FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Análise e Melhoria de um Processo de Garantia de Qualidade de Software em Contexto Empresarial

Miguel Lira Barbeitos Luís

Mestrado Integrado em Engenharia Informática e Computação

Orientador: João Carlos Pascoal Faria

3 de Julho de 2019

Análise e Melhoria de um Processo de Garantia de Qualidade de Software em Contexto Empresarial

Miguel Lira Barbeitos Luís

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Ana Cristina Ramada Paiva
Arguente: Miguel António Sousa Abrunhosa Brito
Vogal: João Carlos Pascoal Faria
3 de Julho de 2019

Resumo

Nos tempos de hoje, a qualidade dos produtos desenvolvidos por uma empresa é considerada uma prioridade. Garantir a qualidade dos produtos tem vindo a ser uma grande aposta no mercado empresarial, tentando criar mecanismos que permitam cada vez mais melhorar a qualidade.

Existem equipas especializadas para que esta qualidade seja garantida, e que o produto chegue às mãos dos clientes o mais próximo possível do esperado. Para tal é necessário analisar a melhor maneira de organizar uma equipa, e ainda de analisar todo o modelo de negócio no qual se vai trabalhar, analisar as arquiteturas usadas e ainda os processos e ferramentas existentes, usadas na empresa para caso de estudo.

Neste documento é tido como propósito, a análise de todo o processo de qualidade existente dentro de uma empresa, de modo a descobrir lacunas para melhoria do mesmo.

Foi analisado o processo de garantia de qualidade da empresa, tendo sido identificadas duas oportunidades de melhoria: automação de testes de validação de lançamento e introdução de testes de mutação para avaliar a qualidade da bateria de testes.

Relativamente à automação de testes de validação de lançamento constatou-se a existência de um trabalho manual significativo nas validações após lançamento em ambiente de produção, com análise dos gráficos representativos do fluxo de dados, das interfaces dos utilizadores e ainda ferramentas auxiliares desenvolvidas para estas validações.

Após análise dos componentes desse trabalho, recomendou-se a automação destas mesmas validações, passando por criar uma bateria de testes para validar os dados que circulam pelas ferramentas auxiliares com testes mais genéricos. Seria executada a bateria de testes aquando do término de uma entrega para ambiente produtivo, deixando para o analista de qualidade o trabalho manual de validação de certos cenários mais específicos, dos gráficos e das interfaces de utilizador.

Relativamente aos testes de mutação, foi avaliada a viabilidade da sua implementação na empresa, com o objetivos de dispor de medidas fiáveis de avaliação e melhoria da qualidade das baterias de testes desenvolvidas. Foi selecionada a ferramenta *Stryker4s* para aplicação de testes de mutação, tendo sido aplicada a 3 módulos de um projeto escrito em *Scala*.

Concluiu-se que os testes de mutação realizados com esta ferramenta fornece informação útil para avaliar e melhorar a qualidade das baterias de testes, mas o tempo de execução muito longo praticados com a versão da ferramenta utilizada, torna impraticável a execução de testes de mutação. Por isso, a adoção de testes de mutação é apenas aconselhável quando os problemas de desempenho da ferramenta em questão forem resolvidos.

Abstract

Nowadays, the quality of products developed by a company is considered a priority. Ensuring the quality of the products has been a great bet in the business market, trying to create mechanisms that allow improving the quality.

There are specialized teams to achieve, as close as possible, the costumers' expectations related to this quality. To do this, it is necessary to analyze the best way to organize a team, and also to analyze the entire selected business model, analyze the architectures used and also the processes and frameworks existing and used in the company for the case study.

The purpose of this dissertation is to analyze the entire quality assurance process used in the company, in order to discover gaps to improve it.

The company's quality assurance process was analyzed and two improvement opportunities were identified: automation of post-release validation tests and introduction of mutation testing to evaluate the quality of the tests suite.

Regarding the automation of post-release validation tests, it was verified the existence of significant manual work in the release of a production environment, such as analysis of representative graphics of the data flow, user interfaces and also auxiliary tools develop for these validations.

After analyzing the components of this work, it was recommended to automate these validations, creating a test suite to validate the data that circulates through the auxiliary tools used with more generic cases. The test suite would be executed upon completion of the release to production, leaving the quality analyst to manually validate certain more specific scenarios and analyze the graphs and the user interfaces.

Concerning the mutation tests, the feasibility of its implementation in the company was evaluated, with the objective of having reliable measures to evaluate and improve the quality of the tests suite developed. The Stryker4s tool for mutation testing was selected and applied to 3 modules of a project written in Scala.

It was concluded that the mutation tests performed with this tool provide useful information to evaluate and improve the quality of tests suite, although the very long execution time practiced with this version of the tool makes it impractical to perform mutation tests. Therefore, the adoption of mutation testing is only advisable when the performance issues of this tool are solved.

Agradecimentos

Um dia, algures no tempo, ouvi que a gratidão é uma memória do coração. É um conjunto de ações e momentos que ficam marcados e registados para que nunca nos esqueçamos de quem nos quer bem.

Na realização da presente dissertação, múltiplas pessoas, de uma forma direta ou indireta, tiveram influência no percurso, e grato estou por terem cruzado o meu caminho. Deixo em seguida os contributos dados para a conclusão desta fase:

- Ao orientador responsável pela dissertação, Professor Doutor João Carlos Pascoal Faria, pela orientação prestada, pelo incentivo e apoio que sempre demonstrou. Aqui exprimo a minha gratidão.
- Ao Engenheiro António Rodrigues, membro da empresa Blip e responsável também pela orientação da dissertação, pela orientação e preocupação diária, pelo incentivo e disponibilidade demonstrada, deixo aqui uma mensagem de gratidão.
- A toda a equipa em que estive inserido na empresa Blip, pela boa disposição, aprendizagens e por me receberem de um modo único. Estou grato a todos.
- À família por mim escolhida, os meus amigos, deixo aqui o meu obrigado do fundo do coração. Agradecer apenas pelo tempo de realização desta dissertação seria pouco, obrigado por todos os anos e todas as memórias vividas.
- Por último, à minha família, sem vós nenhum objetivo ou sucesso, por mim alcançado, seria possível. Pais, irmã e toda a família, do fundo do coração, quero aqui deixar a minha maior gratidão. Obrigado.

Todos os meus sucessos derivam do que sou, e de quem fez de mim o que hoje sou.
A todos, o meu **Muito Obrigado**!

O Miguel Lira Barbeitos Luís

"Success is not the key to happiness.
Happiness is the key to success.
If you love what you are doing, you will be successful."

Albert Schweitzer

Conteúdo

CONTEÚDO

Lista de Figuras

Lista de Tabelas

Abbreviations

API	*Application Programming Interface*
QA	*Quality Assurance*
CI	*Continuous Integration*
PSTQB	*Portuguese Software Testing Qualification Board*
HTTP	*HyperText Transfer Protocol*
URI	*Uniform Resource Identifier*
US	*User Story*
LOC	*Lines Of Code*
CR	*Code Review*
PR	*Peer Review*
VCS	*Version Control System*
DM	*Delivery Manager*
PO	*Product Owner*

Capítulo 1

Introdução

Numa era marcada pela tecnologia, e pelo grande avanço que esta área tem tomado, cada vez mais existe uma preocupação constante com a qualidade dos produtos ou serviços que as empresas fornecem. Esta é uma área na qual as empresas tendem a fazer grandes investimentos, pois a sua imagem e a sua reputação são marcadas pela qualidade colocada nos produtos ou serviços que fornecem aos seus clientes.

Uma vez que assim é, a garantia de qualidade tem que estar sempre assegurada, e no ramo da tecnologia, é necessário que em todas as fases os produtos tentem sempre estar de acordo com o esperado, existindo cada vez mais equipas focadas apenas em garantir esse nível de qualidade.

Neste capítulo serão apresentados o enquadramento no qual a presente dissertação se insere, bem como uma explicação dos motivos que levaram ao aparecimento do problema em questão. Podem também ser encontrados os objetivos estabelecidos, para que no fim, exista um comparativo entre estes objetivos e os resultados obtidos. É ainda descrita uma visão global da estrutura do presente documento, para um melhor entendimento do plano seguido para a elaboração desta dissertação.

1.1 Enquadramento

A presente dissertação enquadra-se na área da qualidade de um projeto, desenvolvido num ambiente de integração contínua. Dentro da área da qualidade, o foco será na garantia da qualidade de um projeto, desde fases iniciais até à sua entrega final ao cliente.

É uma área que tem ganho um maior ênfase nos últimos tempos, e que as empresas tentam cada vez mais investir de modo a que os produtos, por eles desenvolvidos, se aproximem tanto quanto possível às expectativas dos seus clientes e utilizadores.

É importante que a qualidade dos produtos esteja sempre assegurada em todas as fases do desenvolvimento do mesmo, e não apenas no momento final para entrega ao cliente. É então que entra todo o processo de garantia de qualidade, que permite assegurar que desde o levantamento

dos requisitos com o cliente, a qualidade do produto seja tida em conta como uma prioridade e que sejam esclarecidos, sempre que surjam, as dúvidas ou problemas. Assim é possível que erros que surjam em fases prematuras do projeto, não sejam prolongados para fases posteriores e que a correção dos mesmos se torne mais complexa e dispendiosa do que aquando do seu surgimento.

Aliado à garantia de qualidade desde as fases prematuras do projeto, está um ambiente de integração contínua que visa realizar entregas frequentes e a integração dos componentes individuais. É onde testes de garantia de qualidade podem ser aplicados, e se verifiquem os erros existentes. Sendo importante garantir que o projeto esteja em correto funcionamento como um todo e não como componentes individuais, é vantajoso este tipo de ambientes pois erros de compatibilidades podem surgir, bem como um leque de outras situações. Este tipo de ambiente permite a criação de diferentes testes automatizados e outras validações, como já referido, que garantem os padrões de qualidade esperados.

A dissertação será desenvolvida em meio empresarial, mais propriamente na empresa Blip. Com base na cidade do Porto, a Blip é uma empresa de engenharia de software, fundada em 2009, em crescente número de trabalhadores, contando já com cerca de 300. No ano de 2017, foi eleita a melhor empresa para trabalhar em Portugal, melhor empresa para desenvolvimento profissional, melhor local de trabalho e ainda melhor empresa tecnológica para trabalhar. A Blip faz parte do grupo *Paddy Power Betfair*, uma empresa que está na *API Billionare's Club* ao lado do *Twitter*, *Facebook* e *Google* e que se mantém à frente da concorrência devido aos softwares por eles criados, sendo que o código desenvolvido pela empresa dá origem a aplicações utilizadas por mais de 5 milhões de pessoas em todo o mundo.

1.2 Motivação e Objetivos

No decorrer de uma fase na qual as empresas vêm a apostar na área da qualidade, é importante que exista um conhecimento de garantia de qualidade e ainda a interpretação de problemas complexos e dividi-los em problemas menores para a melhor resolução.

É necessário que sejam entendidas várias áreas para que as decisões tomadas sejam as mais acertadas possíveis. Na base destas decisões, existe a familiarização com o normal funcionamento de um *API*, a interação com diferentes tipos de clientes, e o impacto e importância que o software desenvolvido terá no mundo de trabalho onde este se insere.

Embora há uns tempos atrás a indústria de testes estivesse muito focada em testes de *front-end*, tem vindo a evoluir e a apostar em todo o tipo de testes. Ainda não existe um grande leque de documentação focada em todas as áreas de testes, mas novas portas se têm aberto neste sentido. Existe muita pesquisa e muitas ferramentas que têm vindo a surgir como auxílio e como base deste mercado em evolução. A presente dissertação é também uma oportunidade dentro desta área de enorme importância no desenvolvimento de um software.

Assim, o objetivo do trabalho descrito nesta dissertação consistiu na análise do processo de garantia de qualidade numa empresa em concreto, com vista à identificação e demonstração (com provas de conceito) de possíveis melhorias. Foram então identificados objetivos mais específicos

que permitiriam atingir o objetivo final de identificação das melhorias ao processo existente. Sendo assim:

- Inicialmente teria que ser realizado todo o levantamento do processo de garantia de qualidade;
- Análise do mesmo, de modo a encontrar oportunidades de melhoria;
- Análise da viabilidade da implementação dessas melhorias, como automatização das validações manuais em ambiente de produção e implementação de testes de mutação afim de avaliar a qualidade da bateria de testes.

Uma melhoria encontrada foi a automação de testes de validação em ambiente produtivo, tentando diminuir o tempo necessário para executar estas validações. Uma outra oportunidade de melhoria identificada consistiu na introdução de testes de mutação, com vista a dispor de medidas fiáveis de avaliação e melhoria das baterias de testes.

1.3 Estrutura da Dissertação

Servindo de um resumo de cada capítulo que o presente documento contém:

- Capítulo 2 - São apresentados os conceitos fundamentais e o estado da arte existente para a área na qual esta dissertação se insere, de modo a fornecer um conhecimento sobre os tópicos que foram abordados ao longo do desenvolvimento da presente dissertação;
- Capítulo 3 - É apresentado neste capítulo o levantamento do processo de garantia de qualidade praticado na empresa em estudo, seguido de uma análise do mesmo, de modo a encontrar oportunidades de melhorias ao processo. É ainda apresentada a viabilidade da automação de testes de validação em ambiente de produção, tendo sido esta uma das melhorias sugeridas ao processo;
- Capítulo 4 - Contém a análise a uma das melhorias que foi sugerida para melhorar o processo de garantia de qualidade. Aborda a análise da adoção de testes de mutação a ser integrados na *pipeline* usada, onde foi selecionada uma ferramenta para aplicação dos testes, foi também realizada uma experimentação com configurações padrão, e com configurações adaptadas. Por fim, foram levantadas as lições aprendidas com as experimentações realizadas;
- Capítulo 5 - Termina com as conclusões gerais tiradas sobre o tempo de desenvolvimento desta dissertação, todas as contribuições existentes que tornaram possível a conclusão desta dissertação e por fim, sugestões de trabalho futuro a realizar sobre a área na qual se desenvolveu este documento.

Capítulo 2

Conceitos Fundamentais e Estado da Arte

De modo a fornecer todo o conhecimento necessário para entendimento da área que está a ser abordada, serve a presente secção, onde serão fornecidos os conhecimentos fundamentais para que os leitores consigam entender de modo claro o que será tratado.

Poderá então aqui ser encontradas secções relacionadas com gestão de garantia de qualidade, testes de software, desde modelos de software, de níveis de testes, de tipos de testes e ainda de processo de testes.

Ainda neste capítulo são apresentados conceitos relacionados com arquiteturas compostas por micro-serviços, e os desafios que são colocados na execução de testes quando se adota este tipo de arquiteturas.

Para terminar este capítulo, é possível encontrar uma secção de testes de mutação, onde são explicados como funciona todo o processo de testes de mutação, exemplo prático do que acontece quando aplicados, são referidas as vantagens e dificuldades que este tipo de testes trás, e ferramentas existentes para auxiliar todo este processo.

2.1 Garantia de Qualidade

Segundo a Associação Portuguesa de Testes de Software (PSTQB), a garantia de qualidade é "parte da gestão da qualidade que se foca em fomentar confiança de que os requisitos de qualidade serão atingidos"[GO11].

A gestão da qualidade permite que uma empresa consiga gerir a qualidade dos seus produtos, estabelecendo objetivos e políticas de qualidade, abordando quatro fases como planeamento, controlo, garantia e melhoria da qualidade.

Ou seja, permite que a imagem da empresa seja mantida, pois os seus produtos chegarão às mãos dos seus clientes com a qualidade esperada, tendo que os profissionais da área passar por um processo sistemático de realização de testes ao longo do processo de desenvolvimento do software.

Por norma, em contexto empresarial, usando como referência a empresa em estudo, o termo "garantia de qualidade"é usado para englobar atividades de testes, revisão de código, revisão a pares, validações manuais e entrega de versões para ambientes de produção.

De um modo geral, as funções relacionadas com a área da garantia de qualidade passam por [Whaa]:

- Transferência de tecnologia - onde é obtido um documento do design do produto, dados das tentativas e erros e ainda da sua avaliação. Estes documentos são distribuídos, verificados e aprovados;
- Validação - com a preparação do plano e aprovação dos critérios de avaliação para os testes;
- Documentação - Controlar os documentos existentes e cada alteração a qualquer documento tem que realizar todo o procedimento de aprovação;
- Assegurar a qualidade dos produtos;
- Planos de melhoria de qualidade.

Especificando, as atividades relacionadas com esta área são [Whab]:

- Criar um plano de gestão de garantia de qualidade;
- Identificar pontos de verificação, baseados nos objetivos e no alvo a avaliar;
- Aplicar técnicas necessárias para avaliar o que foi desenvolvido;
- Realizar técnicas de revisão formal, como reuniões técnicas - que na empresa em estudo, são designadas de *sharing session* ou *design session*;
- Aplicar várias técnicas de testes, para que os resultados sejam mais fiáveis;
- Realizar avaliação do produto, de modo a verificar que os requisitos levantados foram implementados;
- Realizar monitorização do processo, para que todos as etapas sejam realizadas e da forma correta;
- Controlar a mudança, através do uso de procedimentos manuais e uso de ferramentas automáticas;
- Medir o impacto da mudança;
- Comparar os ciclos de vida da nova versão com a versão estável;

- Documentar todos os resultados obtidos.

Podemos então afirmar que existir um profissional de garantia de qualidade, como uma entidade ativa no desenvolvimento do software, é vantajoso e boa prática, sendo uma prática cada vez mais adotada pelas empresas.

