Mestrado em Engenharia Informática
Estágio
Relatório Final

FeedZai Pulse: Processo de Garantia de Qualidade

Andrey Klimachev
audrey@student.dei.uc.pt

Orientadores:
Prof. Doutor Carlos Fonseca
Eng. Diogo Guerra

3 de Julho de 2013

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Resumo

A FeedZai é uma empresa de software especializada em processamento de dados em tempo real, sendo um spin-off tecnológico da Universidade de Coimbra. Em particular, a FeedZai desenvolve um produto chamado FeedZai Pulse que permite o tratamento de grandes volumes de dados em tempo real. Os objetivos deste estágio foram criar uma infraestrutura para testes automatizados de software e avaliar os testes de sistema existentes, e também melhorá-los. Os resultados foram a criação de um conjunto de testes funcionais e de testes de regressão usando o *Selenium* [1], a criação do suporte para a sua execução, a avaliação da qualidade dos testes unitários existentes por meio de criação dos testes de mutação, e também a melhoria da recolha das métricas da qualidade.

Acrónimos

OBOE	Off-By-One Error
RAM	Random Access Memory
ATM	Automatic Teller Machine
PIN	Personal Identification Number
API	Application Programming Interface
OO	Orientado a Objetos
KPI	Key Performance Indicator
SRB	Pulse Software Review Board
RC	Release Candidate
GA	General Acceptance
XML	Extensible Markup Language
JVM	Java Virtual Machine
POM	Project Object Model
CRUD	Create, read, update and delete
AJAX	Asynchronous JavaScript and XML

Palavras chave:
Qualidade de Software, Processos da Qualidade, Garantia da Qualidade

Agradecimentos

Gostaria de agradecer à minha família que me apoiou e me permitiu conseguir chegar a fase de conclusão do curso.

Gostaria de agradecer ao meu orientador da *FeedZai*, Engenheiro Diogo Guerra, pela orientação séria por toda a exigência para com o trabalho realizado na empresa, e pela ajuda que sempre me deu nos tempos mais, difíceis durante o estágio e fora do âmbito deste.

Gostaria de agradecer ao meu orientador da faculdade, Prof. Doutor Carlos Fonseca, por toda a sua ajuda na elaboração do relatório.

Gostaria de agradecer a *FeedZai* por me proporcionar a oportunidade, não só de realizar o estágio, mas também por me permitir fazê-lo na minha área preferida - Qualidade de Software.

Gostaria de agradecer aos meus colegas e amigos pela compreensão, ajuda e bons momentos passados.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 Âmbito

Este documento constitui o relatório final da unidade curricular de Dissertação/Estágio do curso de Mestrado em Engenharia Informática da Universidade de Coimbra. O estágio foi realizado na área de qualidade de software e teve a duração de um ano académico (de Setembro 2012 a Julho 2013) e decorreu na FeedZai, situada no Instituto Pedro Nunes, e cujo produto é designado *Pulse*.

1.2 Objetivos

A FeedZai S.A. é uma empresa de software especializada em processamento de dados em tempo real, sendo um spin-off tecnológico da Universidade de Coimbra. Em particular, a FeedZai desenvolve um produto chamado FeedZai Pulse que permite o tratamento de grandes volumes de dados em tempo real. A FeedZai possui um interesse forte em melhorar a sua plataforma de controlo de qualidade, em particular, formalizando os mecanismos de teste que são utilizados assim como as suas componentes, a fim de garantir a qualidade global do produto desenvolvido.

PulseViews [2] é uma interface web que permite monitorizar e gerir as aplicações *Pulse*. Inicialmente não existia nenhum conjunto de testes funcionais automáticos para esta interface. Um dos objetivos deste estágio foi criar uma infraestrutura para testes automatizados de software, em particular,

criar um sistema de testes automáticos a nível da interface com o utilizador, multi-browser, para efeitos de testes funcionais e testes de regressão.

Existia ainda um conjunto de testes de sistema, de validação dos motores de backend do *Pulse*, mas não havia quaisquer métricas para avaliar a sua qualidade. Pretendeu-se definir estas métricas e melhorar a qualidade dos testes existentes. Relativamente aos mecanismos de controlo de qualidade atuais, pretendeu-se analisar e melhorar os mesmos.

Em suma, os principais objetivos do estágio são:

- Montar uma infraestrutura de testes automáticos para o *PulseViews*.
- Melhorar os testes automáticos de sistema existentes.
- Melhorar o processo da qualidade.

Os testes automáticos para o *PulseViews* foram criados usando a ferramenta *Selenium* destinada à execução de testes funcionais e testes de regressão para páginas *web*. Na base das funcionalidades existentes e dos *bugs* encontrados foram criados 683 casos de testes. Para avaliar a qualidade dos testes unitários existentes foram criados testes de mutação usando a ferramenta *PIT*[3].

Em relação à melhoria do processo da qualidade, foi desenvolvido um suporte para a análise de código fonte na linguagem de programação *Scala*. Também foi desenvolvida uma solução para o cálculo de cobertura para esta linguagem de programação, usando a ferramenta *JaCoCo*[4]. O projeto da *FeedZai* foi reformulado de modo a permitir a junçaõ de todos os módulos em vez desete teren que ser analisados.

Para unir todas as modificações efetuadas no produto *Pulse* foi criado o serviço de relatórios de qualidade com o objetivo de demonstrar a evolução das métricas da qualidade do produto ao longo do seu desenvolvimento.

1.3 Estrutura do documento

Este documento está organizado de seguinte forma:

- Capítulo 2, Qualidade de Software. Apresenta uma visão global sobre a qualidade de software, bem como os diferentes tipos de defeitos de software, tipos de testes, processo da qualidade e métricas da qualidade.
- Capítulo 3, Garantia da Qualidade do Produto *Pulse*. Apresenta uma visão global do *Pulse*. Descreve a realização do ciclo de release, a me-

todologia usada no processo de desenvolvimento de software, o funcionamento do departamento da Garantia da Qualidade e as suas tarefas, as ferramentas usadas na *FeedZai*.

- Capítulo 4, Testes funcionais automáticos. Descreve em detalhe a arquitetura do sistema onde os testes funcionais e testes de regressão são executados, bem como a arquitetura dos testes em sí.
- Capítulo 5, Melhorias do Processo da Qualidade. Descreve em detalhe como foi feita a análise de qualidade dos testes unitários existentes, como a estrutura foi refeita do projeto *Pulse*, e porquê.
- Capítulo 6, Servidor da Qualidade. Descreve em detalhe os objetivos do serviço de relatórios de qualidade, bem como a sua implementação..
- Capítulo 7, Plano de estágio. Apresenta uma descrição do plano detalhado do primeiro e segundo semestres, e a avaliação do seu desempenho.
- Capítulo 8, Conclusões e Trabalho futuro. Resumo das lições aprendidas e tarefas futuras.

