FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

IPBrick - Plataforma de testes automáticos

Sérgio Manuel da Costa Reis

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Orientador: Mário Jorge Rodrigues De Sousa

Proponente: Luís Miguel Ramalhão Ribeiro

28 de Julho de 2017

Resumo

Este documento é referente a um projeto de Dissertação , tendo a fase de investigação ocorrido em ambiente académico e a fase de implementação foi desenvolvida em ambiente empresarial. A instituição académica trata-se da Faculdade de Engenharia da Universidade do Porto e, em termos empresariais, a instituição de acolhimento foi a IPBRICK SA, uma empresa fabricante e distribuidora especializada em Soluções de Comunicação Corporativas, que tem como dois grandes produtos o IPortalDoc e a IPBrick (Sistema Operativo de Rede).

O objectivo deste trabalho consistiu no desenvolvimento de uma Plataforma de teste. Esta plataforma foi conseguida através da escrita e programação de casos de teste, que já se encontravam definidos e executados de uma forma manual pelos programadores da IPBrick SA. Posteriormente, foi realizada a integração destes testes numa plataforma que foi configurada para integrar, não só os testes automáticos, mas também onde fosse possível fazer a sua gestão, configurá-los e analisar relatórios provenientes dos testes executados, tudo isto através de uma *interface* gráfica.

A metodologia de desenvolvimento consistiu num levantamento inicial de técnicas de teste de *software* que pudessem ser aplicadas ao sistema IPBrick, seguido de um estudo teórico e experimental de um conjunto de ferramentas cujas funcionalidades permitissem a aplicação das técnicas estudadas. Seguiu-se uma fase de implementação, onde todas as ferramentas foram integradas de modo a possibilitar que todo o processo de teste e armazenamento de resultados ficasse automatizado. Este módulo é instalado numa máquina central que executa os testes em várias instâncias que são controladas remotamente através da Plataforma de Testes.

Foram realizadas várias experiências no sistema alvo dos testes. Os resultados obtidos foram validados, de acordo com os resultados obtidos pelos programadores quando executavam o mesmo caso de teste de uma forma manual.

O projeto terminou com o cumprimento de todos os objectivos iniciais.

Abstract

This document refers to a Dissertation project in which the research phase occurred in an academic environment and the implementation phase was developed in a enterprise environment. In an academic environment, it is the Faculty of Engineering of the University of Porto, and in enterprise terms, the host institution was IPBRICK SA, a manufacturer and distributor specialized in Corporate Communication Solutions and has two major products IPortalDoc and To IPBrick (Network Operating System). The objective of this work was the development of a Test Platform. This platform was achieved through the writing and programming of test cases, which were already defined and executed in a manual way by the programmers of IPBrick SA, and a later integration of these tests in a platform that was configured to integrate not only the automatic tests but Also where it is possible to manage them and configure them and analyze reports from the tests performed, all through a graphical interface.

The development methodology consisted of an initial survey of software testing techniques that could be applied to the IPBrick system, followed by a theoretical and experimental study of a set of tools whose functionalities allowed the application of the studied techniques. In the implementation phase, the tools were integrated to enable the entire process of testing and storing results to be automated. This module is installed on a central machine that runs the tests in several instances that are controlled remotely through the Test Platform.

Several experiments were run by running the tests against the target system. The results obtained were validated, according to the results obtained by the programmers when they executed the same test case in a manual way. The project ended with the fulfillment of all the initial objectives.

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

CSS	Cascade Style Sheets
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
FEUP	Faculdade de Engenharia da Universidade do Porto
FQDN	Fully Qualified Domain Name
HTML	HyperText Markup Language
IP	Internet Protocol
ISP	Internet Service Provider
LAN	Local Area Network
MIEEC	Mestrado Integrado em Engenharia Eletrotécnica e de Computadores
PHP	Hypertext Preprocessor
PPTP	Point-to-Point Tunneling Protocol
QA	Quality assurance
SVG	Scalable Vector Graphics
UCoIP	Unified Communications Open Interoperability Program
URL	Uniform Resource Locator
VoIP	Voice over Internet Protocol
VPN	Virtual Private Network
XML	eXtensible Markup Language
W3C	World Wide Web Consortium
WWW	*World Wide Web*

Capítulo 1

Introdução

1.1 Contexto

Este documento realiza-se no âmbito da Dissertação de Mestrado do 5º ano do Mestrado Integrado em Engenharia Eletrotécnica e de Computadores, tem como título "IPBRick – Plataforma de testes automáticos" e vai ser realizada em ambiente empresarial, nas instalações da própria empresa, que está situada em Vila Nova de Gaia na Avenida da República, nº 755.

A IPBRICK SA é uma empresa fabricante e distribuidora especializada em Soluções de Comunicação Corporativas e tem como dois grandes produtos o IPortalDoc e a IPBrick (Sistema Operativo de Rede). O Sistema Operativo de Rede IPBRICK sofre constantemente novos *updates* ou são desenvolvidas novas versões que implicam um processo de testes de aceitação. Estes testes têm por função verificar o sistema em relação aos seus requisitos originais. Até ao momento, os testes são realizados de forma manual, em que são muitas vezes bastante demorados e em diversas situações acabam por ser repetitivos. A execução e repetição manual de um vasto procedimento de testes é uma tarefa dispendiosa e cansativa, ou seja, o procedimento atual de testes, de certa forma, permite que o *software* possa ser disponibilizado com erros, o que acaba por trazer consequências negativas, tanto nos prazos de entrega bem como na qualidade final do *software*. Esta dissertação vai incidir no projeto e no desenvolvimento de uma plataforma dos testes a realizar automaticamente e, no final, a obtenção de um relatório com os respetivos resultados.

1.2 Motivação

Hoje em dia vivemos num mundo extremamente competitivo e, como tal, a qualidade, a eficácia e a eficiência são fatores importantes para o sucesso de qualquer organização. Neste sentido, a execução de testes automáticos tem vindo gradualmente a assumir um papel cada vez mais importante nas equipas de quality assurence (QA) das organizações, o que possibilita, por exemplo, a redução do tempo da execução dos testes complexos ou a eliminação de falhas inerentes a uma interação manual e repetitiva. A IPBRICK, como empresa dinâmica e competitiva que é, propôs esta dissertação, na qual eu tenho o privilégio de participar, e tem como objetivo automatizar grande

parte dos seus testes, aumentando assim a eficiência e eficácia, garantindo uma maior qualidade do seu *software*. Ter a responsabilidade de participar diretamente na qualidade do *software* é um fator que me deixa extremamente motivado, uma vez que tenho a oportunidade de intervir e desenvolver competências numa área tão valorizada como esta.

1.3 Objetivos

A IPBRICK possui uma *checklist* com a descrição dos testes e tarefas que são executadas manualmente com a intenção de testar todas as aplicações e *software* desenvolvidos na empresa. Alguns destes testes são executados recorrentemente sempre que é feito um novo *update* ou um lançamento de uma nova versão do *software* principal (IPBRICK SO). O principal objetivo da minha dissertação é analisar e entender como estes testes são executados para então escolher as ferramentas que melhor se adequam à automatização destes Testes Obrigatórios, que até ao momento são executados manualmente e, por fim, gerar um relatório dessa bateria de testes executada automaticamente.

1.4 Estrutura da Dissertação

Este documento é composto por seis capítulos: Introdução; Revisão Bibliográfica; Descrição do Sistema IPBRICK; Problema e Requisitos;Implementação da Solução ;Conclusão e Trabalho Futuro.

No capítulo 1, Introdução, é apresentada a motivação e o enquadramento, sendo ainda feita uma exposição da temática relativamente à área de teste e qualidade de software. Neste mesmo capítulo são ainda definidos os objetivos deste projeto e é feita uma descrição do problema a resolver.

No capítulo 2, Revisão Bibliográfica, são apresentadas e explicadas uma série de conceitos e metodologias de teste de software. É ainda feita uma análise as várias ferramentas de teste, e outras ferramentas auxiliares que iriam integrar a Plataforma de testes.

Segue-se o capítulo 3, Descrição do Sistema IPBRICK, onde é apresentado o sistema IPBRICK com o objectivo de se conhecer o sistema que será alvo dos testes automáticos.

O capítulo 4, Problema e Requisitos, existe com o objectivo de apresentar o problema e mencionar os requisitos que foram levantados para a Plataforma de testes.

No capítulo 5, Implementação da Solução, visa apresentar a arquitetura da solução, passando por uma abordagem mais aprofundada de como esta funciona, das suas funcionalidade, e das técnicas e padrões utilizados durante o seu desenvolvimento. É também apresentada e explicada como esta solução foi configurada e como pode ser utilizada.

Este documento encerra com o capítulo 6, Conclusão e Trabalho Futuro, onde são discutidas as conclusões que foram possíveis retirar no final, sendo ainda enunciadas uma série de melhorias e funcionalidades adicionais que aumentariam o potencial da solução desenvolvida.

Capítulo 2

Revisão Bibliográfica

Neste capítulo é descrito o estado de arte da tecnologia relacionada com o tema, recorrendo a livros, artigos científicos e outros relatórios provenientes de fontes académicas que se dedicaram ao estudo e ensino acerca de testes automáticos e qualidade de *software*.

2.1 Introdução

O sucesso de qualquer produto de *software* ou aplicação está muito dependente da sua qualidade. Hoje em dia, o teste é visto como a melhor maneira de assegurar a qualidade de qualquer produto.

Os testes de qualidade podem reduzir em grande escala a necessidade de correção dos projetos, que é um fator que aumenta os orçamentos e atrasa a programação.

A necessidade de testes tem vindo a aumentar, uma vez que as empresas enfrentam uma enorme pressão para desenvolver aplicações com prazos de tempo muito reduzidos. O teste surge então como um método que leva à avaliação da qualidade do produto ou serviço de *software*. É também um processo que verifica a exatidão de um produto e avalia quão bem ele funciona. Assim, o processo de teste identifica os defeitos de um produto seguindo um método de comparação, onde o comportamento e o estado de um determinado produto é comparado com um conjuntos de padrões que incluem especificações, contratos e versões anteriores do produto.

O teste de *software* é um processo incremental e iterativo para detetar uma incompatibilidade, um defeito ou um erro [1].

Sendo assim, o presente capítulo começa por uma abordagem geral acerca de testes do *software*, aprofundando gradualmente sobre testes automáticos e os seus métodos de aplicação.

2.2 Testes de Software

O teste de *software* é um procedimento de execução de um programa ou sistema com a intenção de encontrar falhas. É um processo de confirmação de que o produto fabricado pelos programadores é um produto de qualidade, servindo para assegurar que o produto está a trabalhar

de acordo com os requisitos técnicos estabelecidos e satisfaz as necessidades dos clientes. Um bom teste de *software* deve ter seguintes as características:

- Deve ter uma probabilidade de encontrar um erro que ainda não foi descoberto;
- Não deve ser redundante;
- Um teste bem sucedido deve revelar um erro ainda desconhecido;
- Não deve ser nem muito simples nem muito complexo;
- Verifica se o sistema faz o que é esperado que faça;
- Verifica se o sistema está adequado à sua finalidade;
- Verifica se o sistema atende aos requisitos e é executado com sucesso no meio previsto.

A finalidade dos testes é deste modo a verificação, validação e deteção do erro, a fim de encontrar problemas, sendo assim possível corrigi-los [2].

2.2.1 Importância do Teste de *Software*

Na sociedade moderna, os *softwares* passaram a ter um papel vital na condução e execução das organizações, sendo em alguns casos parte intrínseca dos seus negócios. A quantidade de *software* incorporado em produtos de consumo duplica a cada dezoito meses. Com a crescente demanda por novas tecnologias por parte das empresas e pela sociedade em geral, esta importância tende a multiplicar-se nos próximos anos. Perante este cenário e dada a confiança que os clientes depositam no *software*, esta passa a ser um fator crucial para os fabricantes, que se vêem obrigados a ter a capacidade de produzir *softwares* de alta qualidade e em prazos cada vez menores. [3]

O teste de *software* não é algo novo, no entanto, ainda hoje, as empresas preferem desenvolver sistemas sem uma metodologia baseda em qualidade. Esta escolha é muitas vezes fundamentada com o argumento de que testar *software* tem um elevado custo associado. A verdade é que o custo do defeito é progressivo, ou seja, encontrar o defeito na fase de utilização custa cem vezes mais do que encontrar o defeito na fase de engenharia. Deste modo, utilizar o teste reduz custo e não o aumenta [4].

