is able to read each of the view's specifications from the database and generate an excel file on which test cases will be defined.

3. Will a model-based test solution present results that will prove it to be a reliable and effective tool, available to the Quality Assurance Department of Primavera Software Factory?

Yes, the developed model-based test solution presented results that prove it to be a reliable and effective tool, available to the Quality Assurance Department of Primavera Software Factory. The solution covers about 80% of the views of the BusinessSuite application, an application developed with the Athena Framework, and it is expectable that all other applications developed with the framework will have similar coverage.

The Test Automation Framework allows testing to be conducted, automatically, every day to the different lines of the Business Suite application, automatically evolving the test specifications to be up-to-date with each iteration of the application, even if these are daily iterations.

From January 2013 to July 2013, the tests executed by the solution resulted in 24 bugs found and reported. The solution was successful in his higher mission, to find bugs on the application as soon as possible in the SDP. The daily tests performed by the solution give, to the development team, a notion of how is the software quality almost immediately after the developments being made which means that errors can be corrected on the same stage they were introduced, improving productivity, time, costs and the quality of the software.

6.3 Difficulties and challenges

The dissertation was concluded with a huge feeling of accomplishment, but various difficulties and challenges were encountered along the way.

The dissertation thematic, Software Testing, was a novelty. Along my academic course, this

6. Concluding Remarks

theme was never taught. As such, it was difficult to assimilate all the processes and techniques involved in the process of Software Testing.

Technically, I've never had contact with .NET technologies and pretty much all technologies used in Primavera - Software Factory projects are developed using it. This required me to study and quickly adapt to the .Net technologies, mainly the programming language C# and the VisualStudio development tool.

But not all challenges were technological. In fact one of the greatest challenges was to develop a sense of awareness of the organization for the testing process, automated test execution and the value of the tests. Shortly after the organization realized that bugs were being reported in result of the tests performed by the Test Automation Framework, an increase in the interest of the organization about the daily tests execution and test results was noticed. When approaching the date of Service Releases this interest was somewhat notorious.

6.4 Future work

Despite having a Test Automation Framework that exceeds, by far, the initial expectations, there's still room for improvements on the features and on the framework itself:

- Improve the supported views of the Test Automation Framework. This will allow more business processes to be designed and automated, as well as improve the test solution value.
- Study the Test Automation Framework ability to adapt to the execution of tests in other interfaces' technologies other than Silverlight; for example, html5 interfaces.
- Ongoing maintenance of the solution. For the solution to be used continuously by the Quality Assurance Department, it will need continuous improvements and adaptations to the changes made on the Athena Framework.

6. Concluding Remarks

- Conducting automated testing in other applications being currently developed using the Athena Framework.

Bibliography

- [1] "A Vision of Next Generation MRP II," Scenario S-300-339 John Wiley and Sons, Inc., p. 707. 2. Wylie, L. (April 12, 1990). "A Vision of Next Generation MRP II," Scenario S-300-339. Gartner Group. 3
- [2] The Internet Encyclopedia, Volume 1. Bidgoli, Hossein (2004). The Internet Encyclopedia, Volume 1.
- [3] CMMI for Development: Guidelines for Process Integration and Product Improvement, 3rd Edition CMMI for Development: Guidelines for Process Integration and Product Improvement, 3rd Edition, Mary Chrissis, Mike Knorad, Sandra Shrum, Mar 10, 2011.
- [4] The Relationship of System Engineering to the Project Cycle "The Relationship of System Engineering to the Project Cycle," in Proceedings of the First Annual Symposium of National Council on System Engineering, Kevin Forsberg and Harold Mooz, October 1991: 57–65.
- [5] MANAGING THE DEVELOPMENT OF LARGE SOFTWARE SYSTEMS "MANAGING THE DEVELOPMENT OF LARGE SOFTWARE SYSTEMS", Royce, Winston.
- [6] Testing throughout the testing life cycle "Testing throughout the testing life cycle", Software testing Certification through ISTQB and ASTQB Exam, Certification questions, answers, tutorials and more.
- [7] Agile Alliance. Retrieved 14 June 2010 Manifesto for Agile Software Development, Beck, Kent. Agile Alliance. Retrieved 14 June 2010.