2.2 Testes de Software

Antes de serem apresentados os diferentes tipos e níveis de teste, convém esclarecer primeiro o que é um teste. Um teste é um "conjunto de um ou mais casos de teste", sendo que, um casos de teste pode ser definido como um "conjunto de valores de entrada, de pré-condições de execução, de resultados esperados e de pós-condições de execução, desenvolvido para um objetivo específico ou condição de teste, como seja, o acionar de um determinado ramo de execução ou a verificação do cumprimento de um requisito específico"[GO11].

Para que haja um maior entendimento sobre testes, são apresentados, na tabela 2.1, os sete princípios que estão associados à execução de testes de software:

"Os testes mostram a presença de defeitos"	"Os testes mostram a presença de defeitos, mas não provam a inexistência dos mesmos"
"Testes exaustivos são impossíveis"	"Testar tudo (considerando todas as combinações de entradas e pré-condições) não é viável, exceto para os casos triviais"
"Testar cedo"	"Para detetar os defeitos mais cedo, as atividades devem ser iniciadas o mais cedo possível no ciclo de vida do desenvolvimento do software ou sistema, e devem estar focados nos objetivos definidos"
“Agrupamento de defeitos”	“O esforço de testes deve estar focado proporcionalmente à densidade de defeitos por módulo esperada e mais tarde observada”
"Paradoxo do pesticida"	“A repetição exaustiva dos mesmos casos de teste leva à não deteção de novos defeitos. (...) os casos de teste devem ser regularmente analisados e revistos e testes diferentes ou novos devem ser desenvolvidos”
"Os testes são dependentes do seu contexto"	"Os testes são efetuados de forma diferente em diferentes contextos"
"Falácia da ausência de erros"	"Detetar e corrigir defeitos por si só não ajuda se o sistema construído for impossível de utilizar e não satisfazer as necessidades e expectativas dos seus utilizadores"

Tabela 2.1: 7 princípios de execução de testes de software [Bat18]

2.2.1 Níveis de Teste

A frequência com que se verificam mudanças, é o que define a integração contínua (CI) [DDS14]. Vários casos de estudo têm vindo a provar que diminui os períodos de lançamentos/entregas e que o *feedback* recebido é mais frequente. Podendo designar-se como uma maneira de conduzir o desenvolvimento de um software [MSB17].

É uma prática caracterizada por múltiplas junções de sub-projetos ou projetos, com vista a construir um software completo e compacto que deriva do desenvolvimento individual dos seus componentes, obtendo no final o software completo. Um ambiente assim caracterizado permite que os erros sejam isolados em cada componente, trazendo rapidez na correção destes erros, resultando num software mais coeso [SB14].

Testes e o desenvolvimento de software têm as suas atividades relacionadas, isto é, "testes não existe isoladamente". Com isto, podemos deduzir que "os diferentes modelos de ciclo de vida de desenvolvimento necessitam de abordagens diferenciadas para os seus testes"[Int11]. Para o caso em concreto da empresa em caso de estudo, foi usado o modelo em V como modelo de ciclo de vida de desenvolvimento.

O modelo em V é usado para representar as atividades de teste desde o início de uma aplicação até ao seu desenvolvimento [MP16]. A sua representação mais comum pode ser vista na figura 2.1.

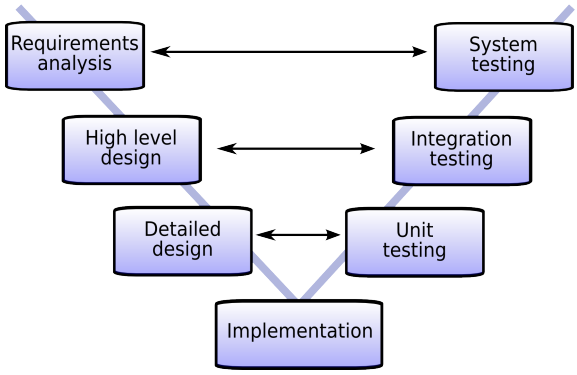


Figura 2.1: Modelo em V [Vmo]

Então, os quatro níveis mais utilizados no modelo que está a ser abordado, e que também se encontra representado na figura 2.1, são [Int11]:

- Testes de componentes (unidade)
- Testes de integração
- Testes de sistema
- Testes de aceitação

Como existem várias variantes e vários contextos em que este modelo é aplicado, "na prática, um modelo em V pode ter mais, menos ou diferentes níveis de desenvolvimento e testes, dependendo do projeto e do produto de software"[Int11]. Na prática o modelo em V pode ser aplicado iterativa e incrementalmente ao longo do desenvolvimento do projeto.

Para que seja possível um melhor entendimento, prossegue-se com a descrição de cada nível.

Testes de componentes incluem testes a pequenas partes do software para verificar o seu comportamento. Isto aplica-se a componentes que possam ser testados separadamente. É o nível de testes mais baixo, realizado principalmente pelo desenvolvedor para testar a unidade de código em questão [HSS15].

Os testes de integração testam as interfaces entre componentes, e interações entre as diferentes partes do sistema [Int11]. Devem ser realizados frequentemente pois, quanto maior for o âmbito da integração, mais difícil se torna o isolamento de falhas num sistema ou num componente específico, o que pode conduzir a um aumento do risco e do tempo adicional para resolução de problemas [MP16]. Existem vários tipos possíveis de abordagens a tomar na realização de testes de integração, podendo ter uma abordagem *bottom-up* ou *top-down*, que são duas abordagens relacionadas com a hierarquia dos componentes no software. Uma outra abordagem possível é partes críticas em primeiro lugar e ainda uma abordagem *Big Bang* onde é realizada uma integração, de uma só vez, de todos os módulos [HSS15].

Os testes de sistema testam todo o comportamento dentro de um sistema/produto [Int11]. É conveniente que o ambiente onde este tipo de testes são realizados sejam o mais similar com o ambiente destinado ao objetivo final, para que o número de falhas relacionadas seja o mais significativas possíveis. Os testes de sistema devem investigar os requisitos funcionais e não funcionais do sistema, e características de qualidade de dados [MP16].

Os testes de aceitação são um teste final do sistema ser aceite para o seu uso final. São muitas vezes da responsabilidade do PO ou dos clientes/utilizadores finais do sistema [Int11]. Realizados tendo por base os requisitos do utilizador, para que se teste o comportamento do sistema confirmando se o resultado é o esperado [MP16].

"O nível de testes está ligado às responsabilidades num projeto", sendo que é definido como um "conjunto de atividades de teste que são organizadas e geridas em conjunto"[GO11].

Pode então ser falado, para cada nível de teste [Int11]:

- Objetivos genéricos
- "Produto(s) de trabalho de referência para derivação de casos de teste"
- Objetos de teste
- "Defeitos mais comuns e falhas a detetar"
- "Requisitos de equipamento de teste e ferramentas de suporte"
- "Abordagens específicas e responsabilidades"

2.2.2 Tipos de Teste

Um conjunto de atividades de teste pode ser destinado à verificação do sistema (ou parte de um sistema) com base num objetivo ou alvo específico de teste, levando à origem dos diferentes tipos de teste existente. Podem ser considerados quatro principais tipos de teste, testes funcionais, testes não-funcionais, testes estruturais e testes relacionados com alterações (re-teste e testes de regressão). A realização de testes pode estar associada a diferentes propósitos, como por exemplo, servir para melhorar a qualidade, realizar verificação e validação e ainda estimar a confiabilidade do software.

Os testes funcionais consideram o comportamento externo do software. Os testes funcionais testam aquilo que é feito pelo sistema. Os testes funcionais são baseados em funções e características (...) e a sua interoperabilidade com sistemas específicos, e podem ser executados em todos os níveis de teste [Int11].

O termo testes não funcionais descreve os testes necessário para medir as características dos sistemas e software que podem ser quantificadas numa escala variável [Int11]. São exemplos de testes não funcionais [MP16] [HSS15]:

- Testes de Usabilidade - determinam, para novos utilizadores, o quão fácil será para eles executar tarefas básicas;
- Testes de Carga - visam compreender o comportamento que o sistema tem quando submetido a uma carga específica requerida;
- Testes de Volume - testa o sistema, sob a quantidade enorme de dados, de modo a verificar o seu limite;
- Testes de *Stress* - testar também o limite do sistema sob condições desfavoráveis;
- Testes de Desempenho - testa o comportamento tomado pelo sistema;
- Testes de Configuração - usando todas as configurações possíveis para a qual o sistema foi construído, é testado o software em todas essas possibilidades;
- Testes de Compatibilidade - em complemento do anterior, é usado para verificar o grau de satisfação do cliente, e que verifica o comportamento do sistema nas configurações para as quais não foi projetado.

Um outro tipo de teste existente, são os testes de alterações. É boa prática e aconselhável a confirmação do que foi testado, então depois de um defeito ser detetado e corrigido, o software deve ser testado novamente para confirmar que o defeito original foi removido com sucesso. Dentro dos testes de alterações, temos ainda os testes de regressão, que são aplicados aquando de uma modificação, de modo a repetir testes num programa já anteriormente testado. Existe a possibilidade de realizar, nos níveis todos, os testes de regressão, uma vez que compreendem "testes funcionais, não funcionais e estruturais". "Os testes devem ser repetíveis para que possam ser usados como testes de confirmação e como suporte aos testes de regressão"[Int11].

Depois do desenvolvimento e entrega de um software, ele estará ativo e em serviço durante alguns períodos de tempo. Um bom planeamento de atualizações ou alterações ao sistema, permite que possam ser programados, com a devida antecedência, testes de manutenção. "Os testes de manutenção são executados num sistema operacional existente, e são desencadeados por modificações, migração, ou desabilitação do software ou sistema". Além de se testar aquando das situações anteriormente mencionadas, ainda realiza testes de regressão para as secções que não sofreram alteração. "Os testes de manutenção podem ser difíceis, se as especificações estiverem desatualizadas, forem inexistentes, ou os testadores (*testers*) com o conhecimento necessário não estiverem disponíveis"[Int11].

2.2.2.1 Técnicas de Conceção de Casos de Teste

Também os testes estruturais, podem ser realizados nos diferentes níveis de testes existentes, mas "as técnicas estruturais são mais bem-sucedidas se aplicadas depois das técnicas baseadas nas especificações, de forma a auxiliar a medição do rigor dos testes através da avaliação da cobertura de um tipo de estrutura". As técnicas de testes estruturais são importantes no sentido de que medem a quantidade da estrutura que foi testada, sendo aconselhável que se tente atingir 100% da estrutura. Caso tal não se verifique, devem ser desenvolvidos mais testes. "Testes estruturais podem ser baseados na arquitetura do sistema, tais como, uma hierarquia de chamadas"[Int11]. Um exemplo de teste estrutural é teste da Caixa-Branca (*White-Box*).

2.2.3 Processo de Testes

O processo de testes de um software não se limita apenas à execução dos mesmos. Cada processo de testes pode ser adaptado às necessidades, tanto do cliente ou utilizador como do software propriamente dito. Como tal, o processo de testes possui várias fases, que mesmo que por norma sigam uma lógica sequencial, pode acontecer de certas atividades se sobreporem ou ocorrerem mesmo em simultâneo.

Então,as atividades de teste incluem [Int11]:

- Planeamento e controlo;
- Escolha das condições de teste;
- Conceção e execução dos casos de testes;
- Verificação dos resultados obtidos;
- Avaliação dos critérios de saída;
- Informar as partes envolvidas sobre o estado do processo e do sistema sob teste;
- Finalizar ou completar as atividades de encerramento.

Na primeira fase, serão definidos os objetivos dos testes, especificando as atividades de testes associadas. Esta atividade tem atenção a todo o *feedback* retornado do controlo e monitorização. Embora nesta fase se encontre a atividade de controlo de testes, é algo que acontece constantemente, realizando comparações entre o planeado e o que realmente foi efetuado [Int11].

Na análise e conceção de testes são criados os casos de teste e as condições de teste a partir dos objetivos gerais. Existem várias tarefas associadas à análise e à conceção de testes como:

- Revisão da base para testes;
- Avaliação da testabilidade da base para testes e dos objetos de teste;
- Identificação e priorização das condições de teste;
- Conceção e priorização de casos de teste de alto nível;
- Conceção da configuração do ambiente de teste e identificação das necessidades da infraestrutura e ferramentas;
- Criação da rastreabilidade bidirecional entre a base para testes e os casos de teste.

No que diz respeito à implementação e à execução de testes, pode ser definida como uma atividade de especificação de procedimentos e *scripts* de teste. Engloba tarefas como [Int11]:

- Finalizar, implementar e priorizar os casos de teste;
- Desenvolver e priorizar dos procedimentos de teste e a criação de dados de teste;
- Criar conjuntos de teste a partir dos procedimentos de teste de modo a alcançar uma execução de testes eficiente;
- Verificar que a configuração do ambiente é a correta;
- Verificar e atualizar, de forma bidirecional, da rastreabilidade entre a base para testes e os casos de teste;
- Executar os procedimentos de teste, tanto manual como usando ferramentas, de acordo com a sequência planeada;
- Registar os resultados da execução dos testes e guardar as identidades e versões do software sob teste;
- Reportar discrepâncias que tenham ocorrido como incidentes e proceder com a análise destes, a fim de encontrar a sua causa.

Falando agora da avaliação do critério de saída, é a atividade onde a execução do teste é avaliada face aos objetivos definidos. Deve ser executada para cada nível. Engloba a tarefas como:

- Verificação dos registos de teste face ao critério especificado para a saída aquando do planeamento dos testes;

- Avaliação da existência de necessidade de executar mais testes ou o critério de saída deve ser alterado;
- Escrever o relatório de testes, fornecendo-o às partes interessadas.

Finalizando com fecho da fase de testes, estes pressupõem a recolha de dados das atividades de teste já encerradas a fim de consolidar experiências, *testware*, factos e números. Marcos no projeto são definidos para esta fase do processo, e podem ser marcos de lançamento do sistema de software ou a conclusão/cancelamento de um projeto, um marco do projeto que foi alcançado ou uma manutenção do projeto foi concluída. Envolve tarefas como [Int11]:

- Verificação das entregas planeadas;
- Fecho de relatórios dos incidentes ou registo de eventuais alterações para os que ainda estão abertos;
- Documentar a aceitação do sistema;
- Finalização e arquivo do *testware*, assim como o ambiente de teste e a infraestrutura para uma futura reutilização;
- Fornecer à entidade responsável pela manutenção o *testware*;
- Análise das lições aprendidas para que se possam saber as alterações em lançamentos ou projetos futuros;
- Usar toda a informação que foi recolhida tentando melhorar a maturidade dos testes.

2.3 Teste em Arquiteturas de Micro-Serviços

Na empresa em análise, as soluções de software desenvolvidas são baseadas em arquiteturas compostas por micro-serviços, que colocam desafios específicos do ponto de vista dos testes.

Arquiteturas baseadas em micro-serviços tem ganho, cada vez mais, lugar no meio empresarial e nas arquiteturas que as empresas adoptam [PJ16].

Os micro-serviços permitem que sejam feitas atualizações aos mesmos de forma independente, uma vez que cada micro-serviço corre o seu próprio processo, comunicando através dos mecanismos existentes com os outros micro-serviços [Bak17]. A manutenção de cada micro-serviço fica facilitada, a sua construção ou melhoria é algo que pode também ser facilitado, uma vez que aquilo que se utiliza está apenas relacionado com o próprio micro-serviço [PJ16][Bak17].

No fundo, permite mudar uma era de uso de arquiteturas monolíticas, e usar arquiteturas em que se tenta separar serviços grandes em serviços pequenos para que seja possível realizar implementações de novas funcionalidades, correção de erros existentes e ainda criação de novos serviços.

A ideia de separar uma aplicação em componentes mais manuseáveis está presente em diversas abordagens para *design* de software, como Arquitetura Orientada a Serviços (*SOA - Service Oriented Architecture*. Um micro-serviço pode ser caracterizado como organização de um sistema, isto é, uma maneira de organizar uma aplicação simples. Embora pareça igual à ideia da divisão da aplicação, são diferentes, sendo que uma das principais diferenças é o facto de micro-serviços serem considerados pequenos serviços que fornecem funcionalidades únicas e específicas, enquanto que em SOA os serviços fornecidos estão inseridos na mesma estrutura [dCSMS16].

Para as aplicações monolíticas, usar uma arquitetura de micro-serviços não resolve todos os problemas. Esta arquitetura apresenta desafios como:

- Complexidade de orquestrar a implementação de múltiplos serviços;
- Gerir e controlar os testes de cada serviço;
- Maturidade da equipa para coordenar mudanças;
- Definir as fronteiras de um serviço;
- Controlar bibliotecas partilhadas;
- Reutilizar código.

Os desafios quando se adota uma arquitetura de micro-serviços incluem os mesmos encontrados em sistemas distribuídos no geral: comunicação inter-serviços, coordenação entre serviços e transações distribuídas. De facto, micro-serviços trouxeram outro desafio, que é dividir corretamente a aplicação em vários componentes [dCSMS16].

Em [dCSMS16] é usada uma abordagem que "consideram testes de desempenho *end-to-end*"(ponta-a-ponta), o que significa que é usada a rede como forma de comunicação, por onde o cliente realiza o pedido para o serviço alvo. A arquitetura apresentada visa resolver os desafios e problemas na execução de testes em micro-serviços, com maior foco em testes de desempenho. Dentro dos testes, com maior foco na especificação dos mesmos e em como podem ser providenciados pelo serviço para testar a aplicação. Usando especificação *HTTP* como base, a implementação da solução é facilmente realizada pelos seus desenvolvedores.

A solução encontrada, possui dois conceitos fundamentais, a especificação e o mecanismo usado para a especificação realizada. A especificação permite o acesso externo, por parte das aplicações, aos parâmetros necessários para realizar os testes. Já o mecanismo permite saber como definir a especificação dentro da aplicação e em como as aplicações externas conseguem extrair a especificação e conteúdo desta para a execução de testes. A arquitetura pode ser vista na figura 2.2.