2.2.2 Princípios

A boa prática de testes de *software* devem seguir alguns princípios fundamentais. Princípios são a regras ou métodos de ação que devem ser seguidos aquando a execução de testes. Seguem-se os diferentes princípios de testes:

- Testar um programa para tentar fazê-lo falhar: O teste é o processo de execução de um programa com o intuito de encontrar erros. Deve expor-se as suas falhas para tornar processo de teste mais eficaz;

- Começar a testar cedo: Isto ajuda na correção de muitos erros nas fases iniciais de desenvolvimento, reduzindo a necessidade de repetição do trabalho quando são encontrados erros na fase inicial;

- O teste é dependente do contexto: O teste deve ser apropriado e diferente para diferentes pontos no tempo;

- Definir um Plano de Teste: O plano de teste geralmente descreve o âmbito do teste, os objetivos do teste, a estratégia do teste, o meio ambiente do teste, resultados do teste, os riscos e mitigação, o cronograma, os níveis de testes a serem aplicados, os métodos, as técnicas e as ferramentas a serem usados. O plano de teste deve atender de forma eficiente às necessidades de uma organização, bem como dos clientes;

- *Design* de casos de teste eficazes: O caso de teste deve ser determinado de maneira mensurável, de modo a que os resultados dos testes sejam inequívocos;

- Teste para condições válidas e inválidas: Além de entradas válidas, devemos também testar o sistema para condições inválidas ou inesperadas;

- O teste deve ser feito por pessoas diferentes: Tendo em conta as diferentes finalidades abordadas a diferentes níveis, a realização de testes deve ser feita por diferentes pessoas, usando diferentes técnicas e níveis;

- Fim do teste: O teste tem de ser parado a certo momento. O teste pode ser interrompido quando o risco está sob um determinado limite ou se existir limitação.

2.2.3 Conceitos

Existem alguns conceitos importantes que compõem o universo de testes de *software*. Por mais simples que eles possam parecer, em muitos casos eles confundem-se devido à ambiguidade de sentidos que estes podem assumir na linguagem corrente. Porém, dentro do escopo de engenharia de *software*, cada termo que será citado abaixo tem o seu próprio significado [5].

2.2.3.1 Erro

Erro é uma manifestação concreta de um defeito num artefacto de software. Diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução de um programa constitui um erro. [5].

2.2.3.2 Defeito

Falha é o comportamento operacional do software diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversos erros e alguns erros podem nunca causar uma falha. [5].

2.2.3.3 Falha

Uma falha é um comportamento operacional do *software* diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversos defeitos e alguns defeitos podem nunca causar uma falha [5].

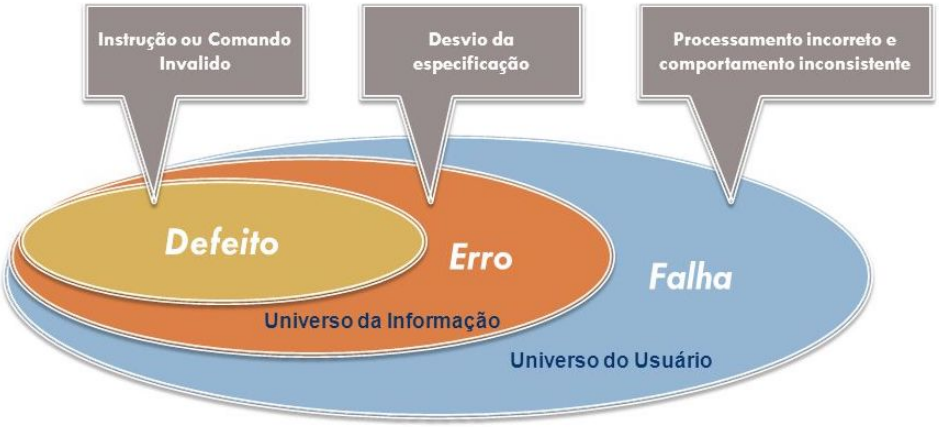

Figura 2.1: Erro vs Defeito vs Falha

2.2.3.4 Verificação

Verificar um *software* consiste em aferir que o produto é inteiramente consistente, ou seja, significa garantir que o *software* atende à especificação de requisitos. A grande maioria dos testes consiste em testes de verificação, em que o produto é verificado em circunstâncias especificadas para garantir a saída esperada [5].

2.2.3.5 Validação

Validar um *software* consiste em garantir que o produto está de acordo com as expectativas do cliente ou utilizador. Validar vai além de apenas verificar se o sistema está de acordo com as especificações, uma vez que também garante que o sistema realiza realmente aquilo que o usuário final espera que ele realize [5].

2.2.3.6 Depuração

Depurar é um processo posterior ao teste. Ocorre após o teste encontrar alguma falha no sistema ou no *software* testado e consiste numa análise do código fonte ou na especificação do *software* com o objectivo de encontrar e corrigir a falta ou defeito que gerou esse erro [5].

2.3 Tipos e níveis de Teste de Software

Os níveis de testes incluem diferentes metodologias que podem ser usadas durante a realização de testes de *software*. Os principais níveis de testes de *software* são os Testes Funcionais e os Testes Não funcionais.

2.3.1 Funcionais

Os testes funcionais baseiam-se nas especificações do *software* que vai ser testado. A aplicação é testada através da inserção de entradas, as saídas são examinadas e estes devem estar em conformidade com o que é esperado [6].

Existem cinco passos que estão envolvidos durante a fase de testes funcionais de um *software*:

- Determinação da funcionalidade que a aplicação a testar realiza;
- Criação de dados de teste com base nas especificações da aplicação;
- Saída com base nos dados do teste e especificações da aplicação;
- Escrita de cenários de teste e execução de casos de teste;
- Comparação dos resultados reais e esperados com base nos casos de teste executados

Os testes de *software* estão envolvidos em todas as fases do ciclo de vida do *software*, mas a forma como os testes são realizados em cada uma das fases do desenvolvimento de *software* tem uma natureza e um objetivo diferente. A figura 2.2 representa essas fases do ciclo de vida do *software* e onde o são aplicados os diferentes tipos de teste.

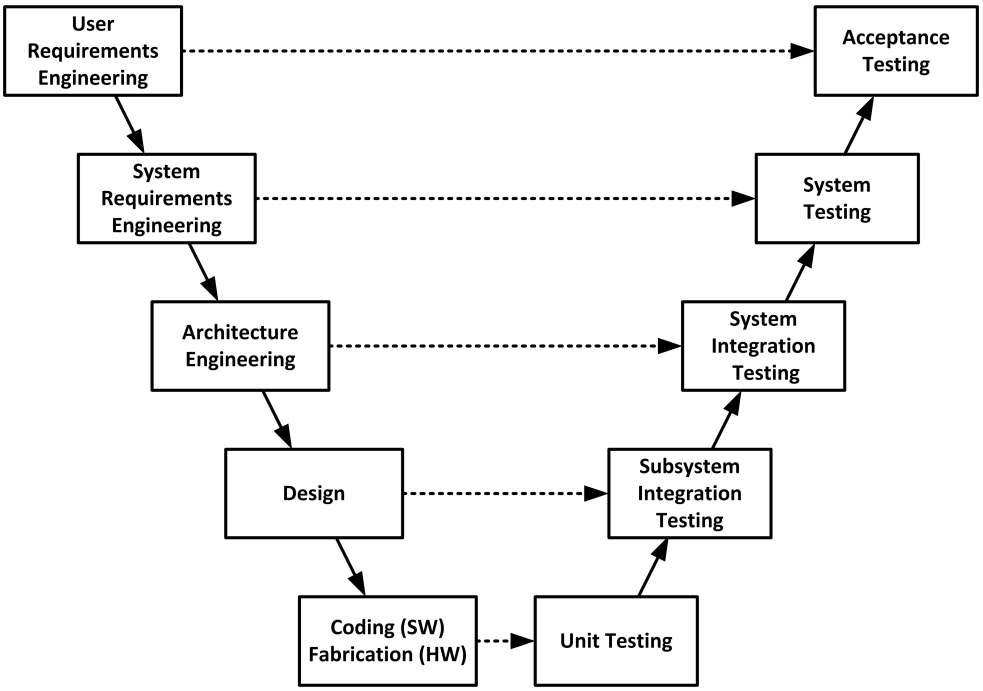

Figura 2.2: Representação da aplicação dos diferentes testes durante o ciclo de vida de um projeto

2.3.1.1 Testes de Unidade

O teste de unidade é um teste baseado no código que é realizado pelos programadores. Este teste é feito principalmente para testar unidades do código separadamente. Os testes de unidade podem ser executados para pequenas unidades de código, geralmente nunca maiores que uma classe. O objetivo do teste de unidade é isolar cada parte do programa e mostrar que as partes individuais estão corretas em termos de requisitos e funcionalidades [6].

2.3.1.2 Testes de Integração

Os testes de integração são definidos como testes de partes combinadas de um *software* para determinar se as mesmas funcionam corretamente. O teste de integração pode ser feito de duas maneiras: teste de integração de *Bottom-up* e teste de integração de *Top-down*. A integração de *Bottom-up* começa com testes de unidade, seguido de testes de combinações progressivas de elevado nível de unidades denominadas módulos ou compilações. No que diz respeito à integração de *Top-down*, os módulos de nível mais elevado são testados primeiro e progressivamente, sendo que os módulos de nível inferior são testados depois [6].

2.3.1.3 Testes de Sistema

Testes de sistema testam o sistema como um todo. Depois de todos os componentes serem integrados, o *software* é testado rigorosamente como um todo para ver se está de acordo com

as normas de qualidade especificadas. Este tipo de teste é realizado por uma equipa de testes especializada. Os testes de sistema são importantes porque:

- São o primeiro passo no desenvolvimento do ciclo de vida de *software*, onde o *software* é testado como um todo;
- O *software* é testado exaustivamente para verificar se está de acordo com as especificações funcionais e técnicas;
- O *software* é testado num ambiente que é muito próximo do ambiente de produção onde o *software* irá ser implantado.

Os testes de sistema permitem-nos testar, verificar e validar tanto os requisitos de negócio como a arquitetura do *software* [6].

2.3.1.4 Testes de Aceitação

Este é sem dúvida o tipo de teste mais importante. Os testes de aceitação não são destinados apenas à deteção de erros simples de ortografia, erros de estética ou lacunas de *interface*, mas também à deteção de eventuais erros na aplicação que irá resultar em falhas do sistema ou grandes erros na aplicação. Através da realização de testes de aceitação num *software*, a equipa de testes irá deduzir como o *software* se irá comportar [6].

2.3.1.5 Testes de Regressão

Sempre que existe uma mudança num *software*, é bastante possível que outras áreas dentro do mesmo *software* sejam também alteradas não intencionalmente. Os testes de regressão são realizados para verificar se um *bug* que foi corrigido não resultou noutra violação de funcionalidade. A intenção dos testes de regressão é assegurar que uma mudança, nomeadamente a correção de um *bug*, não resulta noutra falha de *software* [6].

2.3.1.6 Testes Alfa

Este teste corresponde à primeira fase de testes. Os testes alfa são realizados entre duas equipas: a de programadores e a de qualidade. Os testes unitários, os testes de integração e os testes de sistema, quando combinados, são conhecidos como teste alfa, onde o objetivo é ter em conta aspetos como erros ortográficos e *links* mal direcionados. São também executados testes de *software* em máquinas menos potentes para percecionar a fluidez do *software* com menos recursos [6].

2.3.1.7 Testes Beta

Este teste tem lugar depois do teste alfa ser realizado com sucesso. Nos testes beta é uma amostra do público-alvo que testa o *software*. O teste beta é também conhecido como o teste de pré-lançamento. As versões do teste beta do *software* são idealmente distribuídos por um grande

público na *web*, de forma a dar ao programa um teste do "mundo real"e, em parte, para fornecer uma visão sobre o próximo lançamento [6].

2.3.2 Não Funcionais

Nos testes não funcionais é a qualidade das características do sistema que é testada. O conceito de teste não funcional refere-se a aspetos do *software* que podem não estar relacionados com uma função específica do *software* ou com uma ação específica do usuário. Estes testes têm como objetivo testar questões relacionadas com segurança, fluidez ou *performance* [7].

2.3.2.1 Testes de *Performance*

O teste de *performance* ou desempenho testa o *software* para determinar quão rápido uma certa parte de um sistema é executada sob uma carga de trabalho específica. Este teste pode ter finalidades diferentes tais como demonstrar que o sistema corresponde a certos critérios de desempenho. Pode também comparar dois sistemas para encontrar o que tem melhor desempenho [7].

2.3.2.2 Testes de Carga

Um teste de carga é geralmente utilizado para compreender o comportamento da aplicação sob uma carga específica. O teste de carga é realizado para determinar o comportamento de um sistema sob as condições normal e de pico. Este teste ajuda a identificar a capacidade máxima de operação de uma aplicação, bem como os seus *bottle necks* e determinar qual o elemento que está a causar a degradação do sistema [7].

2.3.2.3 Testes de *Stress*

Os testes de *stress* implicam testar o sistema para além da sua capacidade de operação normal, muitas vezes a um ponto de rutura, de modo a observar os resultados. É uma forma de teste que é usado para determinar a estabilidade de um dado sistema. Coloca maior ênfase na robustez, disponibilidade e manipulação de erro sob uma carga pesada e não naquilo que seria considerado o comportamento correto em circunstâncias normais. O objetivo destes testes é garantir que o *software* não falha em condições de recursos computacionais insuficientes como memória ou espaço em disco [7].

2.3.2.4 Testes de Usabilidade

Os testes de usabilidade implicam testar a facilidade com que as *interfaces* de utilizador podem ser usadas. Estes testam se o produto construído é *user-friendly* ou não [7].

2.3.2.5 Testes de Segurança

Os testes de segurança envolvem testar um *software* a fim de identificar eventuais falhas e lacunas de segurança e vulnerabilidade. É um processo que tem como objetivo determinar se um sistema de informação de dados está protegido e mantém a funcionalidade como pretendido [7].