Capítulo 2

Qualidade de Software

Nesta secção serão abordados alguns conceitos da qualidade de software, nomeadamente, tipos de defeitos de software, definição de casos de teste, diferentes tipos de testes, processos de qualidade e, por último, métricas de qualidade.

2.1 Introdução

Definir qualidade de software é uma tarefa difícil, e muitas definições têm sido propostas. O Dicionário da Língua Portuguesa da Porto Editora define qualidade como "propriedade, atributo ou condição natural de uma pessoa ou coisa que a distingue das outras" [5]. Como atributo de um item, a qualidade refere-se a coisas que podem ser medidas, ou seja, comparadas com padrões conhecidos, tais como tamanho, cor, etc. É mais difícil classificar qualidade em software que em objetos físicos, pois este é uma entidade abstrata.

Ao avaliar um item baseando-se nas suas características mensuráveis, dois tipos de qualidade podem ser encontrados: qualidade de projeto e qualidade de conformidade [6]. Qualidade de projeto refere-se a características que gestores de projeto especificam para um item (performance, tolerância, etc.). Estas características são mais focadas na especificação e na conceção do sistema. Qualidade de conformidade é o nível ao qual as especificações do projeto são seguidas durante o processo de desenvolvimento, focando-se mais na implementação (verificação das funcionalidades consoante os requisitos, testes funcionais, testes de aceitação, etc).

Uma definição possível da qualidade de software é : "conformidade com os

requisitos funcionais e de desempenho explicitamente declarados, com os padrões de desenvolvimento claramente documentados e com as características implícitas que são esperadas de todo software profissionalmente desenvolvido" [6].

2.2 Defeitos de Software

Um defeito de software é um erro no código, e uma falha é o resultado do defeito. Por outras palavras, uma falha é um resultado incorreto ou inesperado que uma aplicação ou um sistema produzem. Os defeitos podem ser classificados do seguinte forma [6]:

- Erros aritméticos
  - Divisão por zero. Situações onde o denominador é 0.
  - Overflow aritmético. Casos em que a operação resulta num resultado superior ao que uma variável consegue guardar.
  - Precisão aritmética. Situações em que se perde precisão aritmética devido a arredondamentos.
- Erros lógicos
  - Ciclo ou recursão infinitas. São ciclos que não têm condição de paragem ou em que a condição de paragem nunca é satisfeita.
  - Off-By-One Error (OBOE). Erro de uma unidade da especificação de uma condição fronteira.
- Uso incorreto de semântica
  - Utilização de um operador errado, por exemplo, realização de atribuição em vez do teste de igualdade.
- Erros de recursos
  - "Null-pointer dereference" ocorre quando aplicação está à espera de um ponteiro válido mas este é nulo.
  - Uso de variáveis não inicializadas.
  - Violações do espaço de endereçamento (*Segmentation fault*). Ocorre quando uma aplicação tenta aceder (para leitura ou escrita) a um

endereço na memória RAM que está reservado para outra aplicação, ou não é utilizável.

  - "*Buffer overflow*", quando uma aplicação tenta armazenar dados para além do espaço reservado.

  - "Stack overflow" quando existem demasiadas chamadas a funções recursivas ou encadeadas.

- Erros de multi-threading

  - "Deadlock", acontece quando a tarefa A não pode continuar até que a tarefa B termine, ao mesmo tempo que a tarefa B não pode continuar a sua execução enquanto a tarefa A não termina.

  - "Condições de corrida", que ocorrem quando o resultado de uma execução depende da ordem de execução de uma ou mais *threads*.

- Erros de interface

  - Uso incorreto interfaces de módulos (*Application Programming Inteface*).

  - Implementação incorreta de protocolos.

- Erros de desempenho

  - Elevada complexidade do algoritmo.

  - Acesso aleatório à memória ou ao disco.

- Erros humanos

  - Diferença entra a especificação das funcionalidades e o funcionamento destas no produto atual.

  - Comentários incorretos ou desatualizados no código fonte.

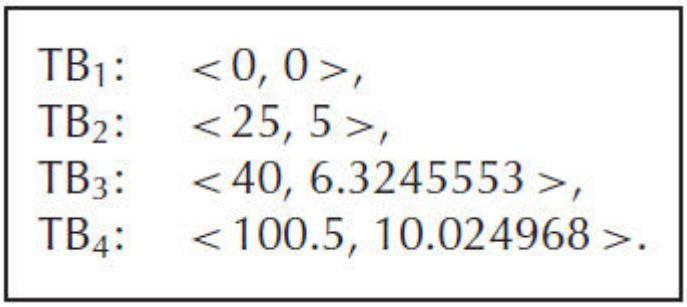

Figura 2.1: Caso de teste básico. [6]

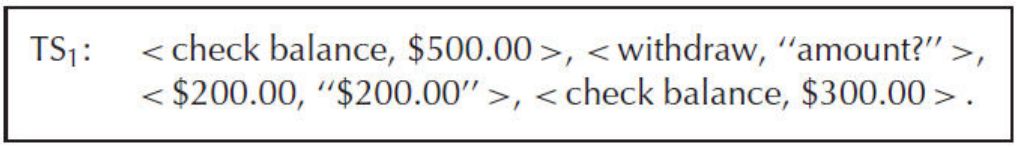

Figura 2.2: Caso de teste complexo. [6]

2.3 Definição de casos de teste

Um caso de teste é um conjunto de condições e variáveis, conjuntamente com o resultado esperado, que servem para verificar se um determinado sistema está a funcionar corretamente ou não. Existem dois tipos de casos de teste: básicos e complexos.

No caso de testes básicos a saída depende unicamente da entrada e a estrutura de entrada é muito simples. Na Figura 2.1 são mostrados alguns pares de entradas e saídas simples para o cálculo da raiz quadrada de números não negativos.

No caso de testes complexos, a saída depende não só da entrada mas também do estado do sistema e um caso de teste pode ser constituído por uma sequência de entradas e saídas. Na figura 2.2 é mostrado o processo de levantamento de dinheiro numa *automatic teller machine* (ATM). Assume-se que o utilizador já inseriu o cartão de multibanco, introduziu o seu *personal identification number* (PIN) e pretende levantar $200 tendo na conta bancária $500. O valor de saída no terceiro tuplo de $200 é o valor dispensado pela ATM. Depois de efetuar o levantamento, o utilizador verifica que o valor na conta é de $300.