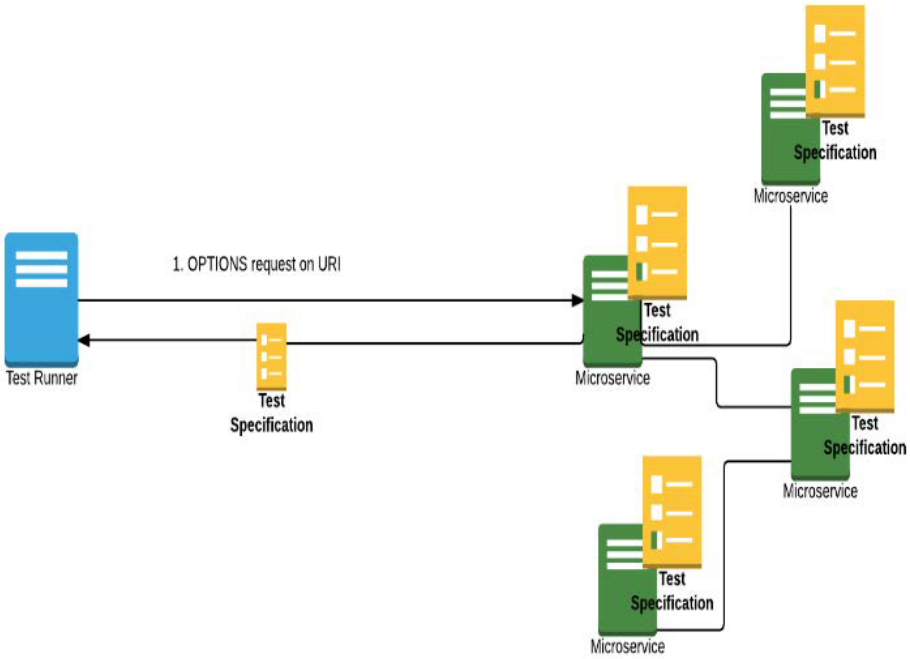


Figura 2.2: Arquitetura da Proposta de Solução para Testes de Micro-Serviços [dCSMS16]

Como referido, foi usado, para exemplificar, o funcionamento da arquitetura, uma resposta a um pedido que use o método *HTTP OPTIONS* num *URI* específico. Este método, permite também que o sistema forneça a informação relativa aos seus recursos e a solução proposta usa um método que não afeta os outros métodos, como *POST* ou *GET*, já fornecidos pelo serviço. Então um documento, com todos os métodos disponíveis, exemplos de dados a usar num pedido, a descrição com a semântica dos métodos e dados de validação, é fornecido na especificação. Na secção 3.2 é descrita uma ferramenta capaz de ser utilizada na arquitetura aqui descrita.

Uma vez que a presente dissertação se insere no tema de automatização de testes, em [dCSMS16] é apresentada a implementação de uma ferramenta que permite a realização de testes de desempenho automatizados, desenvolvida em linguagem *Java* e que usa uma outra ferramenta, *Spring Boot*. O modo de operação usado é filtrar todos os pedidos *HTTP* feitos ao micro-serviço, e sempre que um pedido *OPTIONS* for encontrado, a ferramenta gera a especificação requerida e fornece testes completos ao pedido *URI*. Usando anotações de *Java*, esta ferramenta foi implementada de modo a fornecer ao desenvolvedor, uma opção clara e fácil de construir métodos para retornar a especificação do teste.

A ferramenta é dividida num artefacto para a *API* e outro para a implementação. Na *API* é inserida a especificação propriamente dita, em substituição das anotações e da classe. Já na implementação, está contido o filtro que lida com os pedidos ao serviço e ainda as classes principais que lidam com o processo de construir a especificação de teste. Em tempo de execução, a *API* procura a devida classe e, se a implementação não for encontrada, a ferramenta é desligada para que o serviço possa continuar sem esta. Através de reflexão para retirar a especificação para obter

a devida resposta, permitindo que a ferramenta não interfira com outros pedidos que estejam a ocorrer. A implementação realizada permite também que, sem exigir necessidade de mudanças de código, a ferramenta seja removida de forma simples.

Entrando em detalhe para o interesse dos desenvolvedores, para que comecem a utilizar as anotações e a ferramenta propriamente dita, é necessário:

1. Colocar no serviço a biblioteca da *API*;
2. Implementar os métodos das especificações, um para cada operação;
3. Adicionar aos filtros da rede o filtro da ferramenta que será de facto chamada.

A ferramenta consegue encontrar os métodos necessários para que a execução de testes de desempenho seja executada, através das anotações *Java*. É então retornado um projeto, em *JSON* contendo os parâmetros necessários para a invocação do serviço. A anotação de *Java* guarda o diretório, a descrição e o método que providenciou a operação [dCSMS16].

2.4 Testes de Mutação

A escrita de testes unitários é uma prática comum, que traz um rápido *feedback* acerca de como o código se comporta, podendo servir também de documentação, podendo garantir melhores princípios de *design* e ainda garantir que qualquer mudança, feita acidentalmente, não compromete a funcionalidade.

Mas, o que nos garante que os testes unitários não realizam apenas cobertura de código?

Como garantir que realmente os testes unitários testam adequadamente o código e as suas funcionalidades e fluxos?

Surge então os testes de mutação, considerados um tipo de teste de caixa branca, com o propósito de avaliar a qualidade da bateria de testes existente, ajudando o desenvolvedor a criar testes eficazes ou localizar pontos fracos nos dados de testes usados. Os testes de mutação envolvem a modificação de um programa de maneiras pequenas. Os testes detetam e rejeitam cada mutante criado, fazendo com que o comportamento do programa original seja diferente do mutante. Isto é designado de morte do mutante, e a bateria de testes são avaliadas pela percentagem de mutantes mortos.

Um processo de testes de mutação passa, por norma, por 4 fases [Str]:

- Introdução de erros - Alteração do código fonte de uma maneira muito localizada, injetando erros representativos de erros reais;
- Execução de testes unitários - Corre os testes unitários sobre código original e sobre código mutante;
- Compara resultados - Criando um relatório da aplicação das mutações;
- Análise dos mutantes e da sua relevância.

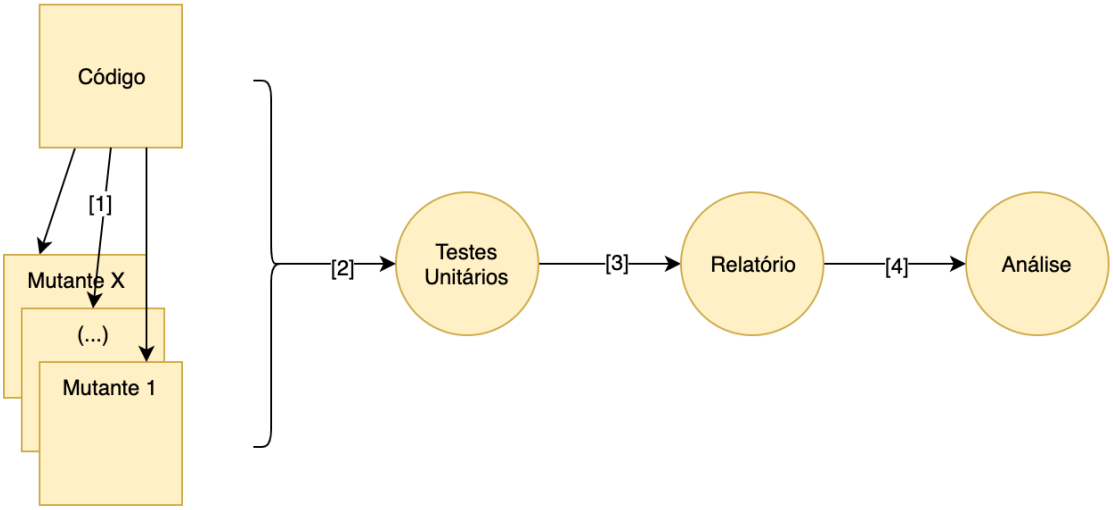


Figura 2.3: Processo de Mutação

Descrevendo agora, de um modo mais específico, a aplicação do processo de mutações ilustrado na figura 2.3, em [1], por norma, existe a utilização de ferramentas responsáveis pela alteração do código e consequentemente de todo o processo. Estas ferramentas começam por analisar o código e, dependendo dos mutantes que estão ativos, marcam no código os locais onde as mudanças serão aplicadas.

Após colocar todas as marcas no código, são corridos os testes unitários que estão presentes, fase [2], onde para cada local específico em que foram deixadas as marcas, são testadas as diferentes hipóteses de mutação. Através da comparação entre testes aplicados ao programa original e ao programa mutante, os resultados são comparados e para cada mutante existem duas hipóteses, ou morrem ou sobrevivem:

- Se forem mortos, significa que foram previstos casos em que as alterações mutantes poderiam surgir, e foram criados testes para tal - o código encontra-se devidamente implementado e testado pelos testes unitários.
- Se sobreviverem, significa que nem todas as hipóteses foram consideradas e que ainda existem situações que podem ser testadas - o código necessita ser revisto e analisado os mutantes aplicados de modo a descobrir fronteiras por testar.

Por fim, em [3], é criado um relatório (em consola e/ou *html*), onde é apresentado a taxa dos mutantes mortos (*mutation score*), tempo de execução e ainda os mutantes que foram aplicados no código e o seu estado, morto ou sobrevivente. A taxa dos mutantes é baseada na quantidade de mutantes mortos,

$$taxa = \frac{mortos}{mortos + sobreviventes} * 100 \tag{2.1}$$

, o que significa que representa a percentagem de mutantes que foram mortos e consequentemente, mostra a qualidade dos testes unitários que foram escritos. Por norma, e como nem todos os mutantes aplicados são relevantes para o contexto onde aparecem, é necessária uma análise de

todos os mutantes para se poder concluir que hipóteses estão ausentes, fase [4]. É de notar que será impossível atingir uma taxa de mutantes mortos de 100%, então, os valores limiares dos estados de sucesso para taxa de mutantes mortos é escolhido pela equipa que aplica os testes de mutação.

Assim, os testes de mutação podem ser usados para avaliar a qualidade dos testes unitários [PKZ+19]. Para ilustrar os testes de mutação, como descrito na secção do estado da arte, vejamos o exemplo da figura 2.4, retirado de [Str].

```
function isUserOldEnough(user) {
  return user.age >= 18;
}
```

Figura 2.4: Função maior de idade (em *JavaScript*)

Imaginemos que é criada a função apresentada na figura 2.4 para determinar se uma certa pessoa possui mais do que 18 anos de idade. Aquilo que uma ferramenta de testes de mutação vai fazer é aplicar algumas mudanças ao código original (mutações), como por exemplo o que é apresentado na figura 2.5.

```
/* 1 */ return user.age > 18;
/* 2 */ return user.age < 18;
/* 3 */ return false;
/* 4 */ return true;
```

Figura 2.5: Modificações aplicadas

Por exemplo, na figura 2.6, encontra-se um exemplo de testes que seriam testados.

```
describe("Age", function(){
    it("checking age", fucntion(){
        u1 = createUser("Manuel", 19);
        u2 = createUser("Maria", 17);
        u3 = createUser("Joao", 18);

        assert(true, u1);
        assert(false, u2);
        assert(true, u3);
    });
});
```

Figura 2.6: Exemplo de teste

Explicando, para o programa original, os testes desenhados passam com sucesso, uma vez que para idades iguais ou superiores a 18 o resultado é verdadeiro, e para inferiores o resultado é falso. Mas olhando para as alterações que os testes de mutação fazem, figura 2.5, o u1 é morto nas modificações 1 e 4, sobrevivendo nas modificações 2 e 3. Com o u2 acontece o oposto do u1, e com o u3, que possui uma idade igual à fronteira, vai ser morto na modificação 4, sobrevivendo nas restantes. Permite assim, concluir que os testes implementados estão a cobrir todos os cenários possíveis, ganhando assim confiança nestes mesmos testes.

De um modo geral, os testes de mutação permitem [PKZ+19]:

- Ganhar confiança nos testes criados de acordo com o valor do *mutation score*;
- Descobrir falhas, para melhoria dos testes, acrescentando novos testes para matar mutantes sobreviventes.

A adoção de testes de mutação no processo de garantia de qualidade permite [Sta][Mut]:

- Descobrir novos tipos de erros;
- Descobrir defeitos mais específicos e difíceis de encontrar;
- Descobrir ambiguidades e redundâncias;
- Fazer *debugging*, de um modo mais fácil, e fazer manutenção de um produto;
- Ganhar confiança com uma métrica de mutantes mortos *versus* mutantes sobreviventes, do que apenas cobertura de código.

Por outro lado, a aplicação de testes de mutação necessita ser devidamente pensada, uma vez que [Sta][Mut]:

- É custosa em termos de processo e tempo de execução;
- É complexa a aplicação e a análise dos seus resultados;
- Não é aplicável a testes de caixa-preta (*black box testing*).

No que toca a testes de mutação, muitas investigações sobre o tema têm surgido nos últimos anos.

Com isto, em [PK18] foi explicado o processo e condicionantes para testes de mutação, e em [JMH11] foi descrita uma aplicação de testes de mutação a nível industrial. Após análise, os autores de [JMH11] concluíram que o processo de análise de mutantes é deveras dispendioso em termos de tempo uma vez que são criados imensos mutantes. Conseguiram também perceber que para que os resultados possam ser interessantes e com alguma relevância, há vantagem em saber onde aplicar os mutantes e que tipo de mutantes aplicar.

Para se proceder com a análise das mutações aplicadas, os autores de [JMH11] criaram um esquema que reflete o processo a executar para a análise. A figura 2.7 mostra o esquema por eles desenhado.

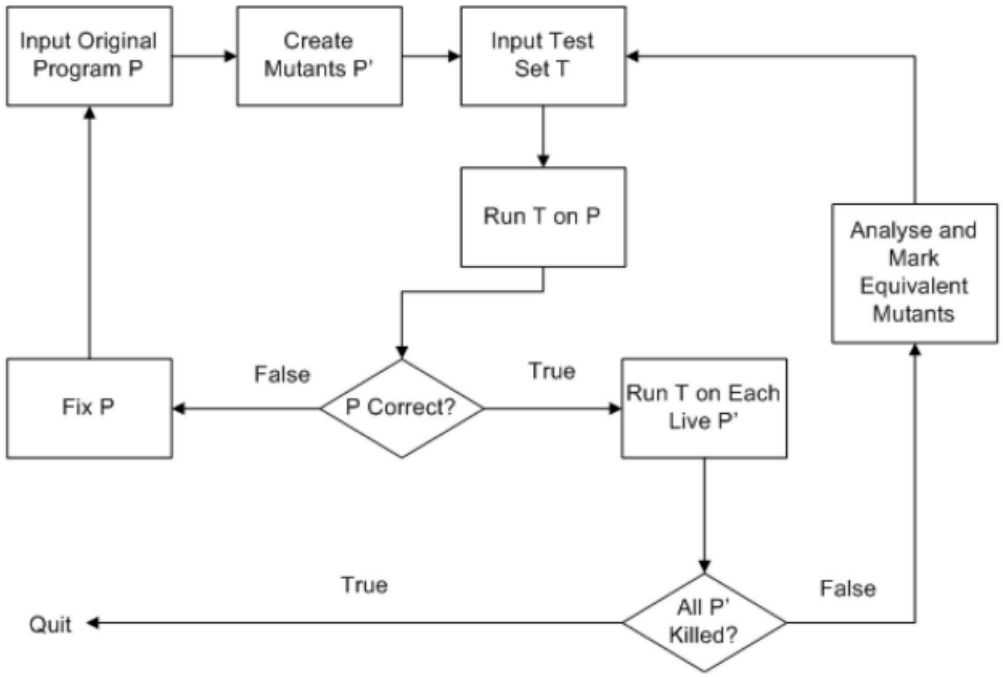


Figura 2.7: Processo Genérico de Análise de Mutantes [JMH11]

Explicando resumidamente, o programa original P é injetado com alterações ao seu código, originando código mutante P'. De seguida é executada uma bateria de testes T sobre o código original P; caso P tenha erro, procede-se com a correção dos mesmos e executa-se o até agora descrito. Caso o código P não tenha erros, são executados agora os testes (T) sobre o código mutante P'. Caso todos os mutantes sejam mortos, o processo termina; caso contrário, são analisados manualmente os mutantes sobreviventes para que se possa perceber que erros estão a acontecer, e volta a correr-se a bateria de testes sobre o código original P mas com os mutantes sobreviventes localizados, podendo perceber como melhorar a bateria de testes [JMH11].

Com a aplicação industrial realizada em [PK18], 3 lições foram retiradas das aplicações acima referidas sobre testes de mutação: nem todos os mutantes podem ser mortos; a análise de mutantes é temporalmente dispendioso; a execução dos testes de mutação é também moroso; e a seleção dos parâmetros a usar influência os resultados obtidos.

Concluíram também que a aplicação a nível industrial tem de facto muitas adversidades e desafios ainda, mas que pesquisas devem ser realizadas cada vez mais para que possa vir a ser vantajoso a aplicação de testes de mutação a nível industrial.

Um outro estudo [HSF+19] foi realizado onde foram comparadas a aplicação de testes de mutação a nível do código fonte ou a nível de representações intermédias. Foi usada, como auxilio, uma ferramenta *SRCIROR* que tem a funcionalidade de conseguir implementar exatamente os mesmos operadores de mutação em ambos os níveis, tomando providências de mutantes repetidos e outras variáveis para que os resultados fossem viáveis.