2.3.2.6 Testes de Portabilidade

O teste de portabilidade refere-se ao processo de testar a facilidade com que um componente de *software* de computador ou aplicação pode ser movido de um ambiente para outro. Os resultados são medidos em termos do tempo necessário para mover o *software* e completar as atualizações e documentação [7].

2.4 Métodos de Teste

2.4.1 Caixa-Preta

É chamado de teste Caixa-Preta ou *Black-Box* à técnica de teste que não tem qualquer conhecimento sobre o funcionamento nem da lógica do *software* a testar. A pessoa que realiza os testes desconhece a arquitetura do sistema e não tem acesso ao código-fonte. Normalmente, durante a execução de um teste Caixa-Preta, a pessoa que realiza o teste, interage com a *interface* do usuário do sistema, fornecendo entradas, e examina as saídas sem saber como e onde as entradas são processadas. O objetivo é testar o quão bem o produto está em conformidade com o requisito esperado.

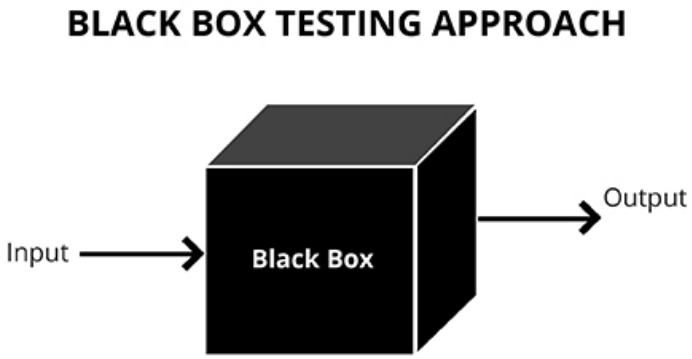

Figura 2.3: *Black-Box Testing*

O método de testar com *Black-Box* usa algumas técnicas como *Equivalent Partitioning, Boundary Value Analysis, Cause-Effect Graphing Techniques* e *Comparison Testing*. Utilizar este método trás vantagens pois não é necessário conhecer a lógica nem o código-fonte do programa para o testar e a pessoa que testa e o programador não estão dependentes um do outro. No entanto, os testes podem tornar-se difíceis de projetar quando as especificações não são claras.

2.4.2 Caixa-Branca

Estes tipos de testes são altamente eficazes na deteção e resolução de problemas, pois os erros podem ser encontrados antes de causarem problemas. O teste Caixa-Branca ou *White-Box* é um processo que atribuí valores às entradas do sistema, verifica como o sistema processa essas entradas e como gera a saída. Os testes de *White-Box* são aplicáveis em níveis de integração, de unidade e de teste de sistema. Este é um método de teste seguro que pode ser utilizado para validar a implementação, *design* e funcionalidade do código do programa que é testado.

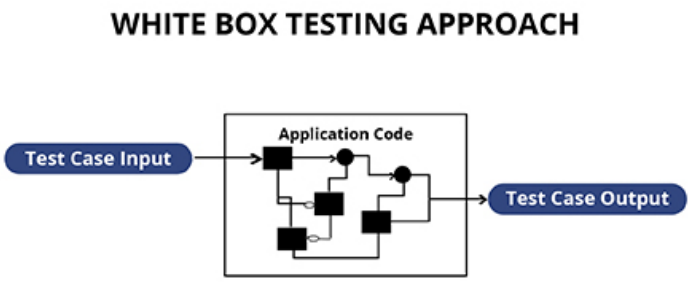

Figura 2.4: *White-Box Testing*

À semelhança do método de Caixa-Preta este método de Caixa-Branca também usa algumas técnicas, nomeadamente *Basis Path Testing, Loop Testing* e *Control Structure Testing*.

A utilização deste método traz vantagens como a capacidade de revelar todos os erros ocultos no código e permite que as operações lógicas sejam ensaiadas. Contudo, traz também uma grande desvantagem dado que implica a necessidade de um conhecimento pormenorizado da estrutura interna do código. Isto é um requisito essencial que é necessário para a pessoa que vai realizar este tipo de testes. O tempo necessário até que a pessoa esteja apta a utilizar este método de teste é bastante elevado, o que provoca um consequente aumento de custos.

2.4.3 Caixa-Cinza

O teste Caixa-Cinza ou *Grey-Box* é um método de teste realizado com informações limitadas sobre a funcionalidade interna do sistema. Quem utiliza o método *Grey-Box*, tem acesso aos documentos de *design* detalhados, juntamente com informações sobre os requisitos.

Algumas das técnicas que este método utiliza são *Regression Testing, Pattern Testing, Orthogonal Array Testing* e *Matrix testing*. Este método combina as vantagem dos métodos de Caixa-Preta e Caixa-Branca.

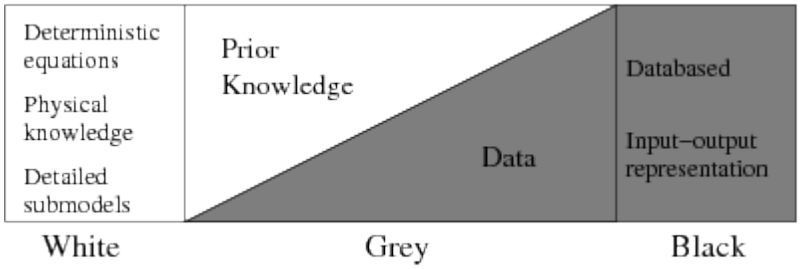

Figura 2.5: Diagrama *Grey-Box*

2.5 Automação de Testes

Neste capítulo, serão estudados os principais casos e aplicações para automação de testes, assim como algumas ferramentas largamente utilizadas nesses processos.

2.5.1 Definição

Automação de testes, por definição, consiste na utilização de *softwares* externos (que não façam parte do *software* que está a ser testado) para controlar a execução dos testes e a comparar os resultados encontrados com os resultados esperados. Através desse processo podem-se automatizar algumas tarefas de teste repetitivas, porém necessárias, que fazem parte do fluxo de teste de um *software* ou adicionar novos testes que seriam muito trabalhosos e, consequentemente, custosos para serem executados manualmente.

2.5.2 Quando Não Automatizar

A automação de testes tem como objetivo reduzir esforços nas tarefas repetitivas, pelo que nem todos os testes devem ser automatizados. Antes de se inserir esse processo num projeto deve-se analisar qual é o volume de testes que o *software* necessita e o tempo que é gasto para executar essas baterias de testes. Se o projeto em questão tiver uma complexidade muito pequena de modo que o volume de teste para aferir a qualidade do sistema é muito baixo, o esforço para inserir essa metodologia de testes pode ser mais custoso do que a manutenção dos testes manuais. Em adição a este aspecto, o tempo para a execução dos testes automáticos não seriam, nesse caso, inferiores aos manuais.

Outro fator importante é que o teste automatizado, assim como qualquer outro *software*, comunica com o *software* que está a ser testado através de uma *interface*. Se essa *interface* sofre constantes modificações, os testes também necessitam de alterações, sendo que este é o pior cenário para a automação de testes. Em casos desta natureza, a automação deve ser evitada pois os testes automáticos serão constantemente alterados de forma manual, contrariando totalmente os princípios do processo em si [5].

2.5.3 Quando Automatizar

A automação de teste deve ser realizada considerando alguns aspetos de um *software*.

Quando o projeto é grande e importante, testes como por exemplo os de regressão são muito dispendiosos a nível de tempo para serem executados manualmente e são muito suscetíveis ao erro humano pois são tarefas repetitivas. Ao automatizar reduz-se drasticamente a possibilidade de o sistema ir para produção com a inserção de um novo defeito numa funcionalidade antiga. Além disso, o ganho em tempo de execução dos testes geralmente justifica a sua utilização.

Outro aspecto importante é quando o projeto exige testes frequentes nas mesmas áreas e as exigências não mudam com regularidade. Nesses casos, é vantajoso proceder à automação dos testes porque os testes manuais repetitivos provocam elevada fadiga mental e implicam uma grande concentração por parte da pessoa que os executa. Assim sendo, justifica-se a automação dos testes com o objectivo de reduzir resultados não verdadeiros devido a falha humana [8].

2.5.4 Como Automatizar

A automação de testes é realizada utilizando linguagens de programação, como Java, Javascript, C e C++, usando aplicações e *frameworks* específicas de *software* de testes automáticos. O processo que pode ser utilizado para automatizar o processo de teste passa por:

- Identificação das áreas dentro de um *software* para proceder à automação;
- Seleção da ferramenta adequada para a automação do teste;
- Escrita dos *scripts* do teste;
- Desenvolvimento do conjunto de testes;
- Execução dos *scripts*;
- Realização de relatórios dos resultados;
- Identificação dos problemas de *performance* ou *bugs* [8].

Neste capítulo é também essencial mencionar a pirâmide de testes e a necessidade de a ter presente quando se pretende automatizar testes de *software*.

A pirâmide de teste apresentada na Figura 2.6 é um conceito desenvolvido por Mike Cohn, descrito no seu livro "*Succeeding with Agile*". O objectivo desta pirâmide é mostrar que quando se pretende automatizar um *software* devemos ter uma quantidade muito superior de testes de unidade de baixo nível do que testes de alto nível que funcionam através de uma *User Interface*.

O uso de ferramentas de teste de alto nível tem algumas vantagens, nomeadamente a facilidade com que é possível registar e reproduzir automaticamente uma interação com a aplicação e com que ela é reproduzida, verificando se a aplicação apresentou os mesmos resultados. Essa abordagem funciona bem inicialmente, uma vez que é fácil gravar testes e os mesmos podem ser registados por pessoas sem conhecimento de programação. No entanto, este tipo de abordagem

traduz-se rapidamente em problemas. Testar através da *interface* gráfica é um método lento, aumenta os tempos de compilação e o mais importante é que esses testes são muito frágeis, dado que uma pequena alteração no sistema pode facilmente acabar por inutilizar muitos desses testes. Uma forma de contornar esse problema é abandonar essas ferramentas de gravação e reprodução e partir para a programação destes testes, tarefa mais difícil e com execução de elevado custo. Mesmo com boas práticas de programação, estes testes continuam a ser frágeis, o que prejudica a sua confiança.

Em suma, os testes que funcionam através da *User Interface* são frágeis, têm um custo elevado e demoram muito tempo para serem executados. Deste modo, este tipo de testes deve ser usado como uma segunda linha de testes de defesa com o objectivo de detetar erros ou *bugs* que os testes de unidade não foram capazes de detetar. Portanto, a pirâmide da Figura 2.6 argumenta que se deve fazer mais testes automatizados através de testes de unidade do que através de testes baseados em GUI tradicionais.

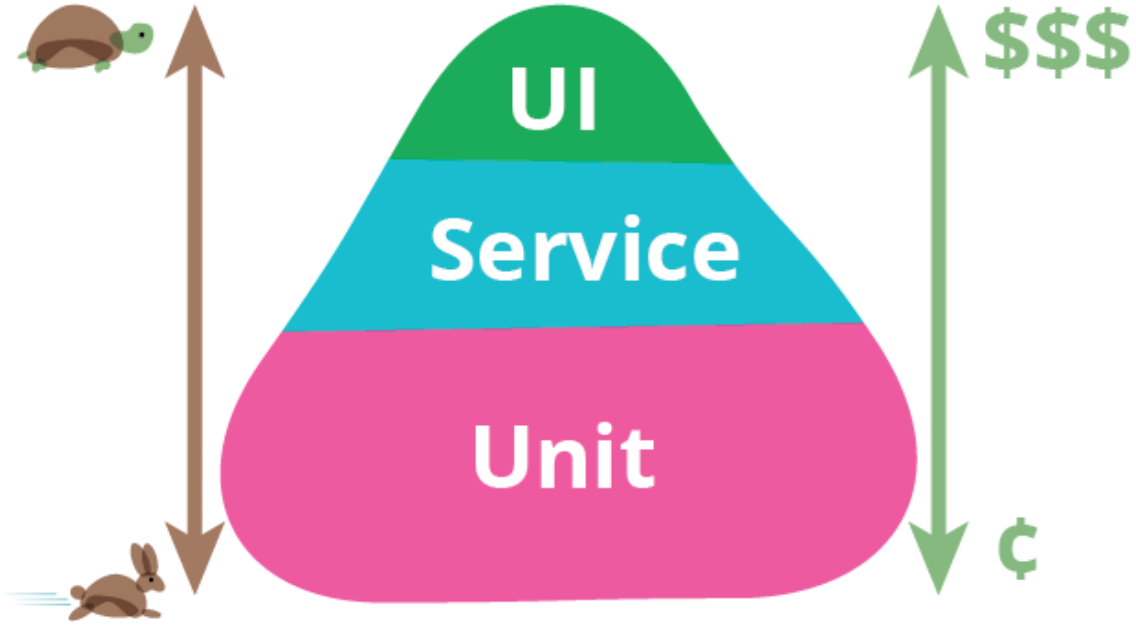

Figura 2.6: Pirâmide de testes

2.5.5 Fases do Ciclo de Vida do Teste de Software

O ciclo de vida dos testes de *software* descreve as fases do ciclo e a ordem em que as mesmas são executadas. Cada fase produz os resultados exigidos pela próxima fase no ciclo de vida e os requisitos são convertidos em *design*. O código é produzido de acordo com o *design*, o qual está integrado fase de desenvolvimento. Após a codificação e desenvolvimento do teste, é confirmado se a entrega da fase de execução está de acordo com os requisitos. A Figura 2.7 resume as características das diferentes fases do ciclo de vida dos testes de *software*.