2.4 Tipos de testes

Aqui são apresentados alguns tipos e técnicas de testes, inclusive os que são aplicados na FeedZai.

2.4.1 Testes unitários

Testes unitários referem-se aos testes de cada unidade do software [7], embora não haja consenso sobre a definição de uma unidade. Em programação orientada aos objetos, estas unidades são geralmente métodos ou classes. Sintaticamente, uma unidade de programa é uma peça de código chamada a partir de fora dessa peça. Uma unidade é testada antes de ser integrada com outras unidades/módulos. Existem duas razões para os testes unitários serem executados isoladamente dos outros módulos de uma aplicação: os erros encontrados na execução dos testes unitários podem ser atribuídos a uma determinada unidade e podem ser facilmente corrigidos, e durante este tipo de testes é desejável verificar se a unidade que está a ser testada produz um valor de saída correto. Os testes unitários têm alcance limitado, uma vez que verificam determinada funcionalidade independentemente das outras funcionalidades.

A execução de testes unitários refere-se a testes unitários dinâmicos. O ambiente para execução de testes é criado através da simulação do contexto da unidade a testar (ver figura 2.3). O **test driver** chama a unidade a ser testada e todas as unidades que são chamadas a partir desta chamam-se **stubs**. O *test driver* e os *stubs* são chamados **scaffolding**.

As funções de cada um dos elementos do *scaffolding* são as seguintes:

- *Test driver*: O *test driver* é uma aplicação que chama a unidade a ser testada. A unidade a testar executa-se com as entradas do *test driver* e, após o seu término, retorna o resultado ao *test driver*. Este, por sua vez, compara o valor retornado com o valor esperado.

- *Stubs*: Os *stubs* são pequenas aplicações chamadas pela unidade a testar. Um *stub* executa duas tarefas – demonstra o que realmente foi chamado pela unidade a testar, imprimindo uma mensagem, e retorna um valor à unidade a testar para esta conseguir continuar a sua execução. Muitas vezes, ao uso destas interfaces chama-se "mocking" ou "mock objects".

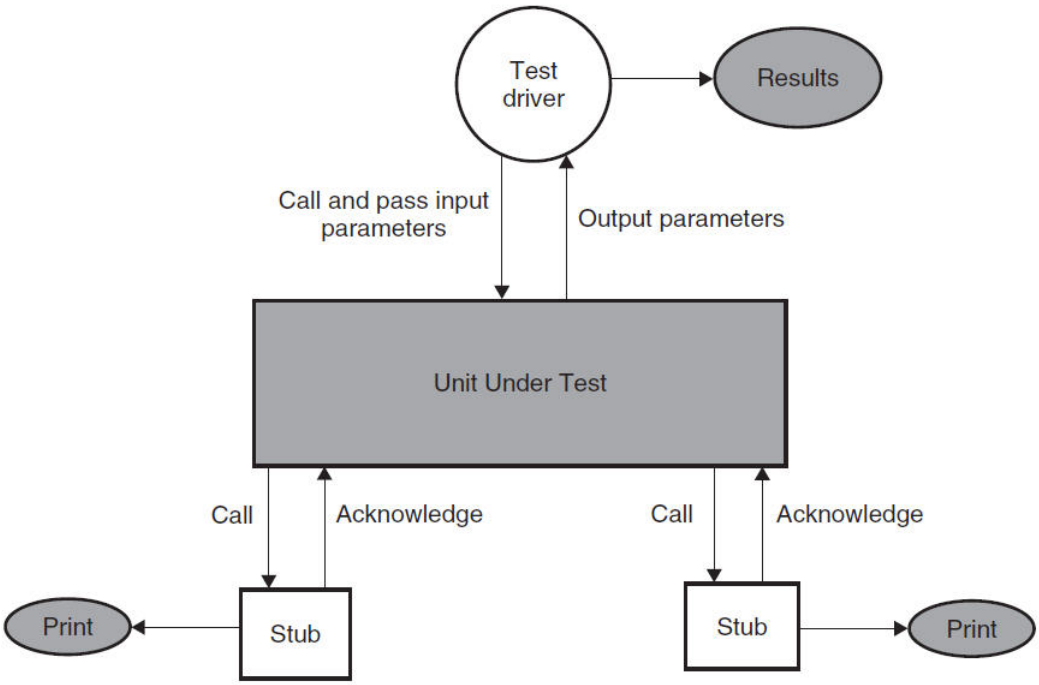

Figura 2.3: Ambiente para a execução dos testes unitários. [6]

```
public static void main(String args[])
{
    int r = 1;
    for(int i = 2; i < args.length(); i++)
    {
        if(Integer.parseInt(args[i]) > Integer.parseInt(args[r])) r = i;
    }
    System.out.println("Value of the rank is "+r);
}
```

Figura 2.4: Código fonte para testes de mutação.

As ferramentas de testes unitários mais comuns são o JUnit, TestNG, *ScalaTest*, JSTes.

Testes de mutação

O objetivo deste tipo de testes é expor e localizar falhas em casos de teste. Um teste de mutação é uma forma de medir a qualidade dos testes unitários. Os testes de mutação não são uma estratégia de teste, da aplicação, mas são usados para complementar as técnicas tradicionais de testes unitários.

Uma mutação de uma aplicação é uma modificação desta criada através da introdução de uma ou mais alterações sintaticamente corretas no código. Depois de introduzir as mutações, os testes unitários são executados. Os testes unitários que não conseguem detetar as alterações do código são considerados testes com defeitos.

Na Figura 2.4 é representado um exemplo do código fonte.

Assume-se que existem 3 casos de testes:

- Caso de teste 1. Valores de entrada: 1 2 3. Valor de saída é 3.
- Caso de teste 2. Valores de entrada: 1 2 1. Valor de saída é 2.
- Caso de teste 3. Valores de entrada: 3 1 2. Valor de saída é 1.