O estudo foi aplicado a 15 programas diferentes para que a amostra fosse suficiente. Com o

término das execuções na amostra de programas, procedeu-se com as conclusões, sendo que se verificou uma melhor taxa de mutação ao nível do código fonte, apesar das taxas serem próximas.

Em [ZZH+18] é proposta uma abordagem Preditiva de Teste de Mutação (PMT - *Predictive Mutation Testing*), em que é caracterizada por ser um modelo de classificação. que tem por base funcionalidades relacionadas com os mutantes e com a bateria de testes, sendo que usa um modelo para conseguir prever onde os mutantes irão aparecer, mortos ou sobreviventes, sem executar os testes de mutação propriamente ditos.

Para que se perceba como PMT funciona, explica-se de seguida a abordagem tomada pelos autores. Foi usada aprendizagem de máquina (*machine learning*) para resolver o problema que estava em mãos. A abordagem é ilustrada na figura 2.8.

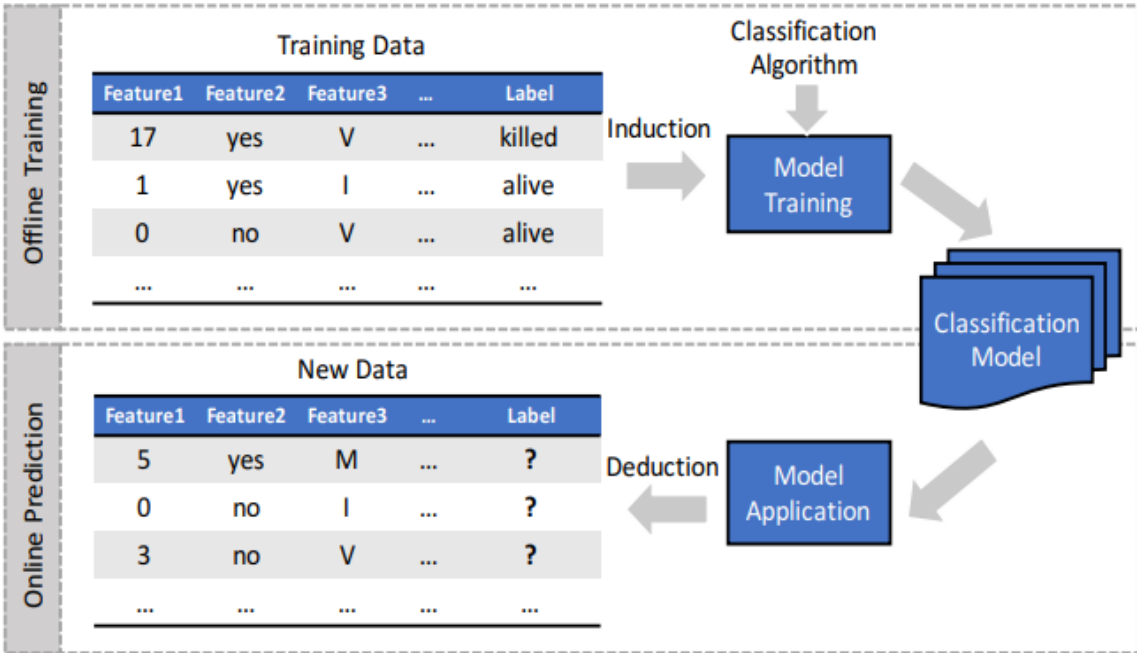


Figura 2.8: Ferramenta genérica PMT [ZZH+18]

O modelo de classificação é que vai permitir prever os resultados das mutações, sem as executar. O algoritmo de aprendizagem escolhido foi o algoritmo *Random Forest*, que é poderoso, flexível e consegue interpretar classificadores de árvores.

O PMT consegue utilizar várias ferramentas distintas e várias baterias de testes, e os resultados obtidos pela aplicação do PMT e relatada pelos autores em [ZZH+18], permitiu concluir que em termos de eficiência houve melhorias no processo, tendo existido apenas uma pequena perda de precisão.

Para um bom leque de linguagens de programação, começam a aparecer diversas ferramentas de testes de mutação, diferenciando-se no nível de maturidade. Para a linguagem de *Java*, existe uma ferramenta de testes de mutação muito conceituada, chamada de PIT. Tendo em vista que a eficácia dos testes de mutação está diretamente relacionada com os mutantes usados, os autores de

[LPK+], procederam com a criação de um conjunto de mutantes a serem injetados na ferramenta, executando de seguida a ferramenta sobre o programa.

Com a execução conseguiram provar que em projetos reais, é de certa forma fácil matar os mutantes aplicados originalmente pelo PIT, bem como de outras ferramentas existentes. Isto levantou preocupações sobre os resultados que o PIT oferecia aos seus utilizadores, achando por bem conferir os resultados utilizando marcas e executando casos de testes mais adequados ao contexto para que certezas sobre os resultados pudessem ser tiradas.

Decidiram então proceder com a implementação na ferramenta PIT deste conjunto de mutantes que eles experimentaram, designando o conjunto de mutantes estendidos (*extended mutants*), que se encontra disponível para o uso de todos desde a altura da análise realizada.

Capítulo 3

Análise do Processo de Garantia de Qualidade

No presente capítulo é abordado todo o processo de qualidade da empresa em estudo.

É realizado o levantamento do processo existente, para que seja possível de seguida proceder com a análise do mesmo de modo a encontrar lacunas para melhoria.

Com a análise efetuada foram encontradas duas possíveis melhorias, automação de testes de validação após entrega de versões e ambiente de produção e a possível adoção de testes de mutação como medida fiável de avaliação e melhoria de testes de qualidade.

Este capítulo terminar com a análise da automação de testes de validação em ambiente produtivo, de modo a perceber o que realizar para a automação e que vantagens traria realmente para o processo.

3.1 Levantamento do Processo de Garantia de Qualidade Existente

A empresa em caso de estudo, tem o seu próprio processo de garantia de qualidade que pode variar de equipa para equipa, isto é, pode sofrer alterações para que o processo seja mais eficaz consoante os elementos e a forma de trabalhar dentro de cada equipa.

A empresa usa uma metodologia *SCRUM*, que permite tirar maior vantagens e benefícios do processo de garantia de qualidade que a empresa utiliza [SQN]. O processo de garantia de qualidade tem as suas etapas distribuídas por pontos específicos da estrutura de etapas usada, *SCRUM*, e encontra-se aqui descrito apenas essas etapas do processo consideradas. Todas as equipas aqui se baseiam para proceder com o desenvolvimento dos seus produtos.

Muito utilizada na gestão de projetos ágeis, permite tornar o processo de desenvolvimento mais eficaz, uma vez que existe uma recolha de *feedback* mais rápida, podendo ajustar-se o produto, a cada iteração, consoante as expectativas das partes interessadas.

Um diagrama dos estágios de uma metodologia *SCRUM* pode ser representado pela figura 3.1.

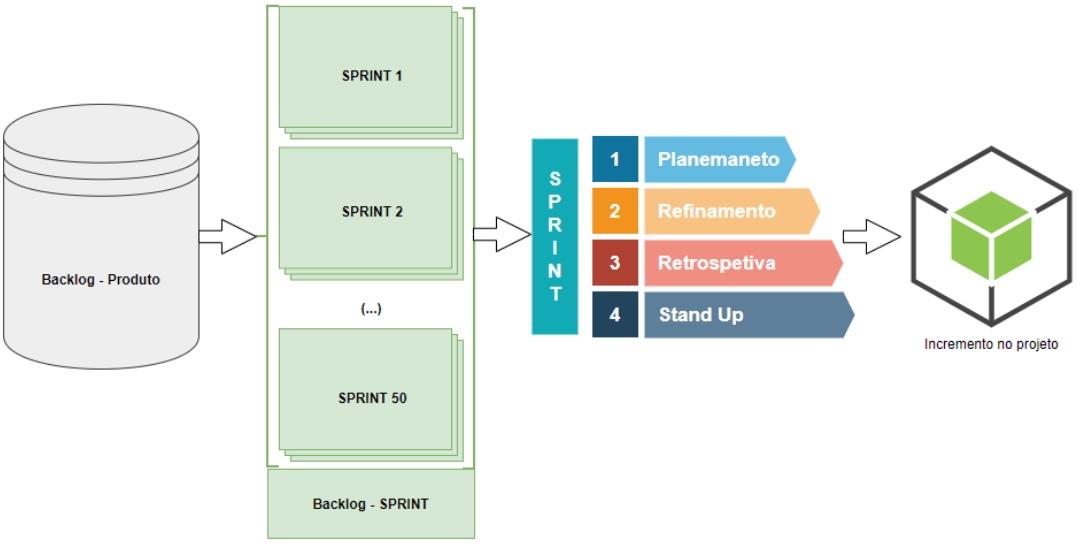


Figura 3.1: Diagrama de *SCRUM*

Antes de se proceder com a explicação do diagrama apresentado, convém apresentar os papéis presentes dentro de cada equipa. Existem 4 diferentes tipos de papéis, sendo eles:

- Gestor de Entregas (*Delivery Manager* (DM)) - responsável por garantir que a equipa entrega algo com valor para as partes interessadas, mantendo a equipa unida e focada nos objetivos estabelecidos. Por norma, existe um DM por equipa, sendo todos os problemas relatados a ele, bem como o estado do trabalho individual e coletivo.

- Proprietário do produto (*Product Owner* (PO)) - responsável por transmitir a sua visão sobre o produto em desenvolvimento ou a desenvolver, utilizando o *backlog* para listar as funcionalidades, organizadas por prioridades, que se designam por história do usuário (*user story)*. Para que as *releases* possam ser feitas pelos desenvolvedores, é necessária aprovação do PO para que tal se suceda. É o único elemento que faz parte da equipa mas não tem que responder perante o DM, sendo que também existe apenas um por equipa.

- Analista de Garantia de Qualidade (*Quality Assurance Analyst* (QA)) - responsável pelo processo que visa garantir que o produto desenvolvido tem a melhor qualidade possível. Existindo um por equipa, o seu papel é deveras importante. Possui uma visão diferente dos outros membros de equipa, de modo que tenta prever eventuais erros futuros para que possam ser implementadas medidas que não deixem que esses erros venham a ocorrer, poupando tempo e custos de correção futuros.

- Desenvolvedor (*Developer*) - responsável por analisar pormenores e o trabalho a realizar para cada US, procedendo de seguida com a implementação da mesma. Além da implementação da US propriamente dita, o desenvolvedor partilha tarefas com o resto da equipa, tais como:

- Implementação de testes
- Revisão por pares (*Peer Review* (PR))
- Revisão de código (*Code Review* CR)
- Entrega de versões (*Releasing builds*)
- Validações manuais

Estas são tarefas que qualquer elemento da equipa está habilitado a realizar, sendo que PR e CR, maioritariamente, são realizadas pelos desenvolvedores, as validações manuais em ambiente de produção são grande parte das vezes realizadas pelo analista de qualidade, e as restantes são realizadas de forma distribuída, ou quando exigido, por qualquer elemento da equipa.

O tempo que leva ao desenvolvimento de um produto, é por norma contabilizado em número de *sprints*, sendo que cada *sprint* tema duração de 2 semanas.

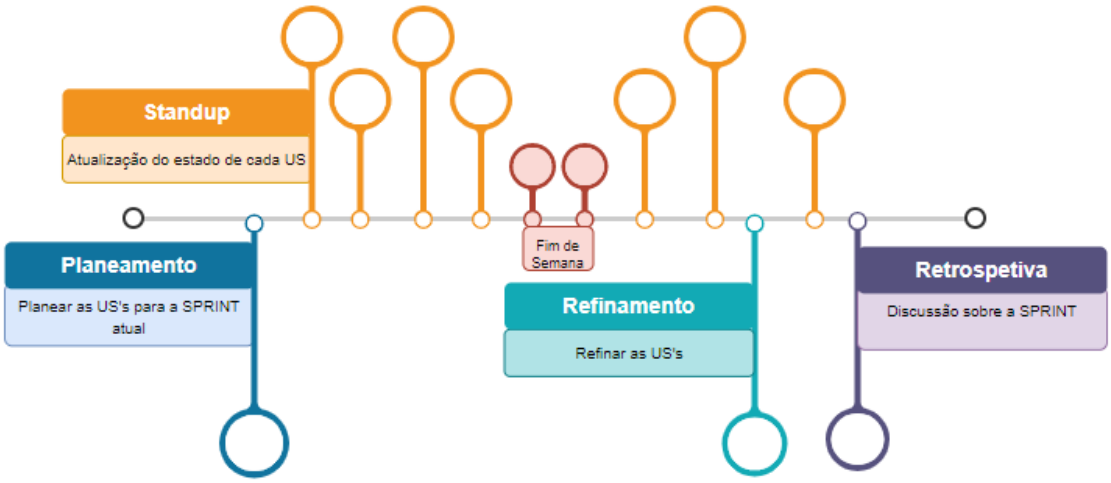


Figura 3.2: Constituição de uma *sprint*

Na figura 3.2 é visível a constituição de uma *sprint*, composta por duas semanas, estando todo o trabalho, para essa *sprint*, programado para este tempo. O trabalho que compõe uma *sprint* é composto pelas *user stories* criadas. Uma *user story* (US) é um requisito de utilizador de alto nível, usado no desenvolvimento de software ágil, e normalmente aparece em linguagem corrente descrevendo que funcionalidade um utilizador precisa e a razão por trás disso, qualquer critério não funcional e também inclui critérios de aceitação.

Apresentando de uma forma breve cada fase de uma *sprint*:

- Planeamento (*Planning*) - é por esta fase que se inicia, onde o seu intuito é, como o nome indica, planear o trabalho para a *sprint* atual. É analisado todo o trabalho contido no *backlog* do produto, por norma organizado por prioridade, para decidir o trabalho a realizar e atualizar o *backlog* da *sprint*.

- *Standup* - é uma reunião diária, exceto nos dias de planeamento e retrospetiva, primeiro e último dia da *sprint* respetivamente, que tem lugar antes de se iniciar o dia de trabalho e onde o propósito é cada elemento falar sobre a(s) sua(s) US(s), informando o trabalho que realizou no dia anterior e o trabalho que conta realizar durante o dia. No final de cada *Standup* existe um momento, quando justificado, de discussão de problemas que surgiram no dia anterior e que é do interesse de toda a equipa ter conhecimento sobre, designado de *parking lot*.

- Refinamento (*Grooming*) - tem lugar perto do final de uma *sprint* de modo a que sejam refinadas US's que estão no topo do *backlog* do produto e que estejam na iminência de entrar no *backlog* da *sprint* seguinte. Nada mais é do que discutir cada US e perceber o trabalho que é necessário realizar utilizando os critérios de aceitação criados e a descrição do trabalho. Quando todos os elementos possuem conhecimento do trabalho, é necessário medir, através de votação, o esforço associado. Caso não seja uma votação unânime, cada elemento está no seu direito de expor o seu ponto de vista a toda a equipa, de forma a que percebam a pontuação atribuída. Depois é realizada uma nova votação para definir o esforço.

  - O sistema de pontuação do esforço associado ao trabalho de uma US é baseado numa sequência de *Fibonacci*. Iniciando em 0 ou 1, o termo seguinte é obtido através da soma dos dois números anteriores, isto é, 0,1,1(0+1),2(1+1),3(2+1),etc. Então a sequência seria a seguinte: 0,1,**1,2,3,5,8,13,21,34,55,...**, sendo que quanto maior o valor maior o esforço associado.

- Retrospetiva (*Retrospective*) - realizada no último dia, é onde é realizada uma retrospetiva de como correu toda a *sprint*. É um espaço de abertura total para com a equipa, no que toca aos problemas relacionados com o trabalho, bem como problemas pessoais, ou seja, literalmente qualquer assunto, positivo ou negativo. O intuito desta conversa/discussão é tentar melhorar o desempenho de equipa que possa estar a ser bloqueado por algum fator e que em conversa de equipa possa surgir a solução. No fim é feito um resumo para sintetizar os pontos principais e ver, ao certo, o que há a melhorar e quais foram as coisas valorizadas.

Numa metodologia típica de *SCRUM* existe sempre uma demonstração para o PO, mas na metodologia utilizada pela empresa em estudo, o PO tem um papel ativo na conclusão das US's uma vez que todas as US's são apenas consideradas concluídas após validação do PO.

Todo o processo de qualidade começa pela definição das US's que irão refletir todo o trabalho a realizar. Sendo assim, o processo de qualidade começa por definir, para cada US, a contextualização do seu aparecimento e estipular os seus objetivos. De seguida, são criados os critérios de aceitação derivados dos objetivos a atingir, e por fim, é refinada toda a US e estimado o custo de execução da mesma.

Depois da definição da US, o processo pode ser dividido consoante o seu tipo, uma vez que para cada uma é exigido uma abordagem diferente da equipa e também do analista de qualidade. Sendo assim, poderia ser dividido como ilustrado na figura 3.3.

Figura 3.3: Tipos de *User Stories*

As US's de produto e de defeitos, são executadas durante o processo usando por base um sistema de *pipelines* com vários estágios e validações intermédias, até poder ser enviada para um ambiente produtivo. Uma ilustração dos estágios que constituem a *pipeline* usada é indicada na figura 3.4.