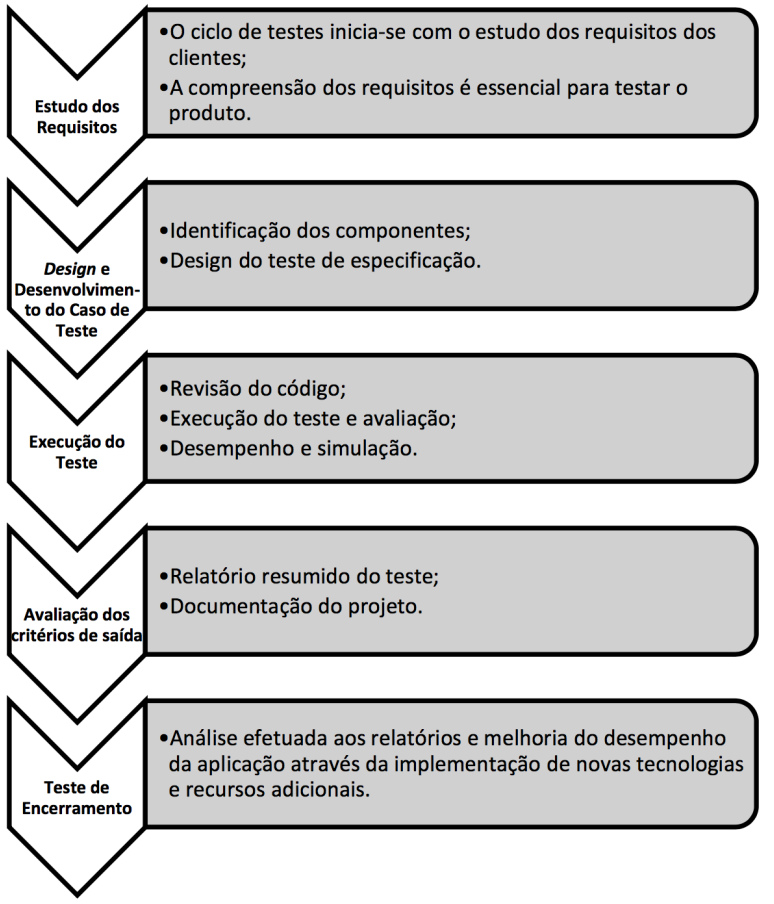

Figura 2.7: Ciclo de Vida dos Testes de *Software*

2.5.6 Ferramentas de Testes Automáticos

Esta secção visa apresentar as ferramentas não só de testes automáticos mas também outras ferramentas adicionais que foram utilizadas na elaboração desta plataforma de testes.

2.5.6.1 *Frameworks de Testes*

- Selenium Webdriver - O Selenium Webdriver é uma *framework* para testes de aplicações web que suporta diversos navegadores e linguagens de programação. O Selenium é a integração do WebDriver, outra *framework* para testes, que usa extensões ou recursos nativos oferecidos pelo próprio *browser* para controlá-lo diretamente. O Selenium é uma *framework* poderosa que permite escrever testes em diversas linguagens de programação (de Java a Erlang, passando por C e PHP), oferecendo um rápido simulador de *browsers*, que é leve e dispensa uma *interface* gráfica. A integração entre o Selenium e o Webdriver permite criar testes mais avançados permitindo simular entradas do utilizador usando chamadas a nível

do sistema operacional. O WebDriver é suportado no Firefox (FirefoxDriver), IE (InternetExplorerDriver), Opera (OperaDriver) e Chrome (ChromeDriver). O suporte ao Safari não foi incluído neste lançamento devido a restrições técnicas, mas pode ser simulado usando o SeleneseCommandExecutor. Também funciona no Android (AndroidDriver) e iPhone (IPhoneDriver), para teste de aplicações web para dispositivos móveis. Existe ainda uma implementação baseada no HtmlUnit que não depende de navegadores em si, chamada HtmlUnitDriver. As APIs do WebDriver podem ser acessíveis a partir de código Python, Ruby, Java e C, permitindo que os desenvolvedores criem testes usando sua linguagem de programação preferida.

- TestNG - TestNG é uma *framework* de testes inspirada no Junit and NUnit. Tem funcionalidades como: suporte de anotações, verificação da segurança do código num ambiente *multithread*, configurações flexíveis, suporte a parâmetros, dependência de métodos e grupos de métodos, onde é possível identificar testes por grupo e facilmente executar todos os testes de um grupo específico ou excluir testes de um determinado grupo.

2.5.6.2 Ferramentas Adicionais

- Apache Ant - Apache Ant é uma biblioteca de Java. É uma ferramenta de linha de comando cuja missão é gerar processos descritos em arquivos de compilação como alvos e pontos de extensão dependentes uns dos outros. O principal uso conhecido de Ant é a compilação de aplicações Java. A Ant fornece uma série de tarefas internas que permitem compilar, montar, testar e executar projetos Java.

- Java Development Kit (JDK) - O Java Development Kit (JDK) é um ambiente de desenvolvimento de *software* usado para desenvolver aplicações Java. Inclui Java Runtime Environment (JRE), um compilador (javac), um arquivador (jar), um gerador de documentação (javadoc) e outras ferramentas necessárias no desenvolvimento de Java.

- GIT - Git é um sistema de controlo de versão distribuído e um sistema de gestão de código fonte. O Git foi projetado inicialmente e desenvolvido por Linus Torvalds para o desenvolvimento do kernel Linux, mas foi adotado por muitos outros projetos. Cada diretório de trabalho do Git é um repositório com um histórico completo e com a capacidade de acompanhamento das revisões, não dependente de acesso a uma rede ou a um servidor central.

- O Jenkins é uma ferramenta que permite a integração de várias ferramentas como o Selenium WebDriver, TestNG, Ant, JDK e GIT que tornam possível a execução e gestão deste projeto de automação de testes. O Jenkins deve ser instalado num servidor onde, a partir de uma *interface* gráfica providenciada por esta ferramenta, é possível gerir e compilar o projeto, seguindo o seguinte fluxo: os programadores enviam o seu código para um repositório que é lido pelo Jenkins; o Jenkins automaticamente compila o código do projeto e executa os testes automáticos presentes no projeto; o resultado da compilação é mostrada na *interface* gráfica do Jenkins, onde é possível analisar relatórios de teste e o estado das compilações.

Os resultados e as notificações dos relatórios da compilação e dos testes executados podem ser enviados também para os programadores no final de cada compilação.

2.6 Resumo ou Conclusões

O presente capítulo faz uma contextualização pormenorizada sobre testes, desde a sua definição como conceito e tipos de testes existentes aos métodos e níveis utilizados na sua aplicação. É feita também uma análise às ferramentas mais usadas na área da automação de testes. Assim sendo, o presente capítulo constitui a base para idealizar uma proposta de solução ao desafio pré-estabelecido pela IPBRICK SA, uma vez que fornece um conhecimento geral e pormenorizado de conceitos fundamentais para a concretização do problema.

Capítulo 3

Descrição do Sistema IPBrick

Uma vez que o objetivo do projeto consiste em implementar uma plataforma para testar automaticamente um dos principais produtos desenvolvidos pela IPBrick SA, um sistema operativo denominado de IPBrick OS, este apresenta-se como um componente importante deste trabalho. Assim sendo, este capítulo pretende fazer uma descrição de como este sistema é constituído.

3.1 IPBrick OS

A IPBRICK OS é uma plataforma de comunicações para sistemas de servidores corporativos com base em Linux Debian e é usado para gerir:

- Comunicações Unificadas;
- Gestão Documental e de Processos;
- E-mail e Ferramentas Colaborativas;
- Rede Social Corporativa [9].

O *software* IPBrick OS está projetado para que três tipos de servidores sejam geridos pelo mesmo administrador. Estes servidores são o Servidor de Intranet, o Servidor de Comunicações Unificadas e o Servidor de Segurança.

A IPBrick OS tem uma *interface web* que é baseada em *HyperText Markup Language* (HTML), AJAX e *Hypertext Preprocessor* (PHP) e pode ser acedida a partir de qualquer *browser*, sendo apenas necessário colocar o endereço *Internet Protocol* (IP) ou *Fully Qualified Domain Name* (FQDN) da máquina como *Uniform Resource Locator* (URL) e introduzir os respectivos dados (utilizador e palavra-passe) de autenticação. Fundamentalmente, é constituída por um conjunto de menus, formulários e campos de seleção que permitem ao utilizador configurar todo o sistema operativo, como por exemplo: adicionar máquinas e utilizadores, configurar as propriedades das *interfaces* de rede (endereço IP, máscara, endereço de rede e *broadcast*), configurar o servidor de *mail*, *Dynamic Host Configuration Protocol* (DHCP) ou *Domain Name System* (DNS), configurar a *firewall*, configurar o serviço VPN (*Virtual Private Network*) e muito mais. A *interface* possui

Figura 3.1: *interface Web* da IPBrick OS

uma opção que permite ao utilizador aplicar as alterações efetuadas, sendo que neste caso é feita uma sincronização com a base de dados e todas as modificações efetuadas são lá armazenadas [10].

3.1.1 IPBrick.I - Servidor de Intranet

Uma intranet consiste na aplicação das tecnologias da internet às redes internas das empresas. A IPBrick.I disponibiliza aos seus utilizadores funcionalidades tais como:

- **Correio Eletrónico e Ferramentas Colaborativas**, um servidor de Correio Eletrónico e disponibiliza Ferramentas Colaborativas (Agenda, Contactos e Calendário).
- **Controlador de Domínio**, trabalha com protocolo LDAP (*Lightweight Directory Access Protocol*) e integra com o *Active Directory*, herdando todas as configurações de contas já criadas. A passagem da tecnologia Microsoft para a tecnologia IPBrick é transparente para o utilizador.
- **Funcionalidades de Intranet**, providencia várias funcionalidades, tais como, servidor de ficheiros e de áreas de trabalho individuais e de grupo, servidor de terminais, impressoras e bases de dados, serviço de *backup* das áreas de trabalho e suporte de aplicações de negócios [11].

3.1.2 IPBrick.C - Servidor de Comunicações

Por sua vez, o servidor de comunicações vai garantir funcionalidades como:

- **Correio eletrónico**: *Mail Relay* e *webmail*;
- **Comunicações**: Fax para e-mail, e-mail para fax e e-mail para SMS;
- **Serviços de Telefonia**: Gateway VoIP, PBX IP e SIP Proxy;
- **Serviços web**: Servidor web, Webphone, proxy e cache HTTP/FTP;

- **Sistema de Mensagens Instantâneas**: Webchat e servidor de Instant Messaging;
- **Segurança de Comunicações**: Firewall, VPN Server, IDS (Intrusion Detection System), antivírus pré-instalado para correio eletrónico e proxy e também anti-spam pré-instalado.

3.1.3 IPortalDoc

O iPortalDoc é um sistema de gestão documental e *workflow* que permite fazer uma gestão do fluxo de documentos ou proceder ao seu arquivo para futura gestão.

O iPortalDoc não funciona de forma independente, funciona em colaboração com o sistema operativo IPBrick. Além deste sistema operativo, o iPortalDoc precisa de uma base de dados onde possa armazenar os documentos. Também está integrado com um servidor web, um servidor de correio eletrónico, um servidor de ficheiros e ainda um servidor de informação e gestão de domínios. Os utilizadores que podem ter acesso a esta aplicação são criados através da aplicação de gestão de contactos da IPBrick [12].

3.1.4 Rede Social Corporativa CAFE

Fazendo parte da rotina de milhares de milhões de pessoas no mundo, as redes sociais vieram para ficar. Estas proporcionam uma troca fácil de notícias, ideias e opiniões entre os utilizadores. Vendo vantagens neste tipo de ferramenta ao nível empresarial, a IPBrick resolveu criar uma rede social empresarial, o CAFE.

Figura 3.2: Rede Social Corporativa *CAFE*

A capacidade de comunicar pelos diferentes departamentos da empresa, trocar documentos em tempo real, diminuindo a necessidade de enviar um e-mail, e de melhorar as relações entre colaboradores, constitui apenas algumas das vantagens da utilização de uma ferramenta como esta em ambiente empresarial. Esta rede social proporciona assim um espaço virtual comum, para colaboradores da mesma empresa trocarem informações úteis sobre o trabalho que se encontram a desenvolver, melhorem as suas relações pessoais e, sobretudo, aumentarem a produtividade.