O código será modificado de seguinte forma:

- Mutação 1. Altera-se a linha *6* para *if (i > Integer.parseInt(argv[r])) r = i;*
- Mutação 2. Altera-se a linha *6* para *if (Integer.parseInt(argv[i]) >= Integer.parseInt(argv[r])) r = i;*

- Mutação 3. Altera-se a linha *6* para *if (Integer.parseInt(argv[r]) > Integer.parseInt(argv[r])) r = i;*

Após a execução dos testes são obtidos os seguintes resultados:

- Mutação 1 é detetada pelo caso de teste 2.
- Mutação 2 não é detetada.
- Mutação 3 é detetada pelos casos de teste 1 e 2.

2.4.2 Testes de integração

Através dos testes unitários os módulos são testados isoladamente. A próxima etapa é juntar diferentes módulos para construir um sistema completo. A construção de um sistema estável a partir de diferentes módulos envolve muitos testes. O caminho a partir das componentes testadas para a construção de um sistema contém duas fases de testes importantes, os testes de integração [6] e os testes de sistema. O objetivo principal dos testes de integração é montar um sistema razoavelmente estável no ambiente de desenvolvimento de tal forma que o sistema integrado possa suportar o rigor de um teste de sistema no ambiente real. Os testes de integração incluem duas abordagens, *white-box* e *black-box* [6]. Os testes black-box ignoram os mecanismos internos do sistema e concentram-se na análise dos valores de saída gerados por este. O software tester sabe o valor da entrada enviado para a black-box e observa o resultado da execução. Os testes white-box utilizam a informação sobre a estrutura do sistema para o testar. Estes contemplam os mecanismos internos do sistema e os módulos [6].

Um dos objetivos dos testes de integração consiste em, após a junção de diferentes módulos de software num sistema, testar esse sistema completo. Para efetuar testes de integração não é preciso esperar que todos os módulos estejam prontos para serem integrados. Algumas abordagens comuns para a realização de integração de sistemas são as seguintes:

- Incremental
- Top down
- Bottom up
- Sandwich
- Big bang

Incremental

Neste caso, os testes de integração são feitos de maneira incremental com uma série de ciclos de testes. Em cada ciclo de testes, vários módulos são integrados e testados. O objetivo é completar um ciclo de testes e corrigir todos os erros encontrados antes de começar o próximo ciclo. O sistema é construído de forma incremental, ciclo por ciclo, até que todo o sistema fique operacional e pronto para a execução dos testes de sistema.

Top down

As abordagens top-down e bottom-up são aplicadas a sistemas com estrutura hierárquica. Na primeira abordagem, o nível superior é decomposto em módulos de segundo nível, que por sua vez ainda podem ser decompostos em módulos do terceiro nível, e assim sucessivamente. Alguns ou todos os módulos em qualquer nível podem ser terminais, ou seja, não ser mais decompostos. Nestas duas abordagens o documento de design é utilizado como referência para a integração dos módulos.

Na Figura 2.5, o módulo A do primeiro nível é decomposto em três módulos, B, C e D. No final desta decomposição, os módulos C e D são substituídos por *stubs*. Os testes são feitos entre os módulos A e B para tentar descobrir problemas de interface. Depois de concluir estes testes, o *stub* D e substituído pelo módulo real D. De seguida, são realizados dois tipos de testes. O primeiro testa a interface entre A e D e o segundo realiza os testes de regressão para procurar defeitos de interfaces entre A e B na presença do módulo D. O *stub* C é substituído pelo módulo real C e os módulos E, F e G são substituídos por *stubs*. Primeiramente, são realizados os testes entre os módulos A e C e, de seguida, os testes são feitos entre os módulos A, B, D na presença do módulo real C. Os níveis seguintes são testados da mesma maneira.

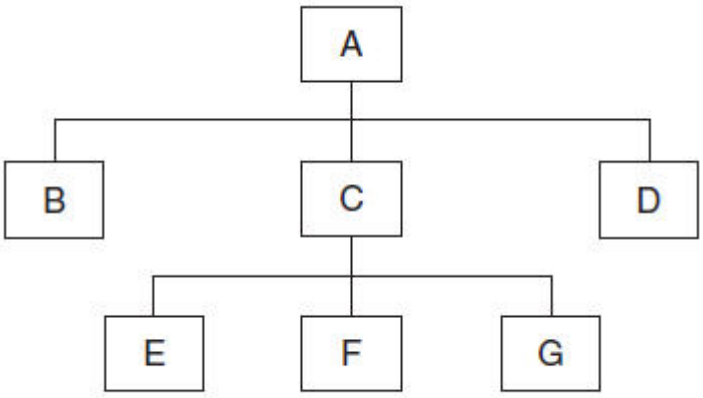

Figura 2.5: Sistema com estrutura hierárquica com três níveis e sete módulos.

Esta abordagem tem vantagens e desvantagens. As vantagens são:

- Corrigir os erros de interface encontrados torna-se mais fácil devido à integração feita de cima para baixo. No entanto, não se pode concluir que o erro de interface é sempre devido a um novo módulo Z integrado. O erro pode acontecer devido a um módulo anteriormente integrado uma vez que a capacidade dos *stubs* é limitada.
- Na avaliação do modulo Z são usados casos de testes que são posteriormente usados nos testes de regressão, após a integração de outros módulos.

As desvantagens são:

- Deve existir um documento de design que explique claramente o modo como os módulos funcionam e a ordem em que devem ser integrados.
- Uma vez que os *stubs* dos níveis inferiores têm comportamento limitado, os testes do nível superior devem ser restringidos de modo a contemplar as limitações dos *stubs*.

Bottom up

Nesta técnica os módulos começam a ser integrados dos níveis mais baixos, ou seja, a partir dos módulos que não podem ser mais decompostos. Para integrar os módulos do nível mais baixo é preciso construir um outro módulo, o *test driver*, que invoque módulos a serem integrados, ver Figura 2.6. Depois de concluir os testes, o *test driver* é substituído por um módulo real. Este processo repete-se até que todos os módulos sejam integrados. A vantagem

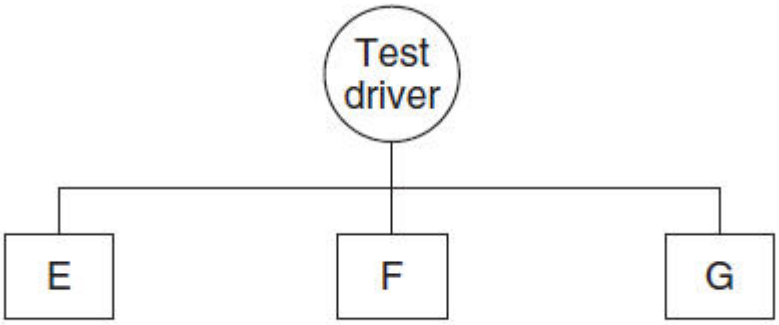

Figura 2.6: Integração bottom up dos módulos E, F e G.

desta abordagem é a construção dos *test drivers* para simular a chamada aos módulos de baixo nível. Este fornecem apenas uma capacidade limitada de teste das interfaces. A descoberta das falhas graves no sistema pode não ser possível até que todos os módulos de nível superior sejam integrados, o que constitui uma desvantagem desta abordagem.