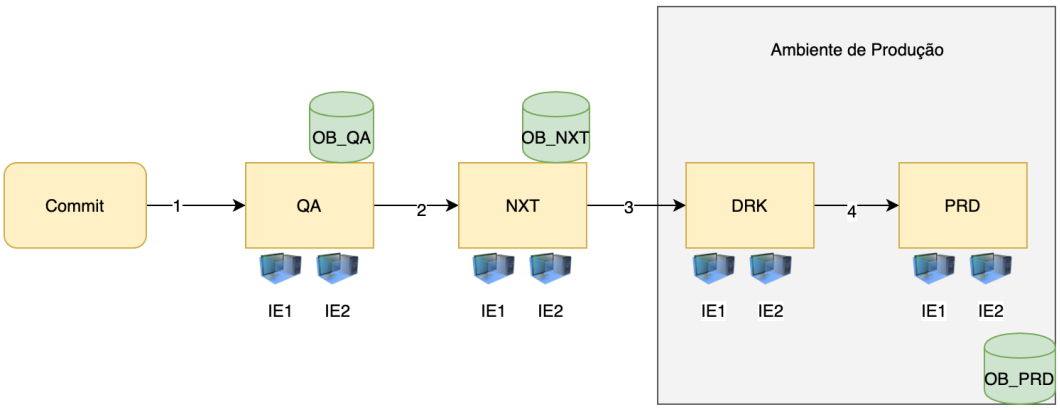


Figura 3.4: *Pipeline* de execução do processo

3.1.1 *User Story* de Investigação

Começando por falar do tipo de US de investigação, é uma fase mais pequena mas que não deixa de exigir esforço. Tem lugar quando existem questões que necessitam de uma investigação pormenorizada e demorada sobre algum ponto específico.

Ocorre quando existem questões incertas e que podem estar relacionadas com questões de produção ou questões de novas US's que estejam em *backlog*. Pode também ocorrer quando um problema levanta questões que possam vir a afetar o trabalho em desenvolvimento ou trabalho presente no *backlog*.

Após toda a investigação realizada sobre o problema em questão, é necessário proceder com criação de páginas de documentação, para que fique documentado o problema que ocorreu e o que levou ao seu aparecimento. Isto permite que haja mais informação sobre o problema e como vir a resolvê-lo futuramente.

Por fim, é revista a documentação, tanto pela pessoa responsável pela investigação como pelos restantes elementos da equipa, e disponibilizada numa plataforma onde contém toda a documentação existente no meio interno da empresa.

3.1.2 *User Story* de Produto

Este tipo de US inicia-se pelo desenvolvimento propriamente dito, através da implementação dos critérios de aceitação, seguida da criação dos testes unitários referentes ao código desenvolvido. É utilizada uma ferramenta de análise de cobertura de código, para que seja possível assegurar que os testes unitários criados atinjam uma percentagem de cobertura de 80%. São implementados também testes funcionais nesta fase, que são já automatizados e integrados, para serem executados na *pipeline*.

Após esta primeira fase, é aberta uma revisão de código, para que todos os membros de equipa olhem para o código implementado. Isto ajuda, para além de que toda a equipa retém o mínimo conhecimento da implementação realizada, a que melhores maneiras de implementação surjam, ou que possam ser discutidos pontos ainda confusos e que necessitem de mais atenção, e permite também que algum esquecimento que tenha ocorrido possa ser notado.

Ao fim de pelo menos 3 elementos da equipa terem completado a revisão de código, cabe ao desenvolvedor responsável pela implementação de proceder com a leitura de todos os comentários que foram feitos na revisão, e implementar sugestões relevantes.

Usando um sistema de controlo de versões, é feita uma verificação ao código, onde são adicionadas as novas propriedades à *build chef* que será usado para proceder com as entregas nos estágios da *pipeline* ilustrada na figura 3.4.

Após a fase de desenvolvimento terminar, é necessário juntar (transição 1) o código realizado para o seu ramo (*branch*) principal. Antes de se prosseguir na *pipeline*, é necessário garantir que o *branch master* consegue realizar a *build* corretamente. Isto é referente ao estágio de QA, e onde são executados os testes funcionais, anteriormente criados. Caso esteja a verde a *build*, é sinal de que não existe nenhum teste a falhar, podendo prosseguir com a *pipeline*.

É então realizada a segunda transição (2) que se encontra na figura 3.4. É iniciada a implantação (*deploy*) desta nova versão no estágio de NXT, onde existe um maior fluxo de dados, de possível manipulação diretamente na base de dados e que permite averiguar o funcionamento desta versão com uma maior quantidade de dados.

São realizadas validações manuais após o *deploy* para NXT, para que estejam asseguradas as funcionalidades implementadas nestes primeiros estágios, permitindo assim que erros sejam detetados atempadamente e que não se prolonguem na *pipeline* até produção.

Prossegue-se com a atualização da documentação existente, e são criadas, ou atualizadas caso já existam, as notas de entrega para que se possa de seguida proceder com o *deploy* para o estágio seguinte.

Caso sejam efetuados desenvolvimentos em bibliotecas externas, deve ser criada uma nova US para que se proceda com a atualização de todos os componentes que usem esta mesma biblioteca.

Sendo que os estágios seguintes (DRK e PRD) na *pipeline* se encontram em ambiente de produção é necessário um maior cuidado. Uma vez que o ambiente de produção utiliza dados reais, é uma parte do processo um pouco mais delicada sendo que é onde pode criar problemas reais e prejudiciais à empresa. Estão de seguida especificados em mais detalhes os passos desta fase mais delicada do processo

Estando então criadas/atualizadas as notas de entrega, procede-se com o *deploy* para o estágio de DRK. Este estágio é uma cópia de PRD e no qual se ganha confiança sobre o código desenvolvido pois consegue analisar-se o comportamento que é tido com dados reais.

Após o *deploy* estar completo, o analista de qualidade, ou outro membro por este designado, tem a função de proceder com validações manuais, designadas de *sanity tests*. Estas validações são realizadas través do uso de gráficos que refletem os fluxos de dados a ocorrer, da verificação nas interfaces da informação disponibilizada ser correta e ainda através de ferramentas internas desenvolvidas para auxiliar nestas validações.

Em seguida, são criadas ou atualizadas as notas de entrega mas agora para o estágio de PRD. Antes de se proceder com a *release* propriamente dita, é necessário informar todos as partes interessadas e todos os clientes dos serviços a ser atualizadas, através dos canais da plataforma *Slack* usada e de *e-mail*, de que se vai proceder com a mesma.

É então criado o *ticket* da *release*, e inicializada para o estágio de produção. Assim que termine, é necessário proceder outra vez com os *sanity tests*, tal como aconteceu em DRK, e assim que tiver terminado, é encerrado o *ticket*, e por *e-mail* e em todos os canais de *Slack* destinados ao componente, dá-se a conhecer que o processo de *release* está terminado.

Por fim, pode dar-se por completa ou fechada a US.

3.1.3 *User Story* de Defeito

Uma US de defeito surge quando foi encontrado algum defeito em algum componente e que a sua correção exija um esforço significativo.

Então, esta US segue o mesmo processo que uma US de produto, como descrito anteriormente, e são ainda criados testes de regressão, de forma a ajudar a perceber em que ponto é que o defeito ocorre e de que tipo de ações ele deriva.

É finalizado o processo com a alteração do estado do defeito para fechado, aquando da finalização do *deploy* para produção.

3.1.4 *User Story* Operacional

Um outro tipo de Us aqui apresentado, é designada de US operacional, e que são referentes a testes de desempenho efetuados sobre os componentes especificados.

Ocorre em alturas específicas, uma vez que não é algo que ocorra com frequência, apenas quando é desejado ou necessário. Pode derivar de um pedido do PO, podendo este pedido derivar de uma *release* para uma atualização de grande dimensão, por exemplo, com muita quantidade de informação e de funcionalidades, ou pode derivar de já existir um intervalo de tempo considerável desde a última execução de testes de desempenho.

Para que estes testes de desempenho ocorram é necessário, primeiramente, definir os cenários de teste, e criar um *ticket* para se poder processar com os testes. Antes da execução, existe a preparação do ambiente em que estes testes vão correr, e quando termine, os testes são executados.

Os resultados obtidos são documentados, para que se mantenha um registo do estado dos componentes aquando das realizações deste tipo de testes. Terminando o processo para testes de desempenho, com uma revisão da documentação e disponibilização da mesma na plataforma de documentos usada pela empresa.

3.2 Limitações Identificadas

Após levantamento do processo de garantia de qualidade que é usado na empresa em caso de estudo, procedeu-se com a análise do mesmo, para que pudessem ser sugeridas melhorias ao processo. Verificou-se que o processo utilizado é bastante desenvolvido e sofisticado, uma vez que detinham pontos de verificação de qualidade muito bem situados e que permitia avançar no processo de qualidade tendo certezas sobre as fases anteriores.

Com algum esforço, conseguiu-se identificar duas oportunidades de melhoria, e que necessitaram de uma análise mais específica, para saber que vantagens e dificuldades iriam trazer a adoção das sugestões de melhorias.

Uma possível melhoria identificada foi a automação das validações realizadas em ambiente de produção (explicada em mais detalhe em **??**), advém da realização de testes completamente manuais sobre os estágios do ambiente mencionado e que servem de validação, sendo morosos em termos de tempo e de encontrar exemplos relevantes para essas mesmas validações.

A outra sugestão de melhoria incide sobre as baterias de testes criadas, de modo a ganhar confiança sobre as mesmas ou proceder com melhorias. Para saber a qualidade dos testes criados, são usados testes de mutação, e a análise e aplicação da sua integração no processo de qualidade, encontra-se detalhada no capítulo 4.

Antes de avançar com a análise da sugestão de adoção de testes de mutação, procurou-se saber dentro da empresa equipas que já utilizassem este tipo de testes nos seus processos de qualidade, existindo uma única equipa de *frontend* que se encontrava nesta situação. A equipa que se encontra em caso de estudo de toda esta dissertação, encontra-se envolvida na área de *backend*, logo as

conclusões tiradas pela outra equipa servem de apoio mas não de sustento às conclusões já tiradas. Foram feitas 3 perguntas, obtendo-se respostas como:

1. 'O que os levou a aplicar testes de mutação?'
    - A resposta obtida foi que a equipa realizou uma pesquisa que mostrou que a ferramenta de cobertura de código que estavam a usar não lhes trazia confiança no que toca à realização de testes de lógica do código.
2. 'Que vantagens os testes de mutação trouxe?'
    - Foi respondido que ajudou a detetar e remover pedaços de código redundante no projeto em questão. Sendo que era aconselhável para *backend*, mas para projetos de *frontend*, nem sempre trazia grandes vantagens para o custo associado.
3. 'Quais as dificuldades sentidas?'
    - Responderam que a ferramenta aplicada tem um estado de maturidade baixo e que como os seus projetos eram apenas de *frontend*, os resultados obtidos face ao trabalho que era necessário realizar não compensava.

Das respostas obtidas por esta equipa, e nos pontos que se podem considerar mútuos, é que de facto, a maturidade das ferramentas ainda é baixa e que tem sido um entrave para a utilização de testes de mutação de um modo eficaz e vantajoso.

Capítulo 4

Estudo Experimental para Adoção de Testes de Mutação

O objetivo é apresentar a análise experimental efetuada para determinar de que forma os testes de mutação podem ser introduzidos na empresa para se verificar o nível de benefícios que podem ser obtidos, e as dificuldades que se podem encontrar na introdução destes tipos de testes em projetos de grande dimensão.

Sendo esta uma das melhorias possíveis ao processo de teste da empresa em estudo, podemos ter em conta os seus potenciais benefícios:

- Os testes de mutação são apontados na literatura como a melhor técnica de avaliação da qualidade dos testes, pois o tamanho do código de testes e cobertura não são, por si, bons indicadores de qualidade dos testes;
- Testes de mutação ajudam a melhorar a bateria de testes (mutantes sobreviventes).

e as suas potenciais dificuldades:

- Tempo de execução elevado;
- Tempo de análise de mutantes sobreviventes é alto;
- Dificuldade em usar de mutantes representativos de defeitos reais;
- Falta de maturidade das ferramentas.

4.1 Seleção de Ferramentas

Para um melhor entendimento da escolha da ferramenta, é necessário em primeiro lugar, entender a escolha do projeto para o estudo da adoção de testes de mutação. O projeto em questão

é escrito na linguagem *Scala*, e é um projeto já antigo e que tem impacto sobre o trabalho desenvolvido por algumas equipas. É um projeto que está envolvido na transferência de dados entre componentes de *streaming* e que assume um papel importante e sobre o qual não havia uma grande confiança sobre a bateria de testes criada.

Com isto, surgiu, logo de início, a sugestão da aplicação de testes de mutação a este projeto. Sendo que o projeto, como um todo, é demasiado grande, e não é aconselhável, para as ferramentas existentes para a linguagem em que o projeto se encontra, a aplicação delas, escolheu-se três módulos de menor dimensão para a aplicação de testes de mutação. O único senão à partida seria que os testes de mutação deveriam ser aplicados a um projeto que ainda se encontrasse em desenvolvimento, e não a um projeto já desenvolvido, tendo mesmo assim prosseguido com a aplicação deste tipo de testes, de modo a conseguir provar e trazer melhorias para o projeto.

A primeira etapa realizada foi a procura de ferramentas de testes de mutação, que permitissem a sua aplicação. Uma vez que o projeto está escrito em linguagem *Scala* foi necessário procurar ferramentas para esta linguagem, tendo sido encontradas duas ferramentas para o propósito.

A primeira foi o *Scalamu* que é uma ferramenta de testes de mutação para *Scala* mas que se encontra sem qualquer atividade e desenvolvimento há cerca de dois anos. Realizou-se a tentativa da aplicação da ferramenta para averiguar que resultados daí resultariam, mas a tentativa falhou pois não se conseguiu compilar o código e a documentação existente sobre a ferramenta em nada ajudou para o entendimento dos erros e resolução dos mesmos.

Uma vez que o *Scalamu* se revelou inutilizável, procedeu-se com a procura de outras ferramentas de testes de mutação. Foi então encontrada a ferramenta *Stryker*.

Figura 4.1: *Stryker* Logótipo

O *Stryker* é uma ferramenta de testes de mutação para a linguagem *JavaScript* que tem um trabalho de desenvolvimento enorme e que fornece bons resultados para os seus utilizadores. Com o mesmo intuito, foram desenvolvidas duas vertentes desta ferramenta, uma para a linguagem *C#*, cujo nome é *Stryker.NET*, e outra para a linguagem *Scala*, que se designa de *Stryker4s*.

Será então usado o *Stryker4s* como ferramenta de testes de mutação para o projeto em questão, escrito em *Scala*. Apresentando agora a ferramenta em si, ela segue o processo de mutação mencionado anteriormente em 2.4. Apesar de se encontrar numa fase de desenvolvimento inicial, o *Stryker4s* já se consegue aplicar a projetos reais, sendo encontradas questões de desempenho e problemas de compilação de projetos de elevada dimensão, algo que já se encontra documentado pelos seus desenvolvedores na documentação da ferramenta.

Retomando o exemplo das figuras 2.4 e 2.5, o que o *Stryker4s* faz depois de injetar os mutantes no código, é correr os testes unitários sobre o código original e de seguida sobre o código mutante. O modo como esta ferramenta sabe onde e quando aplicar mutantes, é através da compilação do código e de identificar no *bytecode* com "*ACTIVATE_MUTATION*"as secções do código onde vai injetar mutantes. A forma como ele as secções de código são identificadas está demonstrada na figura 4.2.

```
sys.env.get("ACTIVE_MUTATION") match {
  case Some("378") =>
    require(providedPayloadDeserializer == null, sys.env.get("ACTIVE_MUTATION") match {
      case Some("379") => ""
      case _ => "payload deserializer is mandatory"
    })
  case _ =>
    require(providedPayloadDeserializer != null, sys.env.get("ACTIVE_MUTATION") match {
      case Some("379") => ""
      case _ => "payload deserializer is mandatory"
    })
}
```

Figura 4.2: Ilustração de mutações injetadas a nível de *bytecode*

Ao longo do processo é possível, através da consola, e no final em formato de relatório *html*, saber que mutantes estão a ser, ou foram, aplicados, algo como mostra a figura 4.3.

```
Mutant killed: /yourPath/yourFile.js: line 10:27
Mutator: BinaryOperator
-                 return user.age >= 18;
+                 return user.age > 18;

Mutant survived: /yourPath/yourFile.js: line 10:27
Mutator: RemoveConditionals
-                 return user.age >= 18;
+                 return true;
```

Figura 4.3: Exemplo relatório na consola

O *Stryker4s* apresenta algumas funcionalidades de que se destaca:

- Velocidade - usa processamento paralelo na análise de código e execução de testes;
- Corredor de testes (*test-runner*) agnóstico - escolha do *test-runner* preferido;
- Multilingua - como mencionado antes, pode ser aplicado a várias linguagens de programação;
- Controlo dos mutantes - possível ter controlo sobre mais de 30 mutantes ou possível escrita dos próprios mutantes;
- Relatórios inteligentes - para permitir saber quais os mutantes sobreviventes e melhorar os testes;

- Código aberto (*Open Source*) - permite que quem o queira ajude ao desenvolvimento da ferramenta;

As classes de mutantes que são suportados para cada linguagem, que esta ferramenta suporta, estão apresentados na figura 4.4.

Mutator	Stryker	Stryker.NET	Stryker4s
Arithmetic Operator	✅	✅	❌
Array Declaration	✅	❌	❌
Assignment Expression	❌	✅	n/a
Block Statement	✅	❌	❌
Boolean Literal	✅	✅	✅
Checked Statement	n/a	✅	n/a
Conditional Expression	✅	✅	✅
Equality Operator	✅	✅	✅
Logical Operator	✅	✅	✅
Method Expression	❌	✅	✅
String Literal	✅	✅	✅
Unary Operator	✅	✅	❌
Update Operator	✅	✅	n/a

Figura 4.4: Mutantes suportados

A primeira coluna é referente à linguagem de *JavaScript*, a coluna do meio para *C#* e, por fim, a coluna da direita referente à linguagem *Scala*. É também possível verificar que para cada linguagem a ferramenta apresenta um nome diferente, sendo que para *JavaScript*, a ferramenta tem o nome de *Stryker*, para *C#* o nome de *Stryker.NET* e para *Scala*, apresenta o nome de *Stryker4s*.