Para além das funcionalidades que tipicamente constituem uma rede social, a IPBrick quis ir mais longe, integrando as suas próprias ferramentas de comunicação nesta. Quer isto dizer que o CAFE inclui o sistema UCoIP, assim como o acesso direto às aplicações empresariais integradas no pacote IPBrick. Deste modo, são apresentadas a seguir as principais ferramentas desta rede social:

- Feed de Notícias e Publicações – capacidade de publicar textos, páginas web, imagens ou vídeo, comentar e editar *posts*;
- Informação recente da empresa – permite melhorar a comunicação ao transmitir toda a informação da empresa, tornando-a acessível a todos;
- RSS Feed – Informações atualizadas, permitindo que os utilizadores sejam informados das últimas notícias sem a necessidade de iniciarem sessão;
- Barra lateral de lista de utilizadores – Acesso a todos os utilizadores pertencentes à rede social. Tem incorporada a funcionalidade de pesquisar os utilizadores por nome e, para cada um deles, acesso direto a 4 tipos de comunicação: e-mail, chat, voz e vídeo WebRTC;
- Acesso direto para fazer chamada – pressionando o botão correspondente ao telefone do utilizador que se pretende contactar, a chamada é feita e reencaminhada para o telefone da secretária do utilizador. Deste modo não há a necessidade de memorizar ou introduzir a extensão do telefone da pessoa a quem se pretende contactar;
- Videochamada via web browser – permite fazer e receber videochamadas para colegas de trabalho, além de receber videochamadas de qualquer pessoa fora da empresa que tenha solicitado o utilizador através da página UCoIP;
- Atividades da empresa – acesso direto a partir de uma única página a atividades específicas como questionários e votações;
- Aplicações Empresariais – Capacidade de autenticação rápida e automática para todas as aplicações empresariais IPBrick, bastando um clique para aceder a qualquer um dos serviços desejados [9].

Capítulo 4

Problema e Requisitos

Feita a revisão dos conceitos teóricos, tecnologias e ferramentas usadas para a automação de testes no Capítulo 2 e uma descrição do sistema IPBrick, que irá ser alvo dos testes automáticos no Capítulo 3, neste capítulo o objetivo é explicar o problema proposto e especificar os requisitos funcionais e não-funcionais para o projeto, bem como apresentar os casos de teste que foram automatizados.

4.1 Levantamento dos Requisitos

Sendo a IPBrick SA uma empresa que prima pela qualidade dos seus serviços, existe a necessidade de despender um número elevado de horas a realizar testes manuais para garantir essa qualidade. Estes testes, para além de consumirem uma boa parte dos recursos humanos da empresa, também acabam por se tornar monótonos e repetitivos, levando a um desaproveitamento e mesmo a uma desmotivação por parte das pessoas que executam estes testes. A implementação de uma plataforma de testes automáticos visa colmatar este problema e, de forma a iniciar este projeto, nesta secção irão ser apresentados os requisitos que eram pretendidos para esta plataforma. Estes foram divididos em requisitos funcionais e requisitos não-funcionais.

4.1.1 Requisitos Funcionais

O principal requisito para este projeto era criar uma plataforma com uma *interface* gráfica onde fosse possível executar automaticamente um conjunto de testes definidos pela IPBrick SA, que até ao momento eram realizados manualmente. Este conjunto de testes, denominados de Testes Obrigatórios, têm como objetivo garantir, do ponto de vista do utilizador, que as funcionalidades principais da IPBrick OS mantenham a sua integridade sempre que é lançado um novo *update* ou uma nova versão. Esta plataforma deverá ser capaz também de gerar relatórios relativos aos testes automáticos executados.

É também essencial fazer uma escolha pertinente das entradas de dados a ser utilizados na execução dos testes automáticos, como por exemplo a utilização e inserção de caracteres especiais ($ % ! #) nos campos de nome e palavra-passe quando se pretende testar um simples formulário de

inscrição num *website*. Nesse sentido, um requisito importante para este projeto era a possibilidade de escolha e configuração das entradas de dados utilizados para executar os testes.

4.1.2 Requisitos Não-Funcionais

Os Testes Obrigatórios representam apenas uma pequena parte da totalidade dos testes que são feitos na IPbrick SA. Sendo objetivo futuro da empresa continuar com a automação dos testes que são realizados nos seus produtos, era pedido que esta plataforma permitisse a integração de novos testes. Para isso, o sistema e o *design* da solução deveriam ser modulares, tanto quanto possível.

A IPBrick SA é uma empresa que apenas usa ferramentas gratuitas. Como tal, outro requisito, era que durante o desenvolvimento deste projeto apenas fossem utilizadas ferramentas gratuitas.

4.2 Casos de Teste

Esta secção visa apresentar os casos de teste que a IPBrick SA realiza manualmente, sempre que é lançada uma nova versão do seu produto. Como já foi referido no capítulo anterior o objectivo deste projeto é realizar a automação deste conjunto de casos de teste, que têm o nome de "Testes Obrigatórios".

4.2.1 *Network Tests*

Este caso de teste visa testar a definições *Network* da IPBrick OS estes testes consistem em:

- *Change Name/Domain/IP/Gateway one by one (apply configurations for each modification).*
- *Change Name/Domain/IP/Gateway at the same time.*

4.2.2 *Users and Machines Tests*

Este caso de teste consiste em fazer testes ás definições dos utilizadores e máquinas os casos de teste desta categoria são:

- *Add users one by one (test also with real user names, with accentuation, etc)*
- *Mass operations (500 users) (test also with real user names, with accentuation,etc)*
- *Check system services in order to know if the users were correctly created (user folder exists, user folder permissions, id, login, ldap, email, im, voip)*
- *Add Machines one by one (telephones, workstations, etc)*
- *Mass operations (500 machines, different types) with and without mac address*
- *Add User and Machine Groups. Associate users and machines to groups. Check system services : id info for users, LDAP*

- *Check system services in order to know if the machines were correctly created (ldap, dns, dhcp, voip (for type phones only))*

4.2.3 *Configuration Replacements*

Este caso de testes tem como objectivo testar a funcionalidade de repor configurações.

- *Replace configurations (Default Configuration)*
- *Replace configurations (Another configuration other than the Default)*

4.2.4 *Updates Tests*

Estes casos visam testar que um *update* foi bem realizado e que as funcionalidades do produto se mantêm de acordo com o esperado mesmo depois do das alterações causadas pelo *update*. Os casos de teste para esta categoria são:

- *Check conflicts and dependencies Rebrands can be used for this test.*
- *Test installation over all RC versions of the update. Install different RC versions in multiple servers.*
- *Test installation over all the replacements handled by this update (fix updates, specific updates) Rebrands can be used for this test.*
- *After the upload of the Update was finished, press immediately "Apply Configurations". Monitor the evolution of the installation through the update log that is available on the folder /opt/system/log/ . On web interface check it is impossible "Apply Configurations"till the log finish with the expression "ALL DONE".*
- *If testing an Security Update, check the installation with a expired licence. The installation should be possible.*
- *Stop all services before install the Update. Install the Update, "Apply Configurations". Reboot the machine and check if any service was started.*
- *Compare the files on the folder "/etc"before and after the Update installation with "Apply Configurations"*
- *If the update install other applications (see sys-control.php), check on the PREINST the minimum disk space required. Try to install the update with low disk space and check the installation is aborted.*
- *Install the update on one IPBrick LVM (current version is 6.2-lmv-RC4). Check the installation log.*

- *Install other applications (iportaldoc, ucoip recording, hotspot, (other system-apps), etc..) and check if they are working correctly*
- *Install other applications (iportaldoc, ucoip recording, hotspot, (other system-apps), etc..) before install the Update, and then try to install the Update.*

4.2.5 CAFE *Integration Tests*

Este caso de teste visa testar a integração de um dos produtos da IPBrick, que é a a Rede Social corporativa CAFE, com a IPBrick OS.

- *On the CAFE, modify the Personal Data of one User: Password, Auto Reply Message, Job and Follow Me. Check on IPBrick if the changes were replicated and are working as defined.*
- *On the IPBrick, modify the definitions of one User: Password, Auto Reply Message, Job and Follow Me. Check on CAFE if the changes were replicated and are working as defined.*
- *Enable the Policies Passwords on IPBrick, and "Apply Configurations". On the CAFE modify the Personal Data of one User: Password, Auto Reply Message, Job and Follow Me. Check on IPBrick if the changes were replicated and are working as defined.*

Capítulo 5

Implementação da Solução

Após ter sido feita uma revisão bibliográfica no capítulo 2 e o levantamento dos requisitos no capítulo 4, este capítulo tem como propósito demonstrar a solução concebida. É apresentada a Plataforma de Testes, sendo expostos todos os seus componentes. Estes são explicados e descritos tanto ao nível da sua arquitetura lógica como física, sendo ainda abordada a estrutura funcional.

A plataforma de testes é composta apenas por um módulo principal. Este módulo constitui uma *interface* gráfica onde é possível fazer uma configuração prévia dos testes, bem como a seleção de quais os testes que se pretende executar. Este módulo também contém os *scripts* e toda a lógica responsável pela execução automática dos testes.

Este capítulo começa por fazer uma abordagem a este módulo descrevendo a sua estrutura, funcionamento, funcionalidades e como foram integradas as várias ferramentas que foram utilizadas e mencionadas no capítulo anterior, de forma a ser possível entender como foram aplicadas e executadas as metodologias e técnicas de teste anunciadas nesse mesmo capítulo.

5.1 Arquitetura da Plataforma de Testes

A plataforma de testes automáticos baseia-se na integração de várias ferramentas. Estas ferramentas tornam possível a integração da parte lógica e a *interface* gráfica dentro do mesmo módulo. Através da *interface* gráfica, é possível configurar e despoletar o processo de execução dos testes automáticos, que é assumido pela parte lógica.

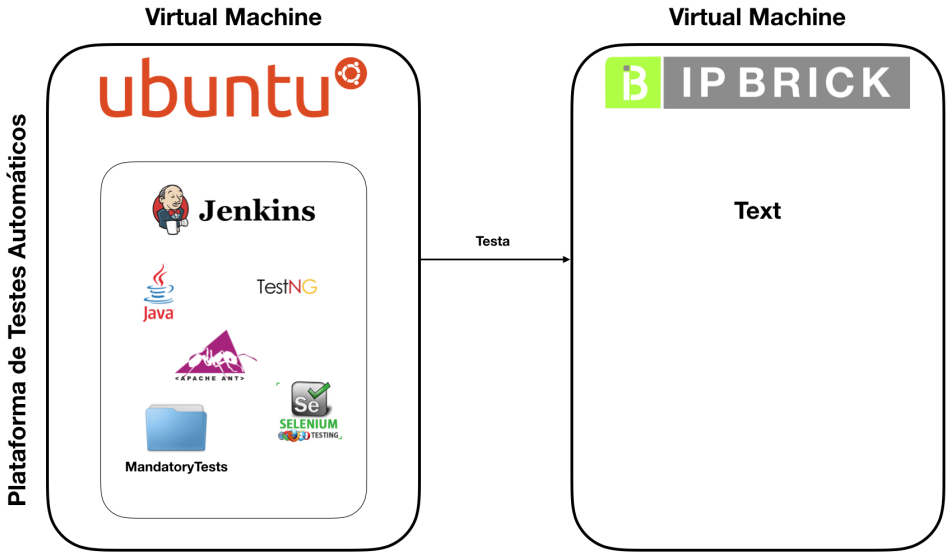

Figura 5.1: Arquitetura geral da plataforma de testes automáticos

Esta plataforma está montada numa máquina virtual que tem o Ubunto como sistema operativo. Neste ambiente foi configurado o Jenkins e é através desta ferramenta que a plataforma de testes ganha forma.

Tal como foi mencionado no Capítulo 2, o Jenkins é uma ferramenta de integração contínua, possui uma *interface* gráfica e tem a capacidade de integrar ferramentas como Apache Ant, Java JDK, Selenium Webdriver, TestNG e GIT, todas elas necessárias para a compilação e execução dos testes automáticos. A utilização do Jenkins neste projeto visou principalmente tirar partido da sua capacidade de integração das ferramentas necessárias para a parte lógica do projeto, bem como do facto desta ferramenta possuir uma *interface* gráfica.

Figura 5.2: integração do JDK no Jenkins

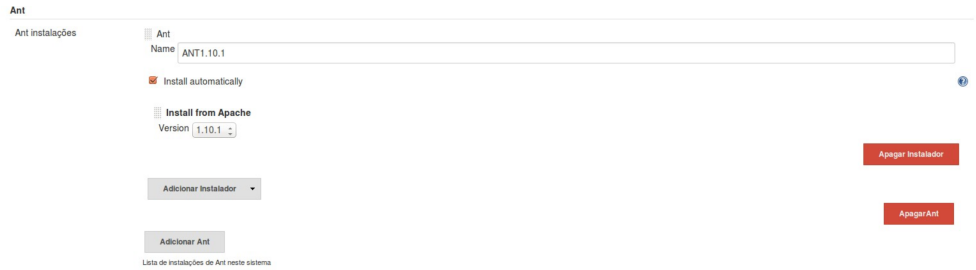

Figura 5.3: Integração do Apache ANT no Jenkins

Como é visível na Figura 5.1 , a *interface* gráfica do Jenkins está acessível a partir de um *browser*. É nesta *interface* que o utilizador tem a opção de fazer a seleção dos testes que pretende executar, bem como fazer as respetivas configurações das entradas dos dados de teste que se pretende utilizar. Após a finalização dos testes ainda é possível fazer uma análise dos relatórios e dos *logs* gerados automaticamente pelo Jenkins.

5.2 Funcionamento e Funcionalidades

Para entender com mais detalhe a forma como esta plataforma funciona, olhemos para o fluxograma da Figura 5.4, no qual é possível observar o fluxo dos processos que são executados. Começa-se por selecionar os testes que se pretender executar até ao final, onde podemos fazer uma análise do relatório gerado por esse teste.