Sandwich

A aAbordagem *sandwich* é uma mistura das abordagens top-dow e bottom-up. A abordagem *bottom-up* é usada para integrar os módulos da camada inferior e *top-down* é usada para integrar os módulos da camada superior. A vantagem desta abordagem é o facto de não serem necessários os *stubs* para simular os módulos de baixo nível.

Big bang

Nesta abordagem todos os módulos são primeiro testados individualmente. De seguida, estes são integrados para construir um sistema que depois é testado como um todo. Esta abordagem é útil para pequenos sistemas.

2.4.3 Testes de sistema

Depois de todos os módulos estarem integrados num sistema operacional devem ser feitos os testes de sistema [6]. Estes são feitos usando a técnica black-box que foi vista na seção 2.4.2. O objetivo destes testes é verificar se o sistema corresponde aos requisitos definidos. Seguem abaixo vários tipos de testes desta categoria.

Testes de robustez

Robustez significa a sensibilidade de um sistema como um todo, quando o ambiente operacional corresponde a situações não previstas ou não desejadas. O objetivo é fazer o sistema falhar, não como um fim em si mesmo, mas como um meio para encontrar erros. Os principais tipos de testes de robustez podem ser vistos na Figura 2.7.

Boundary value

Este tipo de teste está focado na validação dos limites das condições e na inserção de entradas especiais (por exemplo *strings* reservadas pelo sistema). A inserção de valores de entrada inválidos faz parte deste tipo de testes, avaliando se o sistema devolve alguma mensagem de erro ou inicia algum processo de tratamento de erro.

Power cycling

Estes testes servem para assegurar que, quando há uma falha de energia num ambiente de produção, o sistema consigue recuperar o funcionamento normal quando a energia voltar.

Online insertion and removal

Estes testes foram desenhados para assegurar que a inserção e a remoção dos módulos durante o funcionamento normal do sistema não provoca o encerramento ou reinício do sistema, e para garantir que depois da correção das falhas o sistema volta ao funcionamento normal. O objetivo destes testes é garantir o funcionamento do sistema sem falhas quando um módulo com defeitos é substituído.

High availability

Este tipo de testes é desenhado para verificar a redundância dos módulos individuais, incluindo o software que os controla. O objetivo é verificar se o sistema recupera rapidamente depois de falhas de hardware e/ou de software, sem que haja algum impacto no funcionamento do sistema. Este conceito é conhecido como tolerância a falhas.

Degraded node

Estes testes servem para verificar o funcionamento de um sistema quando uma porção do sistema se torna inoperante.

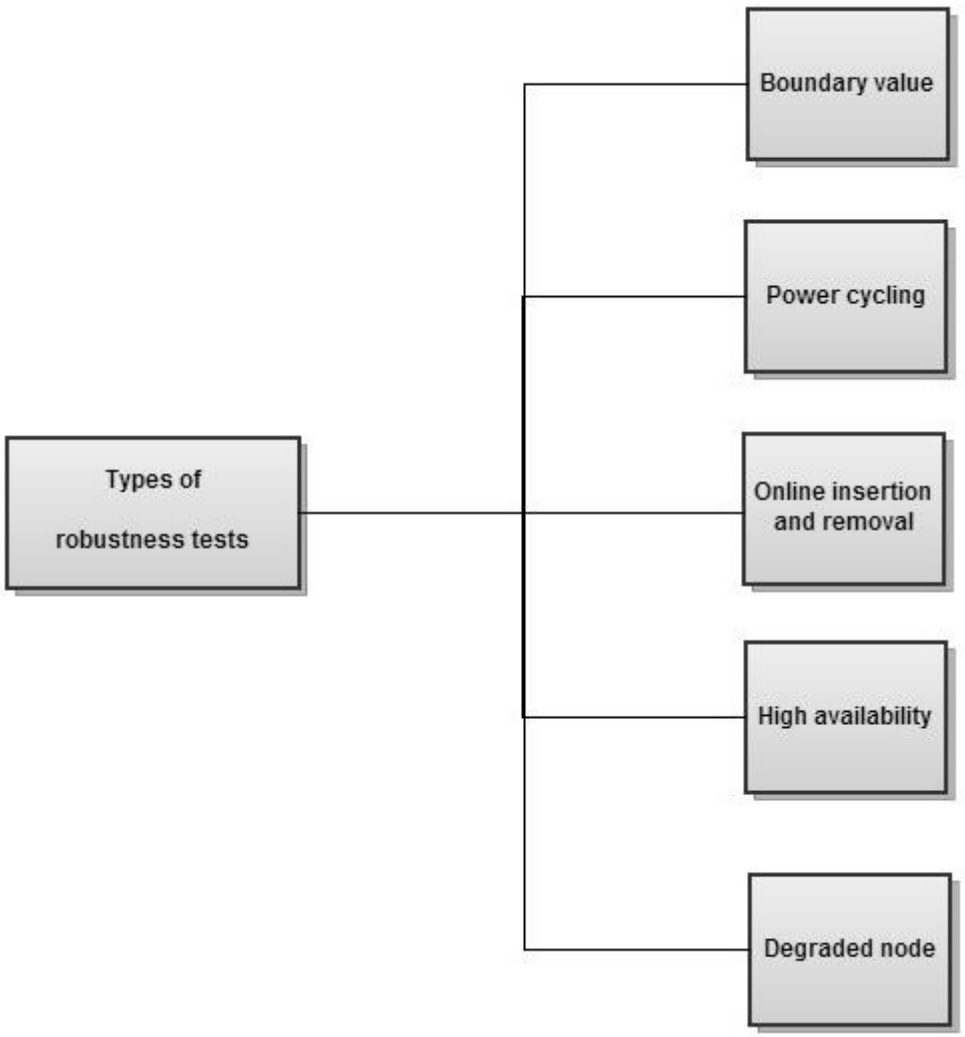

Figura 2.7: Tipos de testes de robustez.