Para cada projeto no qual vai ser aplicada esta ferramenta, é possível criar um ficheiro de configuração com configurações próprias e não as que estão definidas por predefinição. Por predefinição, a ferramenta aplica todos os tipos de mutantes a todos os ficheiros que estejam no diretório do projeto. São criados, também por predefinição, relatórios em consola e *html*, usa limites para classificar a pontuação das mutações, onde alta=80, baixa=60 e partida=0. Mas, configurando um ficheiro próprio, é possível escolher os ficheiro onde aplicar os testes de mutação, escolher os tipos de relatório a criar (consola, *html* ou *json*), excluir tipos de mutantes a não aplicar, definir os próprios limites e ainda escolher o *test-runner* desejado.

Para ser usado um ficheiro de configuração, basta criar um ficheiro com o nome "*stryker4s.conf*"no diretório do projeto, e a ferramenta quando for corrida, irá ler o ficheiro e aplicar as configurações escritas nele. Um exemplo está presente na figura 4.5.

```
stryker4s {
  # Your configuration here
}
```

Figura 4.5: Ficheiro de configuração - *Stryker4s*

Uma curiosidade encontrada durante a análise desta ferramenta, é pelo facto de se encontrar em constante desenvolvimento, e conseguir perceber, na prática, as melhorias que vem a sofrer ao longo do tempo e a que funcionalidades são dadas prioridades no desenvolvimento. É de notar que na primeira aplicação do *Stryker4s* eram apenas aplicadas mutações do tipo *string* e nas últimas aplicações, existe já todas as classes de mutantes mostrados na figura 4.4.

De seguida serão realizadas duas experiências, uma com configurações padrão e outra com configurações escolhidas que melhor se enquadravam no projeto em questão.

4.2 Experimentação com Configurações Padrão

Numa primeira tentativa de aplicação da ferramenta, foram aplicadas todas as classes de mutantes existentes (visíveis na figura 4.4) para que pudesse ser feita uma análise exaustiva ao número de mutantes criados e quais deles se mostravam relevantes. O *Stryker4s* foi aplicado a 3 módulos de um projeto com uma dimensão grande. Nesta primeira iteração da sua aplicação, os resultados foram extremamente baixos para todos eles quando executados com as configurações predefinidas.

No primeiro módulo, "Common", em que foi testada a ferramenta, foram identificados 17 de 33 ficheiros para serem mutados, tendo sido gerados um total de 95 mutantes, tal como mostra a figura 4.6.

```
[MACC02TG36QGTFL:common liram$ mvn stryker4s:run
[INFO] Scanning for projects...
[INFO]
[INFO] -----------< com.ppb.platform.stream:stream-protocol-common >-----------
[INFO] Building stream-protocol-common 2.4.22_0.10-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- stryker4s-maven-plugin:0.4.0:run (default-cli) @ stream-protocol-common ---
[WARNING] Could not find config file /Users/liram/Documents/BLIP/stream-protocol/common/stryker4s.conf
[WARNING] Using default config instead...
[INFO] Found 17 of 33 file(s) to be mutated.
[INFO] 95 Mutant(s) generated.
[INFO] Starting initial test run...
[INFO] Initial test run succeeded! Testing mutants...
[INFO] Starting test-run 1...
[INFO] Finished mutation run 1/95 (1%)
```

(...)

```
[ERROR] Mutation score dangerously low!
[ERROR] Mutation score: 43.16%
[INFO] Written HTML report to /Users/liram/Documents/BLIP/stream-protocol/common
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  16:41 min
[INFO] Finished at: 2019-06-09T00:42:09+01:00
[INFO] ------------------------------------------------------------------------
```

File / Directory	Mutation score		# Killed	# Survived	# Timeout	# No coverage	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
All files	43.16%	43.16	41	54	0	0	0	0	**41**	**54**	**95**
container	42.50%	42.50	34	46	0	0	0	0	**34**	**46**	**80**
model	27.27%	27.27	3	8	0	0	0	0	**3**	**8**	**11**
serialization	100.00%	100.00	4	0	0	0	0	0	**4**	**0**	**4**

Figura 4.6: Módulo *Common*

Considerando que, para este módulo, foram corridos 56 testes unitários (presente na tabela 4.1), o tempo de execução foi considerado razoável, completando os testes de mutação em aproximadamente 17 minutos e com uma taxa de mutação extremamente baixa.

De seguida foi aplicada a um módulo designado "*Metadata*", onde foram corridos 25 testes unitários (presente na tabela 4.1).

```
MACC02TG36QGTFL:metadata liram$ mvn stryker4s:run
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Detecting the operating system and CPU architecture
[INFO] ------------------------------------------------------------------------
[INFO] os.detected.name: osx
[INFO] os.detected.arch: x86_64
[INFO] os.detected.classifier: osx-x86_64
[INFO]
[INFO] ----------< com.ppb.platform.stream:stream-protocol-metadata >----------
[INFO] Building stream-protocol-metadata 2.4.22_0.10-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- stryker4s-maven-plugin:0.4.0:run (default-cli) @ stream-protocol-metadata ---
[WARNING] Could not find config file /Users/liram/Documents/BLIP/stream-protocol/metadata/stryker4s.conf
[WARNING] Using default config instead...
[INFO] Found 6 of 11 file(s) to be mutated.
[INFO] 90 Mutant(s) generated.
[INFO] Starting initial test run...
[INFO] Initial test run succeeded! Testing mutants...
[INFO] Starting test-run 1...
[INFO] Finished mutation run 1/90 (1%)
```

(...)

```
[ERROR] Mutation score dangerously low!
[ERROR] Mutation score: 30.0%
[INFO] Written HTML report to /Users/liram/Documents/BLIP/stream-protocol/metadata/
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  27:05 min
[INFO] Finished at: 2019-06-09T02:07:53+01:00
[INFO] ------------------------------------------------------------------------
```

File / Directory	Mutation score	# Killed	# Survived	# Timeout	# No coverage	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
All files	30.00	27	63	0	0	0	0	27	63	90
kafka/KafkaMetadataRetriever.scala	38.10	8	13	0	0	0	0	8	13	21
serialization	0.00	0	2	0	0	0	0	0	2	2
service/ZookeeperMetadataService.scala	27.50	11	29	0	0	0	0	11	29	40
MetadataCoordinatorManager.scala	29.41	5	12	0	0	0	0	5	12	17
MetadataCoordinatorService.scala	30.00	3	7	0	0	0	0	3	7	10

Figura 4.7: Módulo *Metadata*

Como mostra a figura 4.7, este é um módulo mais pequeno, onde de 11 ficheiros apenas 6 deles foram identificados para mutação. No entanto, foram gerados tantos mutantes como no módulo anterior, cerca de 27 minutos de tempo de execução, resultando numa taxa de mutação mais baixa, 30%.

Para se poder ter mais um termo de garantia dos testes de mutação, foi aplicada a ferramenta a um terceiro módulo, "*Consumer*", que apresenta um tamanho maior, onde foram mutados 35 ficheiros, gerando um total de 407 mutantes.

```
MACC02TG36QGTFL:consumer liram$ mvn stryker4s:run
[INFO] Scanning for projects...
[INFO]
[INFO] ----------< com.ppb.platform.stream:stream-protocol-consumer >----------
[INFO] Building stream-protocol-consumer 2.4.22_0.10-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- stryker4s-maven-plugin:0.4.0:run (default-cli) @ stream-protocol-consumer ---
[WARNING] Could not find config file /Users/liram/Documents/BLIP/stream-protocol/consumer/stryker4s.conf
[WARNING] Using default config instead...
[INFO] Found 35 of 57 file(s) to be mutated.
[INFO] 407 Mutant(s) generated.
[INFO] Starting initial test run...
[INFO] Initial test run succeeded! Testing mutants...
[INFO] Starting test-run 1...
[INFO] Finished mutation run 1/407 (0%)
```

(...)

```
[ERROR] Mutation score dangerously low!
[ERROR] Mutation score: 35.87%
[INFO] Written HTML report to /Users/liram/Documents/BLIP/stream-protocol/
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  05:12 h
[INFO] Finished at: 2019-06-09T07:32:30+01:00
[INFO] ------------------------------------------------------------------------
```

File / Directory	Mutation score	# Killed	# Survived	# Timeout	# No coverage	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
All files	35.87	146	261	0	0	0	0	**146**	**261**	**407**
core	35.91	93	166	0	0	0	0	**93**	**166**	**259**
management	36.28	41	72	0	0	0	0	**41**	**72**	**113**
modules	12.50	1	7	0	0	0	0	**1**	**7**	**8**
ConsumerContainer.scala	40.74	11	16	0	0	0	0	**11**	**16**	**27**

Figura 4.8: Módulo *Consumer*

Neste módulo foram corridos 178 testes unitários (presente na tabela 4.1), pelo que à partida, se esperava um tempo de execução elevadíssimo. Terminando com uma taxa de mutação de aproximadamente 36%, os testes de mutação foram executados num total de 5 horas e 12 minutos, como mostra na figura 4.8, tempo considerado inaceitável.

4.3 Análise de Resultados com Configurações Padrão

Uma vez que os resultados obtidos foram extremamente baixos, procedeu-se com uma análise dos mutantes aplicados, para que fosse possível entender a relevância dos mesmos e a qualidade da bateria de testes. No contexto do projeto, e após bastante análise, verificou-se que todo o tipo de *strings* contidas no código desenvolvido, ou era para variáveis/constantes ou para *debug*, sejam funções de "*log*"ou "*require*". Exemplos de mutantes que eram aplicados estão presentes na figura 4.9.

Figura 4.9: Exemplos de mutantes sobreviventes do tipo *String*

Na figura 4.9, conseguimos verificar duas situações:

- Verificamos que estão a ser aplicados mutantes do tipo *string* e que estão a sobreviver, mas podemos ver que na função *require* apenas a primeira condição é testada devidamente, e que apenas não existe teste que cubra a parte da *string*.
- Na função *log*, verificamos que a *string* não possui testes e daí estar o mutante aplicado a sobreviver.

```
logLevel.toUpperCase match {
  case 59"DEBUG" | 60"FINE" | 61"FINEST" => Logging.DebugLevel
  case 62"INFO" => Logging.InfoLevel
```

Figura 4.10: Exemplo de mutante morto do tipo *String*

Na figura 4.10 podemos ver que a *string* está a representar um estado de *debug*, e que existem testes para este cenário, uma vez que não é uma *string* apenas.

Ou seja, podemos concluir que existem apenas alguns testes unitários criados para cobrir *strings* mas que foram desenvolvidos para locais específicos e que a equipa, dona do projeto, considera que não faz sentido testar esta classe, e que existir erros aqui, não traria problemas aos utilizadores finais.

Para que seja mais explícito, e de forma a suportar a decisão da equipa em usar as *strings* maioritariamente para *debug*, foram analisados todos os mutantes do tipo *string* dentro dos módulos, e construiram-se os seguintes gráficos:

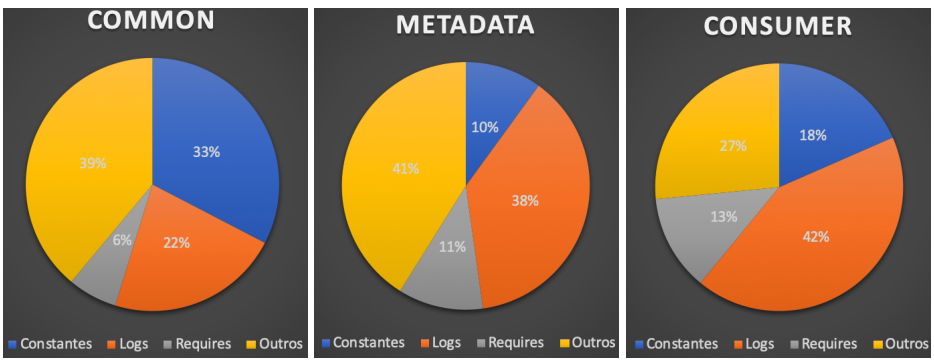


Figura 4.11: Gráfico Módulos Aplicados

Como se pode verificar nos gráficos presentes na figura 4.11, os outros tipos de mutantes que não *string*, representam sempre menos de 50% dos mutantes aplicados e, tentando perceber, que percentagem de mutantes do tipo *string* eram mortos ou sobreviventes, usando apenas o total de mutantes deste tipo, construiram-se os seguintes gráficos:

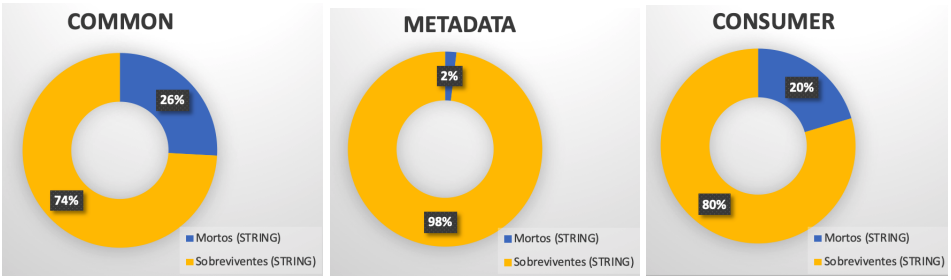


Figura 4.12: Percentagens Mutantes *STRING*

Analisando os gráficos da figura 4.12, pode concluir-se que de facto, dentro dos mutantes de *string*, para cada módulo, mais de 75% dos mutantes estão a sobreviver, pelo que, e devido ao facto de *string* apenas ser usado, maioritariamente, para *debug*, sentiu-se a necessidade de excluir esta mesma classe de mutantes. Foi então que se procedeu com uma nova iteração, onde mutantes do tipo *string* iriam ser excluídos, e cuja análise se pode encontrar na secção seguinte.

4.4 Experimentação com Configurações Adaptadas

A necessidade de excluir mutantes, levou a uma segunda iteração para tentar avaliar de novo a qualidade dos testes unitários, quando aplicadas apenas mutações com relevância. Com isso, foi criado um ficheiro de configuração, com as características desejadas e que pode ser visto na figura seguinte:

```
stryker4s {
  excluded-mutations: ["StringLiteral"],
  thresholds{ high=70, low=50, break=0 }
}
```

Figura 4.13: Novo ficheiro de configuração

Alterou-se então os limiares e excluiu-se mutantes do tipo *string literal*.

Após exclusão dos mutantes do tipo *string*, para o primeiro módulo, "Common", os resultados foram bastante superiores. Como é possível ver na figura 4.14, foram na mesma usados 17 ficheiros, gerados 95 mutantes, mas destes, 59 foram excluídos por serem do tipo *string*.

```
[INFO] --- stryker4s-maven-plugin:0.4.0:run (default-cli) @ stream-protocol-common ---
[INFO] Using stryker4s.conf in the current working directory
[INFO] Found 17 of 33 file(s) to be mutated.
[INFO] 95 Mutant(s) generated. Of which 59 Mutant(s) are excluded.
```

(...)

```
[WARNING] Mutation score: 69.44%
[INFO] Written HTML report to /Users/liram/Documents/BLIP/stream-protocol/commo
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  07:12 min
[INFO] Finished at: 2019-06-14T11:24:00+01:00
[INFO] ------------------------------------------------------------------------
```

File / Directory	Mutation score	# Killed	# Survived	# Timeout	# No coverage	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
All files	69.44% 69.44	25	11	0	0	0	0	25	11	36
container	64.29% 64.29	18	10	0	0	0	0	18	10	28
model	75.00% 75.00	3	1	0	0	0	0	3	1	4
serialization	100.00% 100.00	4	0	0	0	0	0	4	0	4

Figura 4.14: Módulo *Common*

Sendo o número de mutantes bastante menor, o tempo de execução foi também menor, relembrando que para este módulo são executados cerca de 56 testes unitários (presente na tabela 4.1), e terminou com uma taxa de 70%, o que se encontra demonstrado na seguinte figura.

Executando do mesmo modo, com as mesmas configurações, para o módulo "*Metadata*", também se obtiveram menos mutantes para serem executados.

```
[INFO] --- stryker4s-maven-plugin:0.4.0:run (default-cli) @ stream-protocol-metadata ---
[INFO] Using stryker4s.conf in the current working directory
[INFO] Found 6 of 11 file(s) to be mutated.
[INFO] 90 Mutant(s) generated. Of which 54 Mutant(s) are excluded.
[INFO] Starting initial test run...
```

(...)

```
[INFO] Mutation score: 72.22%
[INFO] Written HTML report to /Users/liram/Documents/BLIP/stream-protocol/metadat
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  11:11 min
[INFO] Finished at: 2019-06-14T11:40:07+01:00
[INFO] ------------------------------------------------------------------------
```

File / Directory	Mutation score	# Killed	# Survived	# Timeout	# No coverage	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
All files	72.22% 72.22	26	10	0	0	0	0	**26**	**10**	**36**
kafka/KafkaMetadataRetriever.scala	70.00% 70.00	7	3	0	0	0	0	**7**	**3**	**10**
service/ZookeeperMetadataService.scala	68.75% 68.75	11	5	0	0	0	0	**11**	**5**	**16**
MetadataCoordinatorManager.scala	100.00% 100.00	5	0	0	0	0	0	**5**	**0**	**5**
MetadataCoordinatorService.scala	60.00% 60.00	3	2	0	0	0	0	**3**	**2**	**5**

Figura 4.15: Módulo *Metadata*

Com apenas 36 mutantes para serem corridos, e cerca de 25 testes unitários (presente na tabela 4.1), terminou a sua execução em 11 minutos e 11 segundos, como está visível na figura 4.15. Este módulo foi aquele no qual se verificou um maior aumento na taxa de mutação, cerca de 72%.