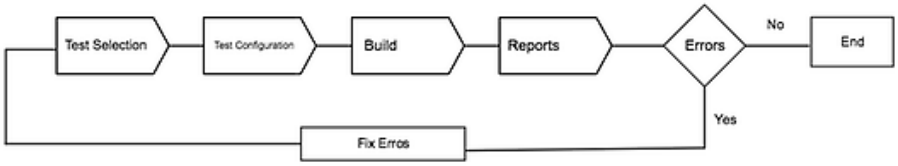

Figura 5.4: Fluxo dos processos na plataforma de testes

5.2.1 Seleção e Configuração dos Testes

A seleção e configuração dos testes é feita a partir da *interface* gráfica do Jenkins. Cada *Suite* ou conjunto de testes possui dois ficheiros xml:

- Ficheiro xml que é todos os casos de testes de cada *Suite* de Testes;
- Ficheiro de configuração das entradas de dados relativo a cada *Suite* de Testes.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="NetworkSuite">
  <test name="NetworkTest" group-by-instances="true" preserve-order="true">
    <classes>
      <class name="Tests.NetWorkTest"/>
      <groups>
                    <run>

                    <include name="ChangeIP"/>
                    <include name="VerifyIP"/>

                    <include name="ChangeDomain"/>
                    <include name="VerifyDomain"/>

                    <include name="ChangeGateway"/>
                    <include name="VerifyGateway"/>

                    <include name="ALL"/>
                    <include name="VerifyALL"/>

                    </run>
                </groups>
    </classes>
  </test>
</suite>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<TestData>
    <url-name>
        <url>https://172.31.3.49</url>
        <ip>172.31.3.48</ip>
    </url-name>

    <user-details>
        <username>admin</username>
        <password>123456</password>
        <implicitlyWait>30</implicitlyWait>
    </user-details>
</TestData>
```

Figura 5.5: Exemplo de Ficheiros de *Suite* e Configuração de Testes

A estrutura de ficheiros do projeto será explicada à frente com mais detalhe mas, por enquanto, a principal ideia a reter é que a seleção dos testes a executar é realizada a partir da edição do ficheiro xml responsável por identificar todos os casos de teste relativos a cada *Suite*.

A configuração das entradas de dados relativos a cada *Suite* é de igual forma feita a partir da *interface* gráfica, editando o ficheiro xml de configuração respetivo a cada *Suite* de testes. É neste ficheiro que são introduzidos dados como o URL ou IP do sistema alvo dos testes bem como informações adicionais necessárias para a realização do teste.

5.2.2 Compilação do programa de Testes

Depois da seleção e edição dos ficheiros xml, o passo seguinte no fluxograma da figura 5.6 é a compilação ou *Build* do projecto. Este processo é despoletado através da *interface* gráfica com a ordem de início dos testes, sendo uma responsabilidade da parte lógica que o compilador, neste caso o Apache Ant, lê um ficheiro Build.xml e inicia o processo de compilação. Este ficheiro Build.xml é um *script* do Apache Ant criado pelo *Plug-in Development Environment* (PDE) que recebe os ficheiros *xml* de configuração e de *Suite* de testes, que são ficheiros que fazem parte da framework TestNG que tal como foi mencionada no capitulo 2 permite fazer configurações flexíveis e suporte a parâmetros, e combina todos os *plugins*, componentes, bibliotecas (Selenium-standalone, TestNg, *plugins* de relatórios de teste) e transforma tudo isso num único ficheiro JAR, pronto a ser compilado.

Neste momento, os testes automáticos são iniciados enviando pedidos e recebendo respostas http pelo *browser* através do Selenium para o sistema em teste.

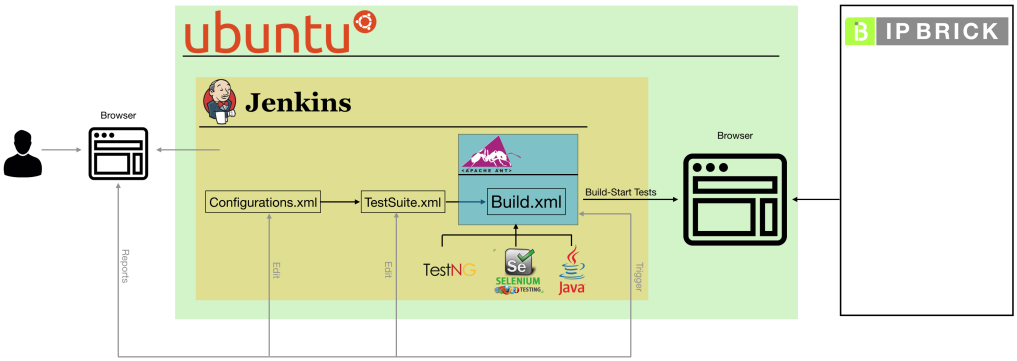

Figura 5.6: Processo de compilação da Plataforma de Testes

5.2.3 Geração de Relatórios e *Logs* de Teste

Depois da *build* estar completa e os testes terem terminado, o Jenkins gera automaticamente um relatório através de um *plugin* de relatórios, onde é possível ser analisado na *interface* gráfica com a informação relativa aos testes corridos.

O Jenkins também contém uma consola integrada onde é possível, também através da interface gráfica, fazer uma análise das mensagens e *logs* gerados durante a compilação dos testes. Todos os relatórios, *builds* e *logs* ficam guardados na base de dados criada automaticamente na instalação do Jenkins, onde estes são organizados por data, hora e utilizador que os realizou, e onde é possível consultá-los através da *interface* gráfica. O processo de testes termina então com a análise do resultado do relatório.

5.3 Técnicas de design e Padrões

Feita a análise da arquitetura e funcionamento da plataforma nesta secção, passemos a uma análise mais detalhada de como o programa de testes automáticos foi construído, quais as técnicas, padrões e métodos que foram utilizados.

5.3.1 Técnicas de Teste Utilizadas

Considerando a figura 2.6 do Capítulo 2 referente à pirâmide de testes de *software*, não restam dúvidas que a melhor abordagem para desenvolver testes automáticos para testar a IPBrick SO seria utilizar maioritariamente testes de unidade e, como uma segunda linha de testes, utilizar testes de UI. Contudo, a IPBrick SO não possui testes de unidade. Como foi mencionado no Capítulo 2, a realização destes testes é responsabilidade dos programadores, uma vez que são eles que têm o conhecimento da forma como desenvolveram o seu código. A realização deste tipo de testes seria um processo muito caro pois iria consumir muito tempo aos programadores da IPBrick SA e, deste modo, o tipo de testes usados neste projeto incidiu maioritariamente na utilização de teste de UI.

Com o objetivo de cumprir o principal requisito para os testes, que passa por garantir, do ponto de vista do utilizador, que as funcionalidades principais do *software* mantenham a sua integridade sempre que fosse lançado um novo *update* ou uma nova versão, o foco dos testes automáticos é então a *interface* gráfica da IPBrick SO, simulando assim a integridade das funcionalidades do ponto de vista do utilizador. Para cumprir este requisito, a opção mais pertinente para a automatização dos testes foi a utilização de testes de Regressão. Os testes de Regressão são do tipo Funcionais, que usam o método de Caixa-preta. Foram realizadas várias experiências onde os testes foram corridos no sistema IPBrick. Os resultados obtidos foram sendo validados, de acordo com os resultados obtidos pelos programadores quando executavam o mesmo caso de teste de uma forma manual.

5.3.2 Padrão de Desenho e Linguagem

Para a automação dos testes foi necessário proceder a uma escolha de *frameworks* de testes. Os *frameworks* escolhidos para a automação dos testes foram o Selenium e TestNG. Esta escolha é justificada pelo facto de estes *frameworks* serem gratuitos, e também por serem os mais versáteis no que toca tanto ao suporte de linguagens de programação como à sua aplicação.

Apesar da *interface* da IPBrick SO estar construída maioritariamente em PHP, a linguagem de programação escolhida para a utilização do Selenium e programação dos testes automáticos foi o JAVA. Como o método utilizado na automação foi o método de Caixa-preta, a linguagem de programação utilizada na automação dos testes não é relevante visto que este é um método não invasivo, sendo que a escolha de JAVA para este projeto incidiu no facto de ser a linguagem que tem mais suporte da comunidade, pois é a linguagem mais aplicada na automação de testes.

Para além da escolha da linguagem, também houve a necessidade de escolher um padrão de *design* para a estrutura dos testes. O padrão utilizado no desenvolvimento dos testes foi o *Page object model* (POM).

O *Page object* é um padrão de concessão que cria um repositório de objetos para os elementos contidos na *interface web* da aplicação. Para cada página da aplicação, deve haver uma classe correspondente. Na classe correspondente a cada página são classificados os *WebElements* que são utilizados para a identificação dos botões, *links* ou outros elementos necessários para a realização do teste. Nessa classe são também definidos os métodos relativos a cada página que executam operações com esses *WebElements*. Os *WebElements* são elementos html que representam os elementos físicos que constituem uma pagina web. O Selenium usa estes *WebElements* para executar as ações que são pretendidas na execução do teste. Estes *WebElements* podem ser identificados pelo seu id, xpath, class, css ou nome. O POM permite manter o código organizado e fácil de compreender, mas a principal vantagem é que este possibilita uma fácil manutenção dos testes caso a *interface* da aplicação sofra alterações. Para isso basta identificar em que página é que a *interface* sofreu alterações e dessa forma alterar os métodos ou os elementos que foram alvo de modificações.

5.3.3 Aplicação do Padrão *Page Object*

Para além de dos *packages* essenciais de um projeto JAVA, a estrutura principal do programa de testes aplicando o padrão de *Page Object*, vai ter uma divisão em três principais *packages*:

- *Pages*;
- *Tests*;
- *Data*.

Na Figura 5.7 podemos observar o aspecto da estrutura final do projeto aplicando o padrão *Page Object* e onde está programada toda a automação dos casos de teste referidos no capítulo anterior. Para uma melhor compreensão da utilização deste padrão, nas subsecções seguintes passemos a uma explicação mais detalhada dos seus *packages* e de que forma contribuem para a estrutura do programa.

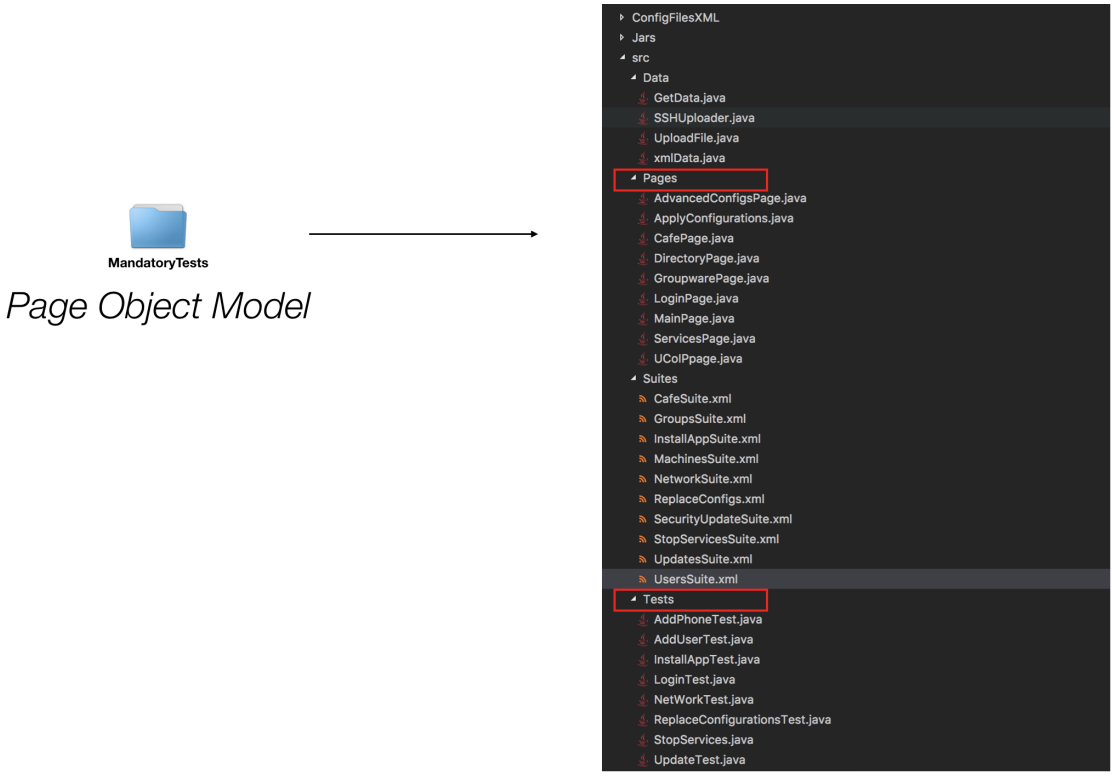

Figura 5.7: Estrutura do programa utilizando o padrão *Page Object*

5.3.3.1 *Package Pages*

Este *package* contém todas as classes de objetos relativas às Páginas da *interface* da IPBrick SO, dentro de cada classe encontram-se definidos os *WebElements* e os métodos, como é referido na secção anterior. Na Figura 5.8 podemos ver o exemplo de uma classe "DirectoryPage". Esta

classe é onde estão contidos todos os *WebElements* e métodos relativos ao menu Directory que vão ser usados posteriormente pelos testes que estão contidos no *package Tests*. O menu Directory é um dos menus da interface gráfica da IPBrick OS onde é possível adicionar, editar e remover utilizadores e máquinas.