Testes de performance

Os testes de performance são desenhados para perceber o desempenho do sistema. As métricas de desempenho variam de aplicação para aplicação. Esta categoria de testes é projetada para verificar:

- tempos de resposta
- tempos de execução (memória, CPU)
- utilização dos recursos
- taxa de tráfego
- volume de tráfego

Os resultados dos testes de performance são avaliados e comparados com resultados anteriores para concluir sobre a eficácia das alterações efetuadas no código fonte.

Testes de stress

O principal objetivo destes testes é determinar e avaliar o comportamento de um sistema ou de uma componente quando a carga é superior à sua capacidade prevista. Estes testes garantem que o sistema funciona corretamente em condições piores do que a carga máxima planeada.

Testes de estabilidade

Os testes de estabilidade servem para assegurar que um sistema é estável durante um longo período de tempo sob a carga nominal. O principal objetivo é avaliar o funcionamento do sistema em condições reais durante várias semanas/meses sem necessidade de reiniciar o sistema. Os testes de estabilidade servem para determinar os seguintes problemas:

- Memory leaks.
- Capacidade projetada insuficiente.

Testes de regressão

Nesta categoria de testes, não são criados novos testes, mas são seleccionados e executados os já existentes para garantir que nada deixou de funcionar corretamente numa nova versão do software. O objetivo destes testes é verificar que não se introduziu algum defeito na parte do código fonte não alterado, devido às alterações feitas noutras partes de sistema.

Caso de teste	Variável X	Variável Y	Variável Z
TC1	True	0	Q
TC2	True	5	R
TC3	True	0	R
TC4	False	0	Q
TC5	False	5	R
TC6	False	5	Q

Tabela 2.1: Testes pairwise do sistema XPTO

Testes funcionais

Os testes funcionais [6] servem para verificar se o sistema desenvolvido corresponde aos requisitos definidos. Isto quer dizer que o sistema tem de estar de acordo com as suas especificações e executar as suas funções corretamente. Normalmente os testes funcionais envolvem cinco etapas:

- identificar as funções que o sistema deverá executar.
- criar os valores de entrada com base na especificação da função.
- determinar os valores de saída com base na especificação da função.
- executar casos de teste.
- comparar resultados reais com os esperados.

Na maioria dos casos, é grande o número de valores de entrada e suas combinações. Existem várias técnicas de resolução deste problema que serão descritas a seguir.

Testes pairwise

Os testes pairwise é um caso especial de todas as possíveis combinações dos valores de entrada para um sistema. Supondo que há n variáveis de entrada denotados por $\left[v_1, v_2, v_3, ..., v_i, ..., v_n\right]$, e que cada uma pode assumir k valores, o numero de vetores de entrada possíveis é k^n. O objetivo dos testes pairwise é cobrir mais do que um caso de teste com uma só combinação dos valores de entrada. Considera-se um sistema *XPTO* com três variáveis de entrada, X, Y e Z. Os possíveis valores da variável X são *true* ou *false*, da variável Y são *0* e *5* e da variável Z são *Q* e *R*. O número total de vetores de entrada é oito, mas com os seis casos de teste na Tabela 2.1 é possível cobrir todas as combinações para cada par de variáveis.

Análise de valor limite

A ideia central da análise dos valores limite é seleccionar um conjunto de valores dentro e fora dos limites de uma condição. O objetivo destes testes é encontrar falhas por incorrecta implementação dos limites. Na prática, muitas vezes os programadores esquecem-se de testar condições limite.

2.4.4 Testes de aceitação

Depois de realizar os testes unitários, testes de integração e testes de sistema, tendo corrigido todos os problemas encontrados, o produto está pronto para ser entregue ao cliente para a realização dos testes de aceitação [6]. O cliente realiza testes de aceitação com base nas suas expectativas em relação ao produto. Os testes de aceitação são testes formais que servem para determinar se o sistema satisfaz ou não cada critério de aceitação. Estes critérios são itens de funcionalidade que, quando cumprem a sua especificação, tornam o produto aceitável para o cliente.

Existem duas categorias de testes de aceitação:

- Bussiness Accceptance Testing
  Dentro da organização do desenvolvimento, o fornecedor verifica os critérios para garantir que o produto passa no *User Acceptance Testing*.
- User Acceptance Testing
  O cliente verifica se o produto satisfaz os critérios de aceitação.

2.5 Processo de qualidade

Hoje em dia o conceito de "processo" desempenha um papel muito importante no desenvolvimento de software. Em engenharia de software, um processo é compreendido como a execução de um conjunto de atividades com vista ao desenvolvimento de um produto de software. As actividades incluem a aplicação de métodos, técnicas, práticas e estratégias. O conceito de processo não se aplica apenas ao desenvolvimento do código, mas também aos documentos de requisitos, plano do projeto e manual de utilizador.

O desenvolvimento de software consiste em diferentes processos para diferentes tarefas, como por exemplo, a definição dos requisitos de sistema, a construção da especificação funcional do sistema, o projeto do sistema, testes do sistema e manutenção deste, realizados no ciclo de vida de um sistema.

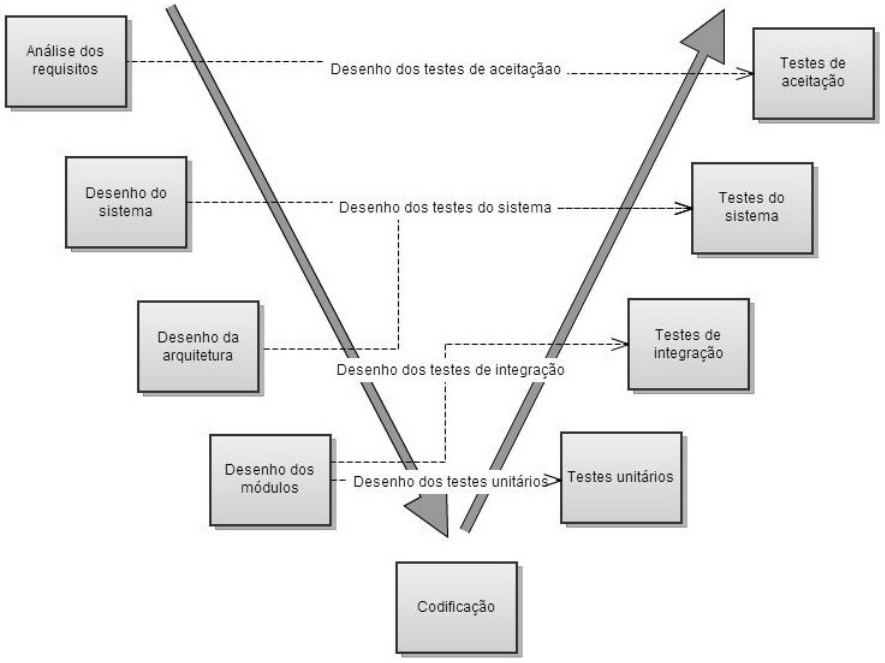

Figura 2.8: Modelo V [8].