Procedendo com o módulo de maior dimensão e com um maior número de testes unitários, 178 testes (presente na tabela 4.1), executou-se a ferramenta com a nova configuração, figura 4.13. Aqui foram ignorados cerca de 300 mutantes nesta iteração, o que reduziu imenso o tempo de execução.

```
[INFO] --- stryker4s-maven-plugin:0.4.0:run (default-cli) @ stream-protocol-consumer ---
[INFO] Using stryker4s.conf in the current working directory
[INFO] Found 35 of 57 file(s) to be mutated.
[INFO] 407 Mutant(s) generated. Of which 299 Mutant(s) are excluded.
```

(...)

```
[INFO] Mutation score: 79.63%
[INFO] Written HTML report to /Users/liram/Documents/BLIP/stream-protocol/consume
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:23 h
[INFO] Finished at: 2019-06-14T13:06:51+01:00
[INFO] ------------------------------------------------------------------------
```

File / Directory	Mutation score		# Killed	# Survived	# Timeout	# No coverage	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
All files	79.63%	79.63	86	22	0	0	0	0	86	22	108
core	78.21%	78.21	61	17	0	0	0	0	61	17	78
management	94.44%	94.44	17	1	0	0	0	0	17	1	18
modules/dependencies	0.00%	0.00	0	4	0	0	0	0	0	4	4
ConsumerContainer.scala	100.00%	100.00	8	0	0	0	0	0	8	0	8

Figura 4.16: Módulo *Consumer*

Ao fim de 1 hora e 23 minutos a execução terminou, da qual resultou a taxa mais alta dos três módulos usados para a análise.

4.5 Análise de Resultados com Configuração Adaptada

Realizando uma comparação entre as duas iterações, pode-se verificar que para todos os módulos analisados, os resultados foram melhores na segunda iteração.

A primeira iteração foi utilizada para perceber o estado da ferramenta e quais as necessidades que o projeto exigia. Então, a primeira iteração foi corrida sob as características padrão, onde se:

- Aplica mutação a todos os ficheiros no formato da linguagem *Scala*;
- Analisa todos os ficheiros dentro do diretório identificado para o projeto;
- Toma como base do diretório a pasta principal do projeto onde vai ler os ficheiros;
- Cria um relatório em consola e *html* no final da execução;
- Aplica todas as classes de mutantes aos ficheiros no formato desejado;
- Utiliza como limiar inferior da taxa de mutação alta 80%, baixo 60% e erro 0%.

Para os resultados obtidos na primeira iteração, foi realizada uma análise dos mesmos e perceber o que estaria na origem de taxas de mutação tão reduzidas. Foram revistos então os mutantes

que foram aplicados e os limiares para o qual o *Stryker4s* estava configurado. Todas as classes de mutantes pareciam fazer sentido para o propósito do projeto em questão, exceto a classe *string literal*, sendo que este tipo apenas é usado para *debug*, como dito na apresentação dos resultados das configurações padrão.

Os limiares que estavam predefinidos foram considerados altos e foi então criado um ficheiro de configuração, figura 4.13, onde foram alterados os limiares de pontuação e foi excluída a classe de mutantes *string literal*.

Com a segunda iteração, foi possível verificar aquilo que foi concluído anteriormente. Com o final da segunda iteração houve, em todos os módulos, melhorias significativas na taxa da mutação. Usando os limiares da nova configuração, dois dos módulos (figuras 4.15 e 4.16) entraram no estado de sucesso e um (figura 4.14) ficou a menos de 1% de chegar a este estado, o que comprova que os mutantes de *string* estavam a ser utilizados em maioria para *debug* e que não tinham testes unitários desenvolvidos para eles propositadamente.

Os resultados obtidos permitiram trazer as garantias procuradas quanto à questão dos mutantes relevantes, mas falta ainda analisar as vantagens que trás a um projeto quando se reúnem todas as características desejadas. Com isso, é de realçar a importância existente na fase de análise dos resultados obtidos pelos testes de mutação. De facto, os mutantes necessitam ser revistos e analisados, pois podem existir casos por testar ou podem estar a ser aplicados mutantes sem relevância para algum caso específico. Por vezes, a taxa pode ser baixa mas não possuir um significado relevante pois os mutantes aplicados podem não ter qualquer sentido, ou pode a ferramenta não estar a conseguir filtrar certos casos já testados por testes unitários. Ou seja, não se deve confiar cegamente na taxa, pois tanto pode ter uma taxa baixa mas existir um grande número de mutantes sem sentido, ou ter uma taxa alta e estar um único caso crítico por testar, que caso se ignore, possa a vir, num futuro, trazer problemas no software ou produto desenvolvido.

De modo a tentar encontrar um caso em que se possa provar que de facto, olhar só para a taxa não é fiável, temos o exemplo da figura 4.17.

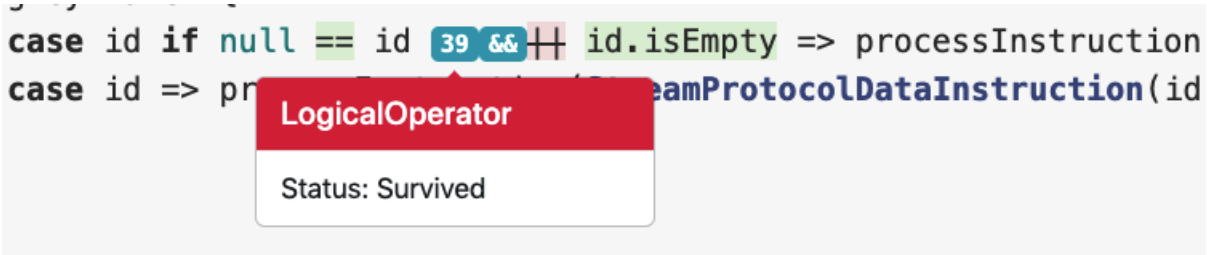


Figura 4.17: Exemplo de mutante em dúvida

No exemplo apresentado na figura 4.17, é visível um caso em que foi aplicada uma mutação que não possui nexo, tanto do nível do projeto como de lógica simples de programação. É dito que a mutação não possui nexo, uma vez que no contexto da lógica do código, a variável *id* não pode ser nula e vazia ao mesmo tempo, e daí o mutante ter sobrevivido. Existe uma verificação para ver se uma variável é nula ou está vazia (operador lógico '||'), sendo que é aplicada um mutante

do tipo de operador lógico, que vai verificar se a variável é vazia e nula (operador lógico '&&'), algo que não é possível de acontecer. Aqui temos um caso que falhou no mutante mas que a sua aplicação poderia ser ignorada por uma análise estática do código.

Depois de uma exaustiva análise, não se conseguiu encontrar situações em que existisse de facto, falta de testar um cenário crítico para o projeto. Foram encontrados alguns casos em que efetivamente se poderia melhorar a bateria de testes, não sendo algo crítico, mas que poderia trazer erros futuros. Para exemplo de caso em que foi possível melhorar a bateria de testes, é aquele que se encontra apresentado na figura seguinte:

```
require(evictionThreashold 72> 0, "evictionThreshold must be

val isEvictable = currentTime - consumerData.metadata.lastUpdateTime 75> evictionThreashold
log.debug("operation=isEvictable, evictable={}, currentTime={}, consumerDataLastUpdateTime={}",
```

Figura 4.18: Mutantes sobreviventes

Com uma taxa, da classe e do módulo, de mutação, antes da adição dos testes, que permitisse matar estes mutantes, de 79.63%.

File / Directory **Mutation score**

DefaultTimeBasedEvictor.scala 66.67% 66.67

File / Directory	Mutation score	# Killed	# Survived	# Timeout	# No coverage	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
All files	79.63% 79.63	86	22	0	0	0	0	86	22	108
core	78.21% 78.21	61	17	0	0	0	0	61	17	78
management	94.44% 94.44	17	1	0	0	0	0	17	1	18
modules/dependencies	0.00% 0.00	0	4	0	0	0	0	0	4	4
ConsumerContainer.scala	100.00% 100.00	8	0	0	0	0	0	8	0	8

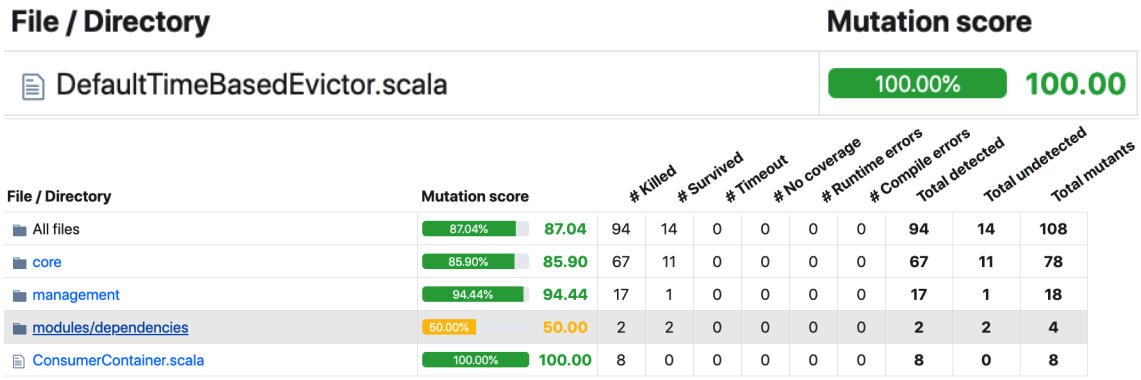
Figura 4.19: Taxa de mutação inicial - antes da adição dos testes

Para demonstrar que a sua melhoria foi deveras implementada, foram adicionados dois testes ao módulo *Consumer*, para cobrir estas duas situações. Foi corrida novamente a ferramenta, obtendo-se a morte dos mutantes anteriormente sobreviventes (figura 4.20), e consequentemente uma taxa de mutação melhor, cerca de 8% maior, como se pode ver na figura 4.21.

```
require(evictionThreashold > 0, "evictionThreshold must be

val isEvictable = currentTime - consumerData.metadata.lastUpdateTime > evictionThreashold
log.debug("operation=isEvictable, evictable={}, currentTime={}, consumerDataLastUpdateTime={}",
```

Figura 4.20: Mutantes mortos



File / Directory	Mutation score		# Killed	# Survived	# Timeout	# No coverage	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
All files	87.04%	87.04	94	14	0	0	0	0	94	14	108
core	85.90%	85.90	67	11	0	0	0	0	67	11	78
management	94.44%	94.44	17	1	0	0	0	0	17	1	18
modules/dependencies	50.00%	50.00	2	2	0	0	0	0	2	2	4
ConsumerContainer.scala	100.00%	100.00	8	0	0	0	0	0	8	0	8

Figura 4.21: Taxas de mutação final - depois da adição dos testes

Numa tentativa de resumir certos aspetos sobre os módulos aos quais foi aplicada a ferramenta, tendo por base a segunda iteração (v1) e a correção de dois mutantes sobreviventes (v2), foi construida a seguinte tabela:

	Common	Metadata	Consumer
Tamanho do código sob teste (LOC)	771	501	3245
Tamanho do código de teste (LOC)	1525	344	4617
Nº de métodos de teste	56 -	25 -	178(v1) 180(v2)
Cobertura dos testes	91.73% -	88.50% -	86.89%(v1) 86.94%(v2)
Taxa de mutação	69.44% -	72.22% -	79.63%(v1) 87.04%(v2)
Número de mutantes	36	36	108
Tempo execução dos testes de mutação	7min12seg	11min11seg	1h23min

Tabela 4.1: Tabela de propriedades dos módulos testados

Explicando melhor a tabela apresentada, para cada módulo, foram retiradas informações sobre o tamanho de código, tanto de código a ser testado como código de implementação de testes, a cobertura que a bateria de testes apresenta, a taxa de mutação retirada da execução da ferramenta de testes de mutação, quantidade de testes unitários e quantidade de mutantes gerados pela ferramenta e ainda os tempos de execução praticados pela aplicação da ferramenta de mutação.

No módulo *Consumer*, para as propriedades cobertura dos testes, taxa de mutação e número de testes unitários, aparecem dois valores (v1 e v2), pois o primeiro (v1) é sobre a segunda iteração, e o segundo (v2) é sobre a segunda iteração com a correção de dois mutantes que estavam sobreviventes.

O que podemos retirar sobre estes dados é que de facto os testes de mutação, quando bem aplicados, podem trazer grandes vantagens para os projetos nos quais estão a ser aplicados. Por exemplo, para o módulo *Consumer* foram identificados duas situações de fronteiras não testadas, sendo que se procedeu com a implementação de dois testes para cobrir essas fronteiras. No que

toca à cobertura dos testes, esta aumentou menos de 1%, mas para a taxa de mutação aumentou cerca de 8%, ou seja, foi deveras relevante para a taxa de mutação e não para a cobertura que a bateria de testes possui. Mais casos foram identificados como possíveis melhorias, em todos os módulos, mas para questões de relevância e demonstração, achou-se que um teste de fronteira é algo essencial e que deve sempre estar testado e coberto, daí apenas se encontrarem estes dois casos de fronteiras melhoradas.

Podemos ainda retirar da tabela que a cobertura nem sempre é representativa no que se refere à testabilidade da lógica do código, isto é, ter maior cobertura num módulo do que noutro, nem sempre significa que no módulo com maior cobertura, estejam de facto a ser testadas mais cenários. Por exemplo, o módulo *Common* é o módulo com uma cobertura mais elevada dos três usados nesta experiência, mas se olharmos para a tabela 4.1, vemos que no que toca à taxa de mutação, é o que possui a menor taxa.

Verificou-se que a quantidade de linhas de código, tanto do código sob teste como do código dos testes, não significa qualidade dos testes, uma vez que, podem estar a ser escritos testes que estejam a testar coisas iguais ou semelhantes e que apesar de cobrir partes do código, não significa que o esteja a testar realmente.

Analisando ainda os tempos de execução e a dimensão dos módulos, isto é, linhas de código sob teste ou de teste, para o número de linhas existente, são praticados tempos de execução elevadíssimos, pois qualquer projeto atinge este número de linhas, e a nível empresarial, este número aumenta consideravelmente.

4.6 Lições Aprendidas e Recomendações

De um modo resumido, conseguimos demonstrar vantagens na aplicação de testes de mutação, mas também se encontraram obstáculos à sua adoção.

Pela análise das taxas, podemos verificar que de facto trouxe vantagens para o projeto, pois conseguimos:

- Demonstrar que os testes de mutação fornecem informação útil sobre a qualidade da bateria de testes implementada;
- Demonstrar que o número de linhas e a cobertura de código não provam e não dão garantias de que os testes têm a qualidade necessária;
- Demonstrar que os testes de mutação podem ajudar a melhorar a bateria de testes e a identificar problemas que não apareceram até ao momento mas que num futuro poderiam aparecer.

Por outro lado, analisando os tempos de execução verificamos que existem muitas questões a colocar relacionadas com este tópico. O tempo de execução associado a correr esta ferramenta aumenta consoante o número de mutantes criados, com o número de linhas de código, e ainda com o número de testes unitários implementados. Isto é visível no módulo *Consumer* onde na primeira

iteração com 407 mutantes e 178 métodos de teste, levou cerca de 5 horas, e na versão final da segunda iteração, levou 1 hora e 23 minutos para 108 mutantes e 180 métodos de teste.

Por fim, uma das maiores adversidades encontradas, derivou da maturidade da ferramenta utilizada para a aplicação de testes de mutação. Apesar de até se ter conseguido obter resultados positivos, existem melhorias que se poderiam fazer na ferramenta para que o seu desempenho e os seus resultados obtidos, sejam melhores e mais relevantes.

Com os resultados obtidos e com as conclusões tiradas, procurou-se obter mais respostas sobre esta ferramenta junto dos seus responsáveis. Falou-se com Hugo Van Rijswijk, atual maior contribuidor para o *Stryker4s* e com ele obtivemos respostas das quais se consegue perceber o rumo que a ferramenta esta a tomar. A ferramenta deriva também de uma dissertação na área da garantia de qualidade e que para ele, deverá num futuro fornecer, com um tempo de execução razoável, uma visão precisa sobre a qualidade da bateria de testes de um projeto. A versão utilizada é já uma versão com algum desenvolvimento mas que precisa de evoluir, estando planeado pelos desenvolvedores da ferramenta as seguintes funcionalidades:

- Melhorias de desempenho integrando mais diretamente *test-runners* - o que eles acreditam que traga grandes mudanças neste tópico;
- Usar análise de cobertura para executar os testes de mutação sobre código que seja de facto coberto pelos testes unitários e que se obtenham melhores e mais precisas taxas de mutação;
- Retroceder/ignorar mutações que causem erros de compilação, para que sejam aplicadas apenas as que consigam passar pela compilação;
- Executar uma análise do tipo de mutante que irá ser aplicado para se garantir que a mutação é possível antes de efetivamente a aplicar.

Com a utilização da ferramenta, ao longo do processo, surgiram 2 melhorias possíveis a implementar.

A primeira melhoria que surgiu foi o facto de identificar realmente as zonas que estariam a ser cobertas. Não foi reportada aos seus responsáveis, uma vez que, já se encontra nos planos futuros do desenvolvimento da ferramenta No código do projeto em questão, existem certos comentários, como demonstrado na figura 4.22 que significam que entre eles, não existe cobertura de código.

```
case t: Throwable =>
  // $COVERAGE-OFF$
  log.error("The actor system could not be shutdown: {}", List(t.getMessage, t))
// $COVERAGE-ON$
```

Figura 4.22: Exemplo de código sem cobertura

Esta sugestão de melhoria será contemplada com a implementação de uma das funcionalidades já pensadas para a ferramenta, que passa por usar a análise de cobertura de código para que os testes de mutação atinjam apenas código coberto e testado pela bateria de testes.