```
14  public class DirectoryPage {
15
16      WebDriver driver;
17      MainPage objMainPage;
18      AdvancedConfigsPage objAdvConfig;
19      UploadFile objUploadFile;
20      xmlData objxml= new xmlData();
21      String ConfigXmlFile="FilesXML/AddUsers.xml";
22
23      @FindBy(id="menuheader0") WebElement DirectoryMenu;
24      @FindBy(linkText="Users Management")WebElement UsersManagement;
25      @FindBy(linkText="Machines Management")WebElement MachineManagement;
26      @FindBy(linkText="Users list")WebElement UserList;
27      @FindBy(linkText="Insert") WebElement Insert;
28      @FindBy(linkText="Mass operations") WebElement Moperation;
29      @FindBy(name="nome") WebElement nome;
30      @FindBy(name="username")WebElement username;
31      @FindBy(name="login") WebElement login;
32      @FindBy(name="server")WebElement server;
33
34      public DirectoryPage(WebDriver driver) {
35          this.driver = driver;
36          PageFactory.initElements(driver, this);
37      }
38      public void goToMachineManagement(){
39          objMainPage= new MainPage(driver);
40          objMainPage.switchToFrame("Default");
41          DirectoryMenu.click();
42          MachineManagement.click();
43          objMainPage.switchToFrame("maquinas_ver_lista");
44          Moperation.click();
45      }
46      public void goToUserListPage(){
47          objMainPage= new MainPage(driver);
48          objMainPage.switchToFrame("Default");
49          DirectoryMenu.click();
50          UsersManagement.click();
51          UserList.click();
52      }
53      public boolean VerifyUser(){
54          objMainPage= new MainPage(driver);
55          objMainPage.switchToFrame("Default");
56          DirectoryMenu.click();
57          UsersManagement.click();
58          UserList.click();
59          objMainPage.switchToFrame("utilizador_ver_lista");
60          return Test.isDisplayed();
61      }
```

Figura 5.8: Aspecto de uma classe referente a uma página *"Advanced Configurations"*

Na Figura 5.8 podemos observar os os *WebElements* definidos através das instruções do Selenium

@FindBy(id="xxx")WebElement XXX;

onde definimos o método como estes *WebElements* são identificados. Estes *WebElements* podem ser identificados através do seu *ID, nome, xpath, ou class*.

É possível ver também os métodos que foram definidos neste página. Estes métodos são responsáveis por executar tarefas como a navegação até á zona pretendida na página, inserção de valores nos campos de dados, seleção de elementos em menus do tipo *drop down*, ou verificar se um certo elemento é visível.

Pegando no método *VerifyUser()* como exemplo, podemos ver que a sua função é retornar se um certo utilizador é visível ou não na lista de utilizadores da página responsável por listar todos os utilizadores dentro do menu Directory. Para isso, este método realiza *clicks* que são também instruções do Selenium, nos *WebElements* identificados previamente, navegando até á página "Users List"onde constam listados os utilizadores e por fim este método retorna se o utilizador se encontra visível ou não dentro dessa lista.

5.3.3.2 *Package Tests*

Este *package* contém as classes onde estão definidos os testes. Cada classe contém mais do que um teste que usam os métodos definidos dentro do *package Pages* para executar as ações pretendidas para o teste. Por fim cada teste deve conter um ou mais *Assertions*, que vão fazer a comparação entre o valor devolvido pelos métodos e o valor esperado, sendo assim possível fazer uma validação do Teste que foi executado. O conjunto formando pelos testes dentro da classe formam assim uma *Suite* de testes. Por exemplo, para o ficheiro de teste “AddUserTest.java” existem quatro testes diferentes:

- Adicionar Utilizadores;
- Adicionar Utilizadores em massa;
- Modificar Utilizadores;
- Remover Utilizadores.

```
27  public class AddUserTest {
28      WebDriver driver;
29      LoginPage objLogin;
30      DirectoryPage objDirectory;
31      MainPage objMainPage;
32      AdvancedConfigsPage objAdvDefPage;
33      ApplyConfigurations objApplyConfigs;
34      UploadFile objUploadFile;
35      GroupwarePage objGroupware;
36      CafePage objCafe;
37      SSHUploader objSSH;
38      xmlData objxml= new xmlData();
39      String baseUrl,baseUrl2,ip,baseUrlcafe;
40      String ConfigXmlFile="FilesXML/AddUsers.xml";
41
42      @BeforeTest (alwaysRun=true)
43      public void setup() throws ParserConfigurationException, SAXException, IOException{…
54      }
55      @Test(groups={"AddIndividualUser"},priority=1)
56      public void AddIndividualUser() throws Exception{…
72      }
73      @Test(groups={"ModifyUser"},priority=1)
74      public void ModifyUser() throws Exception{…
107     }
108     @Test(groups={"DeleteUser"},priority=1)
109     public void DeleteUser() throws Exception{…
142     }
143     @Test(groups={"AddMassUsers"},priority=1)
144     public void AddMassUser() throws Exception{…
163     }
164     @AfterMethod(groups = {"AddIndividualUser","AddMassUsers"})
165     public void shutDownSelenium() {
166         driver.manage().deleteAllCookies();
167         driver.quit();
168     }
169 }
170
```

Figura 5.9: Aspecto de uma *Suite* de testes

O conjunto destes quatro testes formam então a *Suite* de testes *"AddUserTest"*. Como podemos ver na Figura 5.9, um teste é composto por 3 fases distintas:

- @BeforeTest - Onde é feita uma configuração inicial para os testes que vão ser executados. Estas configurações são como por exemplo a definição do Browser que pretendemos usar para a realização dos testes, definição de timeouts bem como o URL da pagina que queremos testar.

- @Test - Nesta fase o teste é executado, pegando no exemplo do teste "AddIndividualUser", este inicia-se acedendo ao URL definido na fase *@BeforeTest* e depois utiliza os métodos definidos no *package Pages* para navegar pelos menus até chegar á página responsável pela inserção dos dados que permitam a criação de um novo Utilizador na interface gráfica da IPBrick OS. Após a introdução dos dados e criação de um novo utilizador o teste faz um *assert* para validar que o utilizador foi criado corretamente. Este *assert* é uma função da ferramenta *TestNG* que consiste em verificar que depois da introdução correta dos dados de um utilizador é devolvida uma mensagem de Sucesso. Caso esta mensagem não seja

devolvida o *assert* falha e o teste assume que algo falhou na criação de um novo utilizador e o teste consequentemente falha.

- @AfterMethod - Esta fase é responsável pela finalização do teste, onde o browser é encerrado, e os *cookies* são apagados.

```
42      @BeforeTest (alwaysRun=true)
43      public void setup() throws ParserConfigurationException, SAXException, IOException{
44          System.out.println("@BeforeTest: Setup");
45          driver = new FirefoxDriver();
46          driver.manage().window().maximize();
47          String timeString=objxml.getxml(ConfigXmlFile,"implicitlyWait");
48          long time=Integer.parseInt(timeString);
49          driver.manage().timeouts().implicitlyWait(time, TimeUnit.SECONDS);
50          baseUrl=objxml.getxml(ConfigXmlFile,"url");
51          baseUrl2=objxml.getxml(ConfigXmlFile,"url2");
52          baseUrlcafe=objxml.getxml(ConfigXmlFile,"cafe");
53          ip=objxml.getxml(ConfigXmlFile,"ip");
54      }
```

Figura 5.10: Fase @BeforeTest

```
55      @Test(groups={"AddIndividualUser"},priority=1)
56      public void AddIndividualUser() throws Exception{
57          System.out.println("@Test: Add Individual User");
58          objLogin = new LoginPage(driver);
59          objMainPage = new MainPage(driver);
60          objAdvDefPage= new AdvancedConfigsPage(driver);
61          objApplyConfigs= new ApplyConfigurations(driver);
62          objDirectory= new DirectoryPage(driver);
63
64          driver.get(baseUrl);
65          objLogin.loginToPage(objxml.getxml(ConfigXmlFile,"username"),objxml.getxml(ConfigXmlFile,"password"));
66          objAdvDefPage.goToDefenitionsPage();
67          objDirectory.AddIndividualUser();
68          objApplyConfigs.goToApplyConfigsPage();
69          objApplyConfigs.applyConfigs();
70          Assert.assertTrue(objMainPage.getSuccessfullyUpdated().contains("Successfully updated!"));
71          objMainPage.clickLogout();
72      }
```

Figura 5.11: Fase @Test