Hoje em dia existem vários modelos e técnicas para o desenvolvimento de software, sendo neste relatório abordado um modelo que demonstra a relação entre o processo de desenvolvimento e o processo da qualidade. Este modelo, o modelo V [8], consiste em estabelecer a relação entre as fases de desenvolvimento e fases de testes, assegurando que a garantia da qualidade está presente em todo o ciclo de vida de um produto, como se pode ver na Figura 2.8.

Análise dos requisitos

A primeira fase é a análise dos requisitos. Estes são recolhidos conforme as necessidades do utilizador. Normalmente o documento dos requisitos descreve o funcionamento do sistema, a interface, a segurança, etc, e na base deste são feitos os testes de aceitação pelo cliente.

Desenho do sistema

O desenho do sistema é um documento feito para os engenheiros avaliarem os requisitos definidos e arranjarem técnicas para a implementação dos mesmos. Este documento deve conter a organização geral do sistema, estrutura de interfaces e de dados, etc., para permitir a realização dos testes ao sistema.

Desenho da arquitetura

Esta fase refere-se à arquitetura do software. Tipicamente o documento de conceção da arquitetura consiste numa lista de módulos com uma breve descrição de cada um destes, por exemplo, contendo a estrutura da base de dados, possíveis dependências, etc. Este documento é usado na organização e criação dos testes do sistema.

Desenho dos módulos

A conceção dos módulos refere-se a conceção de baixo nível. O sistema concebido é dividido em unidades e cada uma destas é explicada para que o programador possa iniciar a fase de codificação. A conceção de baixo nível contém a especificação detalhada do funcionamento lógico, em pseudocódigo. Por exemplo:

- tabelas de bases de dados com tipo e tamanho de cada elemento
- lista das mensagens de erro
- todos os detalhes da interface

Toda a informação de conceção dos módulos é usada para criar os testes de integração e testes unitários.

2.6 Métricas da qualidade

Um elemento chave de qualquer processo de engenharia é a medição. As medidas são usadas para melhorar o entendimento dos atributos dos modelos desenvolvidos e avaliar a qualidade do produto ou de um sistema desenvolvido. Ao contrário de outras engenharias, a engenharia de software não é baseada em leis quantitativas básicas. Ao invés disso, tenta-se arranjar um conjunto de medidas indiretas que fornecem uma indicação da qualidade do software.

Embora as métricas para software não sejam absolutas, estas fornecem uma maneira de avaliar a qualidade através de um conjunto de regras bem definidas. As métricas existentes são inúmeras, pelo que vão ser descritas apenas algumas.

2.6.1 Cobertura de código

A cobertura de código é uma medida que permite quantificar as linhas/blocos de código realmente executadas aquando da execução dos testes automatizados. Há uma série de critérios de cobertura, sendo os principais:

- **Line coverage** - Representa a percentagem das linhas de código executadas por testes unitários.
- **Branch coverage** - Representa a percentagem de ramificações possíveis em estruturas de controlo de fluxo que foram seguidas durante a execução de testes unitários.

2.6.2 Complexidade ciclomática

A complexidade ciclomática de uma parte do código fonte é a quantidade de caminhos independentes no código. Se o código contém uma condição, então há dois caminhos possíveis, um quando a condição é avaliada como verdadeira, e outro quando a condição é avaliada como falsa.

Matematicamente, a complexidade ciclomática de um programa estruturado é representada por um grafo direcionado que contém os nós básicos do programa e arestas entre esses nós. Um nó é um dos pontos de decisão como é o caso de uma condição *if-else*. As arestas representam as ligações entre os nós. Por exemplo, uma condição *if-else* vai criar dois nós e a ligação entre o nó pai e cada nó descendente é representada por uma aresta. Esta complexidade pode ser calculada de seguinte forma, $M = E - N + 2 * P$, onde:

Em que:

- **M** - complexidade ciclomática
- **E** - número de arcos
- **N** - número de nós
- **P** - número de nós de saída

Um exemplo de código a analisar quanto ao cálculo da complexidade ciclomática é apresentado na Figura 2.9. O diagrama de fluxo está representado na Figura 2.10. O nó vermelho é o ponto de entrada do método, e o nó azul é o ponto de saída. Os restantes nós representam o fluxo de controlo. Neste caso, o número de nós é igual a 7, o número de arestas que ligam os nós é

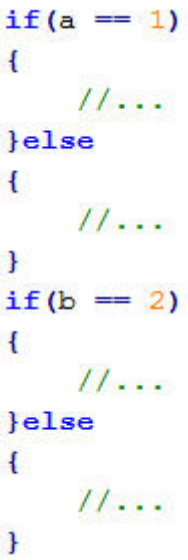

Figura 2.9: Exemplo do código para o cálculo da complexidade ciclomática.

igual a 8 e o número de nós de saída é igual a 1. Portanto, a complexidade ciclomática no presente exemplo será igual à 8-7+(2*1)=3.

2.6.3 Linhas de código fonte

Linhas de código fonte é uma medida de software usada para medir o tamanho de um programa, através da contagem do número de linhas do código fonte de uma aplicação. As linhas de código fonte são normalmente usadas para prever a quantidade de esforço que será necessário para desenvolver uma aplicação ou *feature*, bem como para fazer a estimativa de produtividade de programação, quando o software é produzido.