Um problema encontrado durante a execução da primeira iteração com as configurações padrão, está relacionado com a variável '*sys*' que é usada na compilação do código para mutação, figura 4.2, e com uma variável presente num ficheiro de código, demonstrada na seguinte figura:

```
sys.addShutdownHook {
  Option(actorSystem).foreach { sys =>
    log.info("Shutdown hook called: Shutting down the actor system")
    shutdownActorSystem(sys) {}
  }
}
```

Figura 4.23: Variável em observação

O uso desta variável no código, lança, quando usado em modo *debug*, um erro na consola, falhando a compilação, como demonstrado na figura 4.24.

```
[DEBUG] [ERROR]       log.info(sys.env.get("ACTIVE_MUTATION") match {
[DEBUG] [ERROR]                          ^
[DEBUG] [ERROR] one error found
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  12.585 s
[INFO] Finished at: 2019-06-17T12:22:20+01:00
[INFO] ------------------------------------------------------------------------
```

Figura 4.24: Erro encontrado e compilação falhada

Para este erro encontrado, foi criado um tópico na página de erros do projeto da ferramenta *Stryker4s*, como é provado na figura 4.25.

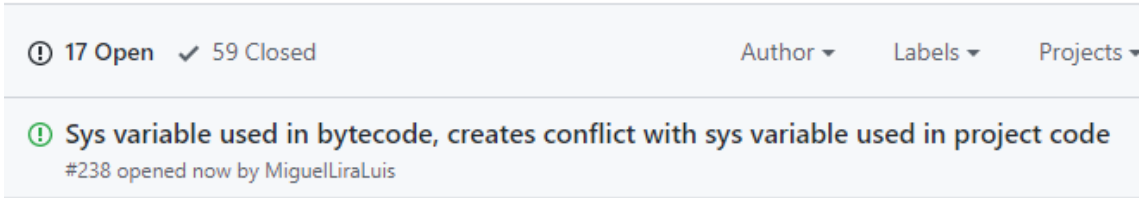


Figura 4.25: Erro reportado na página de erros da ferramenta *Stryker4s*

A solução encontrada para resolver este problema, foi a alteração do nome da variável no código, tal como mostra a figura 4.26. Com isto foi sugere-se uma melhoria na compilação do código mutante de modo a não existir incompatibilidades entre variáveis de código e de compilação. Sugeriu-se ainda a informação, junto da documentação da ferramenta, da possibilidade, de atualmente existir incompatibilidades de variáveis.

```
sys.addShutdownHook {
  Option(actorSystem).foreach { sys2 =>
    log.info("Shutdown hook called: Shutting down the actor system")
    shutdownActorSystem(sys2) {}
  }
}
```

Figura 4.26: Correção do erro de compilação

Ocorreu ainda o aparecimento de um mutante duvidoso, figura 4.17, que poderia ter sido não aplicado se existisse uma análise estática do código. Isto porque, existe uma variável que ou é nula ou é vazia, não podendo ocorrer, no contexto do código escrito, as duas coisas em simultâneo.

Para o *Stryker4s* foram estas as melhorias sugeridas, e tendo em conta as melhorias para a empresa em caso de estudo, foram sugeridas a integração de testes de mutação no seu processo de qualidade. É sugerido que exista muita ponderação na utilização de testes de mutação, uma vez que, apesar de existir vantagens na identificação de certos cenários ou casos pequenos de métodos de testes, existe duas grandes adversidades, tempo de execução elevado e maturidade das ferramentas ser baixa, que a nível empresarial vêm travar o uso deste sistema de testes de mutação.

Imaginando que a ferramenta possui uma maturidade considerável e que permita que os tempos de execução sejam aceitáveis para a sua utilização em meio empresarial, é sugerida a integração de testes de mutação, no processo de qualidade e na *pipeline* de ambientes pela qual todos os projetos de cada equipa e empresa desenvolvidos passam. Uma sugestão de integração na *pipeline* utilizada pela empresa está presente na figura 4.27.

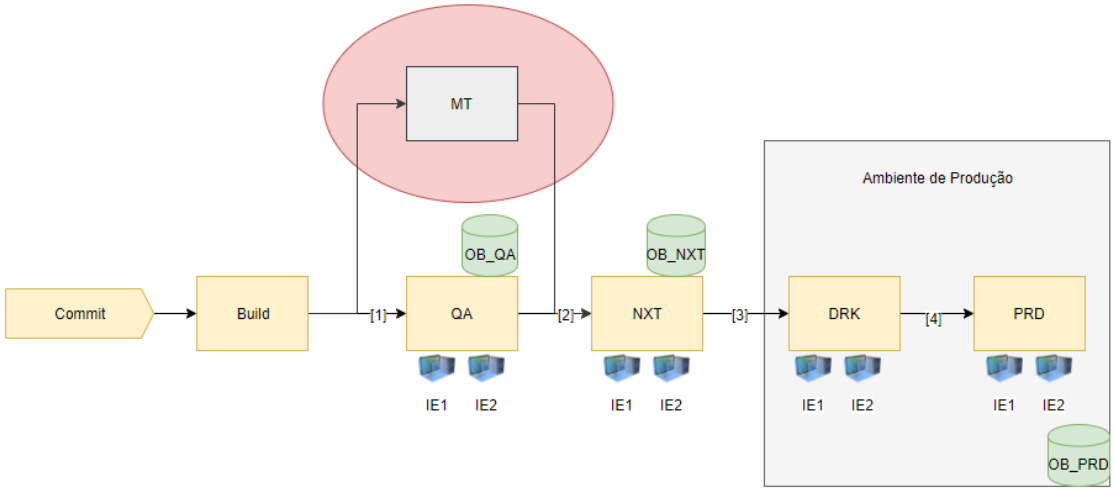


Figura 4.27: Integração de Testes de Mutação

A figura 4.27 representa o sistema de *pipelines* utilizado pela equipa em caso de estudo, onde é possível verificar que o estágio MT é o estágio de testes de mutação a integrar na pipeline, correspondendo à implementação da melhoria sugerida. Para compreensão da melhoria sugerida, de nada influência saber o que é IE1 e IE2 presente na figura 4.27, mas a título de curiosidade,

cada estágio da pipeline é composto por dois centros de dados (*Data Center*), em que um está ativo e o outro passivo para que aquando do ativo falhe, o passivo passe rapidamente para ativo, não comprometendo assim todas as plataformas de utilizadores.

É então sugerido que os testes de mutação sejam integrados em paralelo com o estágio de QA, utilizando configurações já anteriormente mencionadas na 4.13, não sendo a *pipeline* partida quando as taxas de mutação não correspondam ao estado de sucesso. É apenas sugerido que seja corrido em paralelo com este estágio e que lance um alerta, consoante o estado que resultante com a taxa obtida, para que os desenvolvedores estejam cientes e que possam proceder com uma análise crítica de modo a avaliar a relevância dos mutantes aplicados.

Capítulo 5

Conclusões e Trabalho Futuro

O atual capítulo tem como principal objetivo, resumir todo o conhecimento aprendido, as conclusões obtidas, contribuições para a conclusão da dissertação e ainda os trabalhos futuros para realizar.

Nas conclusões é pretendido verificar se o levantamento do processo de qualidade foi realizado com sucesso, verificar ainda se a análise deste mesmo processo foi realizada de um modo significativo, verificar que da análise realizada sugestões de melhorias foram feitas, e por fim, que foi realizada também a análise da viabilidade de uma eventual adoção ou integração das melhorias sugeridas.

5.1 Conclusões

A presente dissertação, realizada em meio empresarial, incidiu sobre uma área crítica e que tem ganho ênfase nos dias de hoje e no qual as empresas vêm a apostar firmemente. Desde a apresentação do processo de qualidade praticado na empresa e alvo de estudo, até às melhorias ao mesmo, benefícios e dificuldades, tentou-se alcançar os objetivos estabelecidos inicialmente.

Tendo sempre em mente estes objetivos, foi analisado o processo de qualidade, tendo sido verificado que este processo usado pela equipa, e pela empresa, está bastante desenvolvido e sofisticado. Mesmo assim, foram encontradas duas limitações importantes.

Após o surgimento de várias ideias, concluiu-se que existiam duas melhorias possíveis de analisar e vir num futuro a ser aplicado ao processo. Devido ao custo temporal que as validações manuais realizadas após a entrega para produção, foi então analisada a possibilidade de automatizar estas validações, passando pela criação de uma bateria de testes para ser executada após *release* para produção. Teriam ainda que ser realizadas validações manuais de análise dos gráficos referentes ao ambiente de produção, validações às interfaces dos utilizadores para verificar que a informação disponibilizada é a correta, e ainda a realização de testes manuais para algum cenário impossível ou difícil de cobrir com a bateria de testes. Conclui-se que de facto, traria poupança

de tempo a criação de uma bateria de testes automáticos, uma vez que testes e novos cenários podem ser adicionados à bateria de testes, uma vez que poderia ser criada ou atualizada aquando do desenvolvimento das USs e que traria bastantes reduções no tempo de validação no ambiente de produção.

A outra melhoria para o processo de qualidade veio no seguimento de saber a qualidade das baterias de testes que estão a ser usadas para cobrir a lógica do código desenvolvida.

Esta melhoria iria incidir sobre o ambiente de QA, demonstrado na figura da *pipeline* do processo (demonstrado na figura 4.27), e que permitiriam averiguar a qualidade da bateria de testes. Surgiu então os testes de mutação e começou-se por procurar uma ferramenta que pudesse auxiliar este processo. *Stryker4s* foi, apesar da sua baixa maturidade, a ferramenta escolhida, e foram analisados, através de uma experiência prática, fatores como cobertura de código, taxa de mutação, número de mutantes, tempo de execução, entre outros. A escolhida desta ferramenta baseou-se no facto de que para a linguagem *Scala* apenas existirem duas ferramentas disponíveis, e uma delas não possuir qualquer desenvolvimento há cerca de 2 anos. Com os resultados obtidos, foram encontrados exemplos de aplicação no qual a aplicação não teve sentido sobre o código em questão e outros com correta aplicação de mutantes, tendo resultado em várias possíveis melhorias, sendo que foi usada, para exemplo, apenas uma para melhoria da bateria de testes. Uma vez que o tempo de execução era um fator determinante para a análise da viabilidade de integração de testes de mutação no processo, apesar das melhorias trazidas por este tipo de testes, para o tamanho de código existente o tempo de execução obtido é impraticável em meio empresarial, com tempos superiores a 1 hora de execução de testes para módulos com 3245 LOC.

Em resumo, para as melhorias sugeridas, a automatização dos testes de validação pós entrega em ambiente de produção é aconselhável, tendo apenas um custo inicial na criação da bateria de testes automatizados e melhorias após criação e integração destes testes na *pipeline*. Já a integração na *pipeline* de testes de mutação não é viável enquanto a ferramenta não for mais madura e oferecer melhores desempenhos a nível de tempos de execução, precisão e relevância nos mutantes aplicados. A utilização da ferramenta *Stryker4s* para os testes de mutação permitiu, a nível interno, detetar falhas na bateria de testes pois foram encontradas fronteiras no código por testar. E a nível externo, permitiu contribuir para a documentação de erros que impedem o funcionamento correto da ferramenta.

5.2 Trabalho Futuro

A nível da empresa, o presente trabalho permitiu levantar e documentar o processo de garantia de qualidade que praticam, algo que ainda não possuíam, permitiu ainda a análise da viabilidade de oportunidades que foram sugeridas durante a análise do seu processo de garantia de qualidade, sendo que existe uma forte possibilidade de automatização das validações manuais em ambiente produtivo. A implementação da outra melhoria, não existe a probabilidade da sua integração, uma vez que a maturidade que a ferramenta que permite a aplicação de testes de mutação é baixa e os tempos de execução não são viáveis.

Na empresa então um trabalho futuro a ser realizado, passaria portanto por automatizar as validações, com a criação de uma bateria de testes com os cenários desejados e possíveis de incluir na bateria. Integrar a bateria num estágio final da *release* e proceder com a correção de eventuais erros que acontecessem na execução da bateria de testes, ou então finalizar o processo com verificação dos gráficos e das interfaces finais.

Algo que poderia ter sido efetuado e que não foi possível neste trabalho e durante todo o período de desenvolvimento deste trabalho é a realizar a automatização das validações e analisar os seus resultados, de modo a possuir provas reais de que a redução do tempo de validação manual reduziria imenso. E a nível da integração dos testes de mutação, algo que infelizmente não foi realizado por consumo exagerado na execução dos testes de mutação, era a integração na *pipeline* e analisar todo o mecanismo de *release* até ambiente final de produção, de modo a perceber se para diferentes projetos se obtinham estados em que a *pipeline* procede com sucesso ou então lança um alerta relativo à baixa taxa de mutação obtida.

Externamente à empresa, trabalhos futuros a realizar, seria, em termos práticos, contribuir para o desenvolvimento da ferramenta analisada, para que uma versão melhorada e mais fiável pudesse ser fornecida para utilização.

Um ponto que surgiu, mas que seria de elevada complexidade, seria analisar o mutante sobrevivente que foi aplicado e tentar sugerir várias opções de testes unitários que cobrissem esse caso. Isto poderia ser feito dentro da ferramenta ficando integrado nela, ou desenvolver uma ferramenta externa, que pudesse ser usada com uma ferramenta de mutação, sendo que seria necessário conseguir entender a lógica da função aplicada para conseguir fornecer sugestões válidas.

Referências

[Bak17] Kapil Bakshi. Microservices-Based Software Architecture and Approaches. *2017 IEEE Aerospace Conference*, pages 1–8, 2017. doi:10.1109/AERO.2017.7943959.

[Bat18] Graham Bath. Certified Tester Foundation Level Specialist Syllabus Performance Testing American Software Testing Qualifications Board. 2018.

[dCSMS16] André de Camargo, Ivan Salvadori, Ronaldo dos Santos Mello e Frank Siqueira. An architecture to automate performance tests on microservices. *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services - iiWAS '16*, pages 422–429, 2016. URL: `http://dl.acm.org/citation.cfm?doid=3011141.3011179`, doi:10.1145/3011141.3011179.

[DDS14] Adam Debbiche, Mikael Dien e Richard Berntsson Svensson. Integration : A Case Study. pages 17–18, 2014.

[GO11] Daniel Gomes e Joel Oliveira. Glossário Standard de termos usados em Testes de Software Produzido pelo " Glossary Working Party " Contributos para a versão Portuguesa. 1:1–75, 2011.

[HSF+19] Farah Hariri, August Shi, Vimuth Fernando, Suleman Mahmood e Darko Marinov. Comparing Mutation Testing at the Levels of Source Code and Compiler Intermediate Representation. *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 114–124, 2019. doi:10.1109/ICST.2019.00021.

[HSS15] Itti Hooda, Research Scholar e Rajender Singh Chhillar. Software Test Process, Testing Types and Techniques. *International Journal of Computer Applications*, 111(13):975–8887, 2015.

[Int11] International Software Testing Qualifications Board. Programa de Certificação de Testador (Tester) de Nível Foundation. 2011.

[JMH11] Yue Jia, Student Member e Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011. doi:10.1109/TSE.2010.62.

[LPK+] Thomas Laurent, Mike Papadakis, Marinos Kintis, Christopher Henard, Yves Le Traon e Anthony Ventresque. Assessing and Improving the Mutation Testing Practice of PIT. *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 430–435. doi:10.1109/ICST.2017.47.

[MP16] Prasad Mahajan e Bharati Pune. Different Types of Testing in Software Testing. pages 1661–1664, 2016.

REFERÊNCIAS

[MSB17] Torvald Mårtensson, Daniel Ståhl e Jan Bosch. Continuous Integration Impediments in Large-Scale Industry Projects. *Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017*, pages 169–178, 2017. doi:10.1109/ICSA.2017.11.

[Mut] Mutation testing. `https://www.tutorialspoint.com/software_testing_dictionary/mutation_testing.htm`. (Accessed on 06/28/2019).

[PJ16] Claus Pahl e Pooyan Jamshidi. Microservices : A Systematic Mapping Study. 1(Closer):137–146, 2016.

[PK18] Goran Petrovi e Bob Kurtz. An Industrial Application of Mutation Testing : Lessons , Challenges , and Research Directions. (Section V), 2018. doi:10.1109/ICSTW.2018.00027.

[PKZ+19] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon e Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019.

[SB14] Daniel Ståhl e Jan Bosch. Continuous integration flows. *Continuous software engineering*, 9783319112831:107–115, 2014. doi:10.1007/978-3-319-11283-1-9.

[SQN] International Software, Testing Qualifications e Copyright Notice. Standard Glossary of Terms used in Software Testing All Terms.

[Sta] Start killing mutants: Mutation test your code – itnext. `https://itnext.io/start-killing-mutants-mutation-test-your-code-3bea71df27f2`. (Accessed on 06/28/2019).

[Str] Stryker mutator. `https://stryker-mutator.io/`. (Accessed on 06/28/2019).

[Vmo] V-model-en - modelo v – wikipédia, a enciclopédia livre. `https://pt.wikipedia.org/wiki/Modelo_V#/media/Ficheiro:V-model-en.png`. (Accessed on 07/03/2019).

[Whaa] What is quality assurance(qa)? process, methods, examples. `https://www.guru99.com/all-about-quality-assurance.html`. (Accessed on 07/15/2019).

[Whab] What is software quality assurance (sqa): A guide for beginners. `https://www.softwaretestinghelp.com/software-quality-assurance/`. (Accessed on 07/15/2019).

[ZZH+18] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia e Lu Zhang. Predictive Mutation Testing. 14(8), 2018. doi:10.1109/TSE.2018.2809496.