BIBLIOGRAPHY

- [8] Agile software development http://en.wikipedia.org/wiki/Agile_software_development
- [9] Certified Tester Foundation Level Syllabus. International Software Testing Qualifications Board, Version 2010.
- [10] Guide to the Software Engineering Body of Knowledge (SWEBOK). IEEE Computer Society, Alain Abran, James W. Moore, Pierre Bourque, Robert Dupuis, 2004 Version.
- [11] The GUISurfer tool: towards a language independent approach to reverse engineering GUI code. Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems, Silva, João Carlos and Silva, Carlos and Gonçalo, Rui D. and Saraiva, João and Campos, José Creissac, 2010
- [12] SmellSheet Detective: A Tool for Detecting Bad Smells in Spreadsheets SmellSheet Detective: A Tool for Detecting Bad Smells in Spreadsheets, Jacome Cunha, Joao Paulo Fernandes, Pedro Martinsy, Jorge Mendesy, Joao Saraiva.
- [13] The Economic Impacts of Inadequate Infrastructure for Software Testing. Prepared by: RTI for National Institute of Standards & Technology, Program Office Strategic Planning and Economic Analysis Group, may 2002
- [14] Automated Defect Prevention: Best Practices in Software Management. Kolawa, Adam; Huizinga, Dorota (2007).
- [15] Test Automation Frameworks. Carl Nagle(digital version)
- [16] On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension. Software Evolution, Leon Moonen and Arie van Deursen and Andy Zaidman and Magiel Bruntink, 2008
- [17] One evaluation of model-based testing and its automation. Proceedings of the 27th international conference on Software engineering, Pretschner, A. and Prenninger, W. and Wagner, S. and Kuhnel, C. and Baumgartner, M. and Sostawa, B. and Zolch, R. and Stauner, T., 2005

BIBLIOGRAPHY

- [18] Practical Model-Based Testing: A Tools Approach. Practical Model-Based Testing: A Tools Approach, Utting, Mark and Legeard, Bruno, 2007
- [19] Testing Experience. te testing experience, The Magazine for Professional Testers, March 2012(digital version)
- [20] Automatically Inferring ClassSheet Models from Spreadsheets. Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing, Cunha, Jacome and Erwig, Martin and Saraiva, Joao, 2010
- [21] Domain Specific Languages. Addison-Wesley, The Addison-Wesley Signature Series, Fowler, M. , 2010
- [22] HaLeX: A Haskell Library to Model, Manipulate and Animate Regular Languages. Proceedings of the ACM Workshop on Functional and Declarative Programming in Education, João Saraiva, September 2002
- [23] Microsoft Silverlight. <http://www.microsoft.com/silverlight/>
- [24] Inspect. [http://msdn.microsoft.com/en-us/library/windows/desktop/dd318521\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/dd318521(v=vs.85).aspx)
- [25] XAML overview. Microsoft Msdn, [http://msdn.microsoft.com/en-us/library/cc189036\(v=vs.95\).aspx](http://msdn.microsoft.com/en-us/library/cc189036(v=vs.95).aspx)
- [26] Microsoft, Visual Studio, Code Generation and T4 Text Templates. <http://msdn.microsoft.com/en-us/library/vstudio/bb126445.aspx>
- [27] White Home, White Plugin. <http://www.codeproject.com/Articles/289028/White-An-UI-Automation-tool-for-windows-applicatio>