```
174     @AfterMethod(groups = {"AddIndividualUser","AddMassUsers"})
175     public void shutDownSelenium() {
176         driver.manage().deleteAllCookies();
177         driver.quit();
178     }
```

Figura 5.12: Fase @AfterMethod

5.3.3.3 *Package Data*

Este *package* contém alguns métodos com funções auxiliares. Durante a execução dos testes, por vezes é necessário que estes executem uma ligação *ssh* para fazer a verificação de alguns

parâmetros. Dentro deste *package Data* é onde estão definidas todas as classes responsáveis por métodos auxiliares tal como a função que executa comandos através de uma ligação *ssh*.

5.4 Solução Final

Esta secção tem como objetivo apresentar a solução final, começando por fazer uma apresentação detalhada de como a plataforma de testes automáticos pode ser configurada e como esta pode ser utilizada e colocada em prática.

5.4.1 Configuração e Utilização

Após ter sido apresentada a arquitetura, padrões aplicados, funcionamento e funcionalidades da plataforma de testes, passa-se à especificação dos passos necessários a seguir para configurar a plataforma, de modo a que esta fique pronta a ser utilizada.

5.4.1.1 Criação e configuração da Máquina virtual

Como foi explicado no presente capítulo, esta plataforma está montada sob uma máquina virtual. A máquina virtual para este projeto foi criada com recurso ao Archipel que é a ferramenta que a IPBrick SA usa para gerir, supervisionar, e criar máquinas virtuais. Para que a plataforma de testes funcione de uma forma eficiente, a máquina virtual que suporta esta plataforma deverá ter os seguintes requisitos:

- **Memory:** 6 GB;
- **Number of CPU's:** 6;
- **OS:** Ubuntu 14.04 LTS or superior;
- **Browser:** Mozilla Firefox
- **Install ssh:** apt-get install ssh;
- **Install git:** apt-get install git;

5.4.1.2 Instalação e configuração do Jenkins

Após ter a máquina virtual criada e configurada, o passo seguinte é proceder à instalação do Jenkins. Para isso basta correr os seguintes comandos:

- wget -q -O - https://pkg.jenkins.io/debian/jenkins-ci.org.key | sudo apt-key add -
- sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'
- sudo apt-get update
- sudo apt-get install jenkins

Depois de correr os comandos apresentados, podemos aceder à *interface* gráfica do Jenkins através do *browser*, introduzindo o seguinte endereço "localhost:8080", onde será possível encontrar uma página que nos indicará os passos seguintes a realizar para finalizar a instalação.

Figura 5.13: Painel inicial de configuração do Jenkins

O passo seguinte passa por configurar o Jenkins para que este integre todas as ferramentas necessários ao funcionamento dos testes automáticos. Para isso, no menu de configurações globais de ferramentas deve-se instalar o Java Development Kit bem como o Apache Ant, tal é possível ver nas Figuras 5.14 e 5.15

Figura 5.14: Instalação do *JDK* no painel de configurações globais do Jenkins

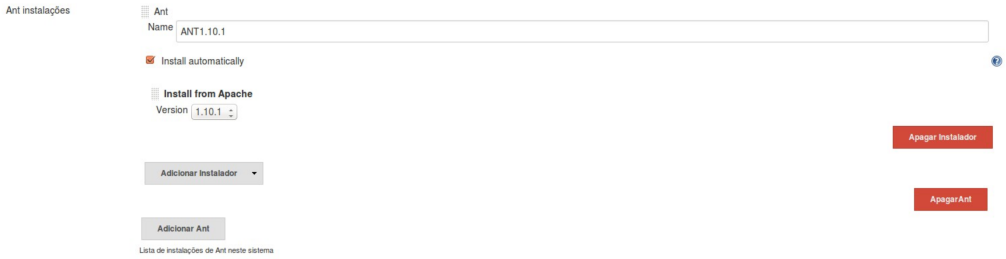

Figura 5.15: Instalação do Apache Ant no painel de configurações globais do Jenkins

Para completar a configuração do Jenkins é necessário instalar os *plugins* necessários para a apresentação e geração dos relatórios dos testes, bem como para a edição dos ficheiros de configuração. Os *plugins* necessários são:

- *Build timeout plugin*;
- *Extensible Choice Parameter plugin*;
- TestNG *Results Plugin*;
- *Text File Operations*.

5.4.1.3 Criação do Projeto no Jenkins

Depois de ter o Jenkins configurado e pronto a ser usado, basta incluir os testes automáticos nesta plataforma. Para isso, vamos iniciar um **Novo Projeto** selecionando a opção **Freestyle Project** como é possível ver na Figura 5.16.

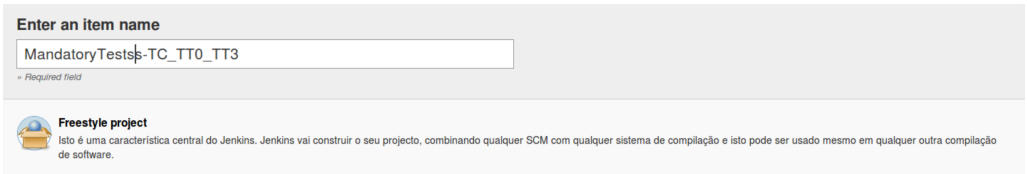

Figura 5.16: Criar Novo Projecto no Jenkins

Este passo vai criar um projeto vazio no Jenkins. O que se segue é a inclusão do código fonte relativo á automação dos testes. Para isso, no menu *Advanced Configurations* devemos incluir o *path* onde está localizada a pasta de todo o código fonte do projeto.

Figura 5.17: Inclusão do código fonte no Jenkins

Para finalizar o processo de criação de um novo projeto basta então adicionar mais duas configurações. Estas configurações são a definição da ferramenta que é utilizada para a compilação e a configuração da parametrização do projeto. A parametrização do projeto tem como objectivo a possibilidade de realizar uma seleção prévia dos testes que pretendemos executar.

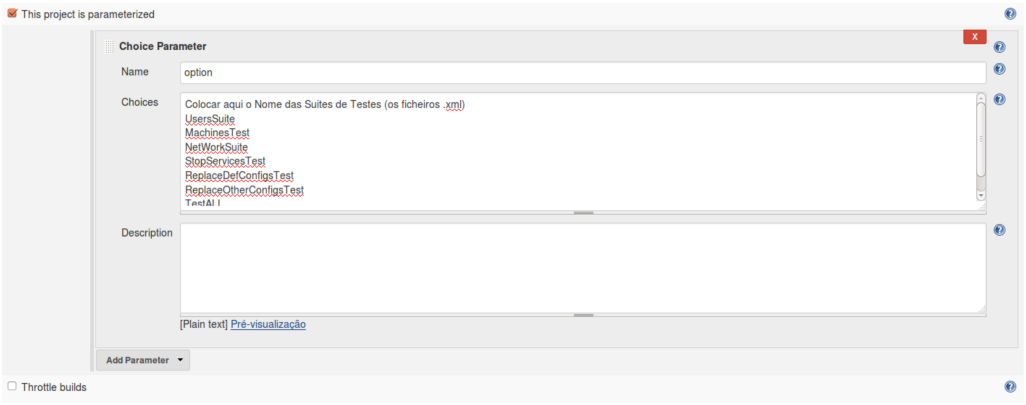

Figura 5.18: Parametrização do Projeto

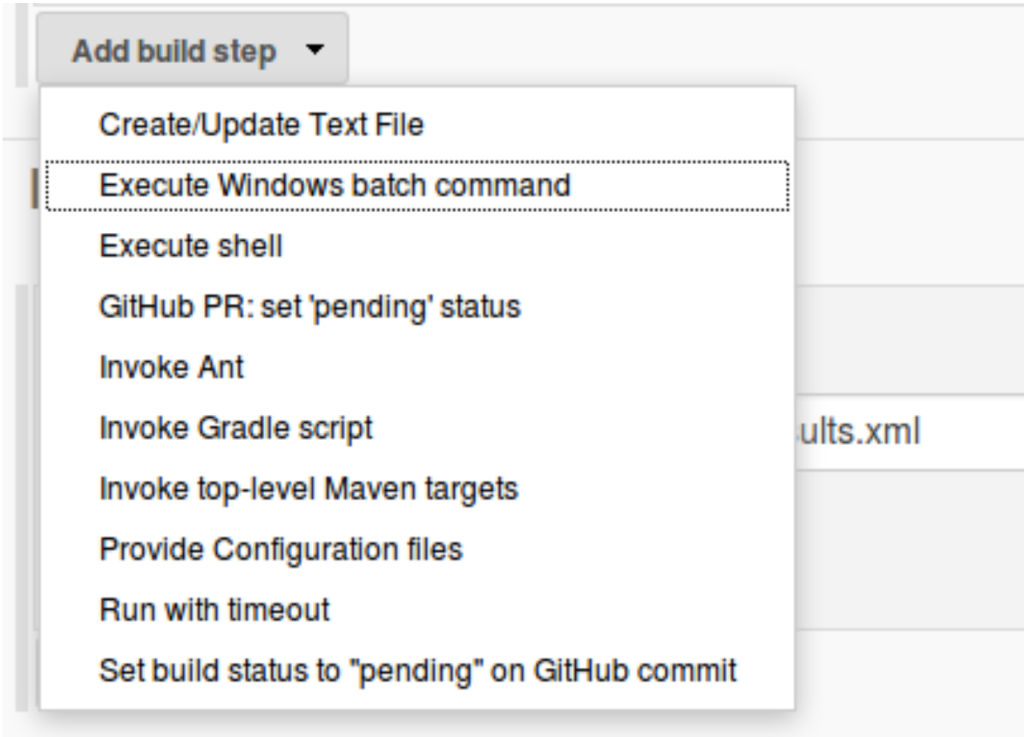

Figura 5.19: Escolha do compilador do Projeto

Com estes passos concluídos basta então aceder à página inicial do Jenkins, onde podemos encontrar o nosso projeto pronto a ser executado. Para executar os testes automáticos basta dar a

ordem de compilação do projeto através do botão "*Build*", o projeto será em seguida compilado e os testes automáticos serão executados.

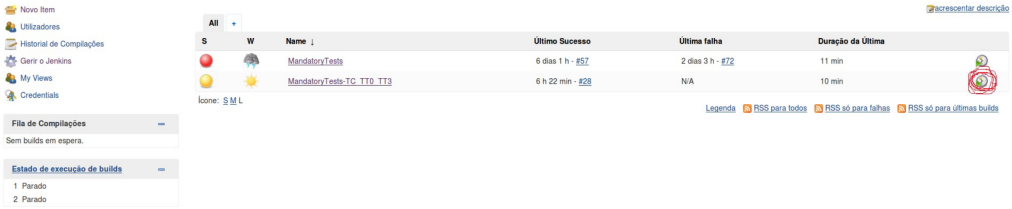

Figura 5.20: Página inicial do Jenkins com o projeto pronto a ser compilado

Capítulo 6

Conclusões e Trabalho Futuro

O presente Capítulo destina-se à apresentação das conclusões finais deste trabalho. Também será feita uma referência à satisfação dos objetivos, às dificuldades encontradas e ao trabalho futuro.

6.1 Satisfação dos Objetivos

De uma forma geral, os resultados obtidos foram satisfatórios. Os requisitos foram todos cumpridos e, com este projeto a IPBrick SA, tem uma base de automação estruturada por onde poderá começar a expandir os processos automáticos de teste e qualidade.

6.2 Dificuldades

Desde o início do projeto, foram encontradas várias dificuldades, nomeadamente a incerteza e ambiguidade daquilo que era pretendido por parte da IPBrick SA para este projeto. A IPBrick SA não possui nenhuma pessoa especializada a desempenhar funções de *tester*, nem manual nem automático. A inexistência de alguém com experiência neste campo refletiu-se numa dificuldade acrescida na concretização do projeto, pois tornou mais complicada uma orientação clara e concisa daquilo que são as boas práticas da automação de *software*.

Outra dificuldade consiste na inexistência de uma cultura atual de garantia de qualidade de *software* automatizada. A IPBrick SA, através deste projeto, está a dar os primeiros passos no que toca a automação de testes para o seu *software*, mas a ausência desta cultura de automação de testes traduziu-se na imposição de requisitos inapropriados, o que leva ao aumento da complexidade do teste. Um exemplo disso foi o requisito que pedia a possibilidade de configurar a entrada dos testes automáticos. Outra consequência da inexistência de uma cultura de automação de testes de *software* é a ausência de testes de unidade no código da IPBrick OS e seus produtos. Isto levou á necessidade de realizar testes mais complexos para colmatar a falta dos testes de unidade e permitir que estes fossem configuráveis.

Com o aumento da complexidade dos testes aumenta também o risco destes testes se tornarem *flacky*. Um teste *flacky* é um teste que pode falhar ou passar com sucesso quando executado exatamente nas mesmas condições.

6.3 Trabalho Futuro

Este projeto é apenas o inicio de um trabalho importantíssimo no que toca à garantia de qualidade dos produtos da IPBrick SA. Este não é um projeto estático, em vez disso é um projeto continuo que exige uma constante manutenção e melhoria sucessiva. Para as melhorias e trabalho futuro, com o objetivo de aumentar a utilidade complexidade e confiança dos resultados obtidos a partir do projeto, propõe-se a realização das seguintes tarefas:

- Criar testes de unidade - Com o objetivo de criar uma estrutura sólida e sustentável de testes automáticos a IPBrick SA deve fazer este investimento e criar a cultura do desenvolvimento de testes de unidade no seu *software*. Este investimento pode ser bastante dispendioso mas que deve ser considerado como uma prioridade, pois trará um retorno positivo, não só a nível financeiro, como do ponto de vista de satisfação por parte dos recursos humanos da organização, que poderão dispender mais tempo a desenvolver e criar do que a testar.

- Automatizar a instalação e configuração das máquinas virtuais - Este é um processo regular quando se pretende testar uma atualização do IPBrick OS. A utilização de *containers* ou de ferramentas pode ser útil na automação deste processo que é bastante moroso.

- Criar uma *Delivery Pipeline* - Esta é uma das tarefas mais importantes. A criação de uma *Delivery Pipeline*, como é possível ver na Figura 6.1, consiste em criar um processo com vários ambientes. Estes ambientes têm como objetivo ser alvo dos diferentes tipos de teste para que, sempre que é feito um *Merge* do código para a *Master Branch*, os testes sejam executados, garantindo de uma forma correta que as alterações feitas pelos programadores não causam erros de regressão no produto final, dando assim um verdadeiro significado aos Testes de Regressão.

- Reduzir o número de testes *flacky* - Como foi mencionado no Capítulo anterior, os testes grandes e complexos podem tornar-se *flacky*. Uma tarefa importante para trabalho futuro será reduzir a complexidade dos testes, tornando-os mais simples e concisos reduzindo assim a possibilidade de estes falharem;

- Reduzir a quantidade de tempo de execução dos testes - Partindo do princípio que a tarefa anterior será executada, não é pertinente ter um tempo de compilação ou *build* de cada ambiente muito demorado. O tempo de cada *build* não deve exceder os 10 minutos e, tendo em conta que os testes fazem parte da *build* do ambiente, o tempo que estes demoram a ser executados também deve ser reduzido. Para reduzir o tempo de execução dos testes pode ser pertinente *mockar data*. *Mockar data* consiste em criar um servidor *mock* auxiliar que

simula o comportamento de objetos reais de forma controlada. São normalmente criados para testar o comportamento de outros objetos. Os objetos *mock* são objetos falsos que simulam o comportamento de uma classe ou objetos reais para que o foco do teste seja na funcionalidade a ser testada;

- Criar testes não funcionais - Criar um plano de testes não funcionais com o objetivo de realizar principalmente teste de carga e performance ao IPBrick OS e seus produtos.

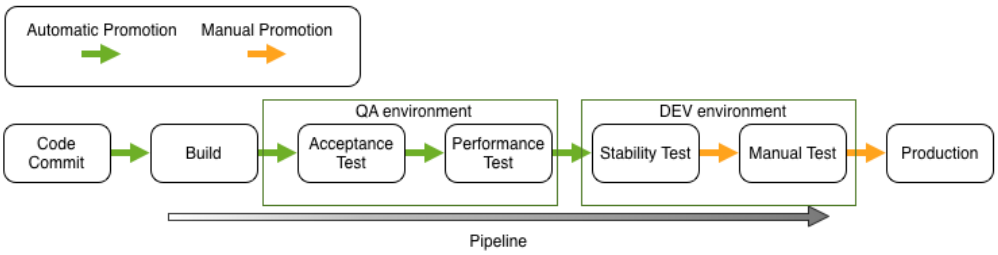

Figura 6.1: Exemplo de uma *Delivery Pipeline*

Referências

[1] Jerry Gao, H-SJ Tsao, e Ye Wu. *Testing and quality assurance for component-based software*. Artech House, 2003.

[2] Sunita Sangwan Isha. Software testing techniques and strategies.

[3] Jan Rooijmans, Hans Aerts, e Michiel Van Genuchten. Software quality in consumer electronics products. *IEEE software*, 13(1):55–64, 1996.

[4] Alexandre Bartié. *Garantia da qualidade de software*. Gulf Professional Publishing, 2002.

[5] IEEE Standard 610-1990. Ieee standard glossary of software engineering terminology, ieee press. 1991.

[6] Kshirasagar Naik e Priyadarshi Tripathy. *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011.

[7] ISTQB. "ISTQB Foundation Level and Agile Tester Certification guide, Junho 2017. http://istqbexamcertification.com/.

[8] tutorialspoint. "tutorialspoint softwaresting, Junho 2017. http://www.tutorialspoint.com/software_testing/.

[9] IPBrick. "Ipbrick os - ipbrick, Junho 2017. http://www.ipbrick.com/pt-pt/ipbrick-os/.

[10] André Osório de Castro Ferreira. Controlo e monitorização do parque informático. 2011.

[11] IPBrick. "Ipbrick.I - Servidor Intranet, Junho 2017. http://www.ipbrick.com/pt-pt/produtos/ipbrick-i/.

[12] Rui Miguel da Silva Carvalho et al. Integração do iportaldoc com sistemas erp. 2012.