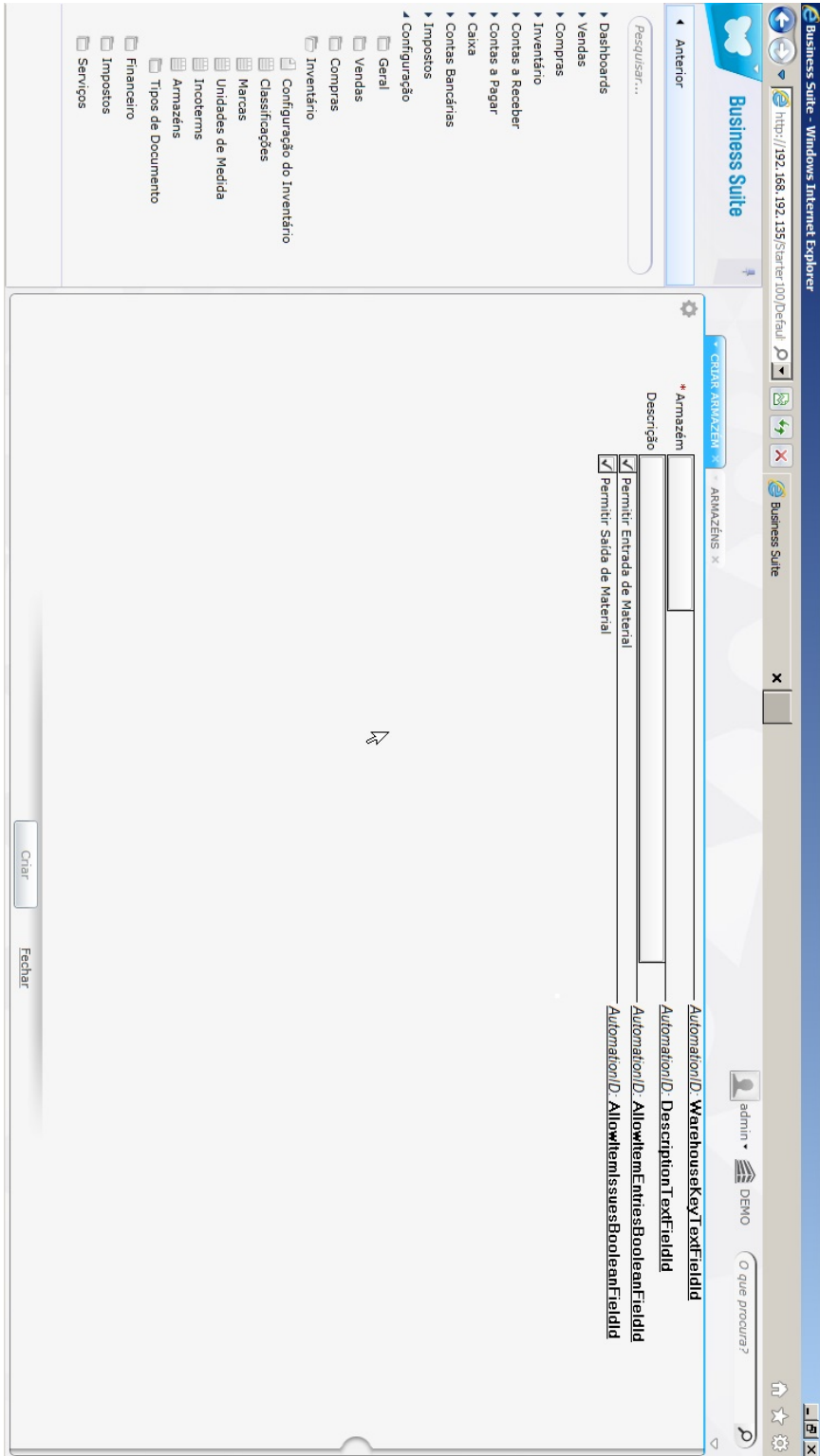
BIBLIOGRAPHY

Appendix A

User interface view "Warehouse"

A. User interface view "Warehouse"

A.1 "Warehouse" view



Appendix B

Test specification excel of the view

"Warehouse"

B. Test specification excel of the view "Warehouse"

B.1 Controls sheet, "Types"

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	WarehouseKey/TextField	ShortText														
2	Description TextField	Text														
3	AllowItemEntiesBooleanField	Boolean														
4	AllowItemIssuesBooleanField	Boolean														
5																
6																
7																
8																
9																
10																
11																
12																
13																
14																
15																
16																
17																
18																
19																
20																
21																
22																
23																
24																
25																
26																
27																
28																
29																
30																
31																
32																
33																
34																
35																
36																

B.2 Main sheet, "Warehouse"

	A	B	C	D	E	F	G	H	I	J
1	TestCaseId	Enabled	TestType	TaskFriendlyName	TaskId	Polarity	LoadInitialValues	ListForEdition	OpenFromList	FileName
2	TC1	TRUE	OpenApplication	OpenApplication	OpenApplication	Positive	FALSE	FALSE	FALSE	iexplore.exe
3	TC2	TRUE	Login	Login	Login	Positive	FALSE	FALSE	FALSE	iexplore.exe
4	TC3	TRUE	Insert	Armazens	RES_MenuCaption_Task_Warehouses	Positive	FALSE	FALSE	TRUE	iexplore.exe
5										
6										
7										
8										
9										
10										
11										
12										
13										
14										
15										
16										
17										
18										
19										
20										
21										
22										
23										
24										
25										
26										
27										
28										
29										
30										
31										
32										
33										
34										
35										

B. Test specification excel of the view "Warehouse"

	K	L	M	N	O	P	Q	R	S
Arguments	Name	Type	BrowserType	Username	Password	NewPassword	AuthenticationFailedTextBlock	WorkspaceItem	
1	http://192.168.192.130/Starter100/Default_tests.aspx	Business Suite	Silverlight IE	admin	a		FALSE	DEMO	
2	http://192.168.192.130/Starter100/Default_tests.aspx	Business Suite	Silverlight IE	admin	a		FALSE	DEMO	
3	http://192.168.192.130/Starter100/Default_tests.aspx	Business Suite	Silverlight IE	admin	a		FALSE	DEMO	
4	http://192.168.192.130/Starter100/Default_tests.aspx	Business Suite	Silverlight IE	admin	a		FALSE	DEMO	
5									
6									
7									
8									
9									
10									
11									
12									
13									
14									
15									
16									
17									
18									
19									
20									
21									
22									
23									
24									
25									
26									
27									
28									
29									
30									
31									
32									
33									
34									
35									
36									

B. Test specification excel of the view "Warehouse"

	T	U	V	W	X	Y	Z
1	RememberOrendentials	RememberWorkspace	NaturalKey	WarehouseKeyTextFieldId	DescriptionTextFieldId	AllowItemEntitesBooleanFieldId	AllowItemIssuesBooleanFieldId
2	FALSE	FALSE					
3	FALSE	FALSE					
4	False	False		GM	GM	True	False
5							
6							
7							
8							
9							
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							
20							
21							
22							
23							
24							
25							
26							
27							
28							
29							
30							
31							
32							
33							
34							
35							
36							