2.6.4 Análise da documentação do código

A métrica "análise da documentação do código" é composta por duas categorias principais – a análise dos comentários no código e a análise da documentação do métodos públicos (API). Esta métrica serve para identificar as partes documentadas do código produzido. A "Análise da documentação do código" não analisa a qualidade dos comentários. Normalmente, esta métrica é composta pelas seguintes submétricas:

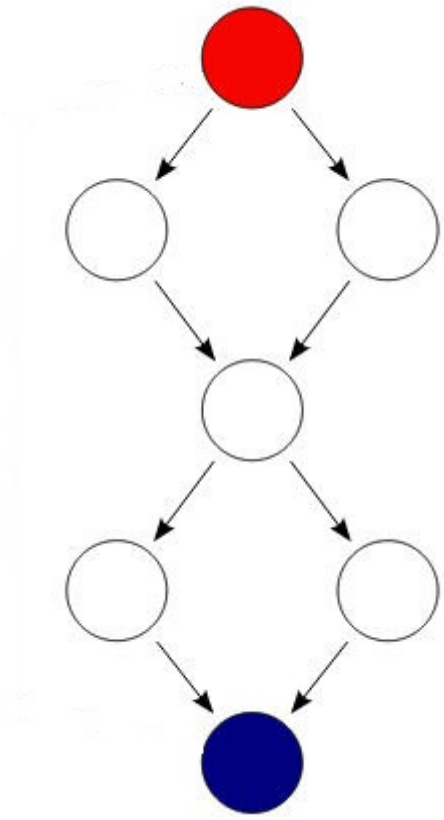

Figura 2.10: Exemplo do fluxo dos dados para o cálculo da complexidade ciclomática.

- **Comentário vazios** - permite identificar os comentários sem conteúdo.
- **Linhas comentadas** - permite calcular o número das linhas que contém comentários.
- **Percentagem de comentários no código** - a métrica calcula-se de seguinte forma: *Linhas comentadas*/(*Linhas comentadas*+*Linhas de código sem comentários*) * 100.
- **Percentagem de documentação dos métodos públicos** - a métrica calcula-se de seguinte forma: *Número de métodos públicos documentados*/*Número total de métodos públicos* * 100.
- **Número de métodos públicos não comentados**

2.6.5 Duplicação de código

A métrica "duplicação de código" permite quantificar as partes duplicadas do código. Esta métrica é composta pelas seguintes submétricas:

- **Blocos duplicados**.
- **Ficheiros duplicados**.
- **Linhas duplicadas**.
- **Percentagem de duplicações** - esta métrica calcula-se de seguinte forma: *Linhas duplicadas* / *Linhas do código* * 100

2.6.6 Motivação para o uso de Métricas da Qualidade

Independentemente da métrica usada, estas têm sempre os mesmos objetivos:

- Melhorar o entendimento da qualidade do produto.
- Verificar a efetividade do processo.
- Melhorar a qualidade do trabalho realizado a nível do projeto.

2.7 Conclusão

O processo da qualidade é um processo essencial para as empresas que desenvolvem o software. Normalmente, o processo da qualidade está integrado no processo de desenvolvimento, o que ajuda a perceber em detalhe as funcionalidades do produto a fim de conseguir criar um conjunto dos casos de testes corretos. No próximo capítulo é apresentado um processo de qualidade concreto.

Capítulo 3

Garantia de Qualidade do Produto Pulse

Nesta secção é dada um visão geral do produto da FeedZai, da organização atual da equipa de garantia de qualidade, das tarefas por esta exercidas, e dos testes realizados em cada release do produto.

3.1 Introdução ao Pulse

FeedZai Pulse é a próxima geração de inteligência de negócio em tempo real e é também uma ferramenta de análise. O núcleo do Pulse é um motor de processamento de eventos complexos que integra os dados em tempo real com informações históricas, criando uma plataforma completa para gerir, compreender e extrair valores a partir da grande quantidade de dados que fluem nas empresas modernas.

O objetivo do Pulse é fornecer informação atualizada a cada segundo aos decisores usando Key Performance Indicators (KPIs) sobre determinadas operações, comparar KPIs com os seus valores históricos e gerar ações e alarmes quando ocorre um evento inesperado. Na Figura 3.1 é demonstrado o conceito de KPI, e como este pode ser comparado com o passado e utilizado para fazer previsões sobre o futuro.

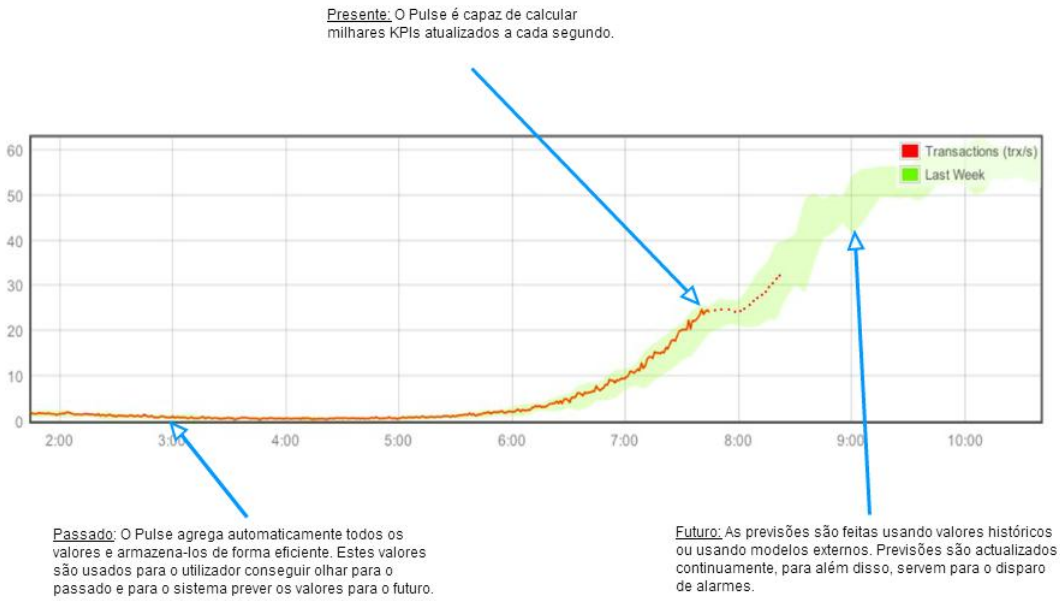

Figura 3.1: Como o Pulse trabalha com o passado, o presente e o futuro. Fonte: Apresentação interna

3.1.1 Conceitos

- Uma **stream** representa uma fonte de dados em tempo real.
- Uma **hierarquia** representa o modo como se pretende navegar os dados para os KPIs relacionados.
- Um **KPI** representa uma métrica de negócios que deve ser monitorizada em tempo real, podendo ser associado a uma só hierarquia.
- Uma **baseline** representa os valores esperados para um KPI num determinado momento.
- Um **alerta** permite monitorizar os seus KPIs e desencadear ações quando certas condições são satisfeitas.

3.1.2 Arquitetura

Em termos da arquitetura, os dados entram no Pulse em tempo real através de adaptadores de entrada que podem ser *Sockets*, *WebServices*, *HTTP-*

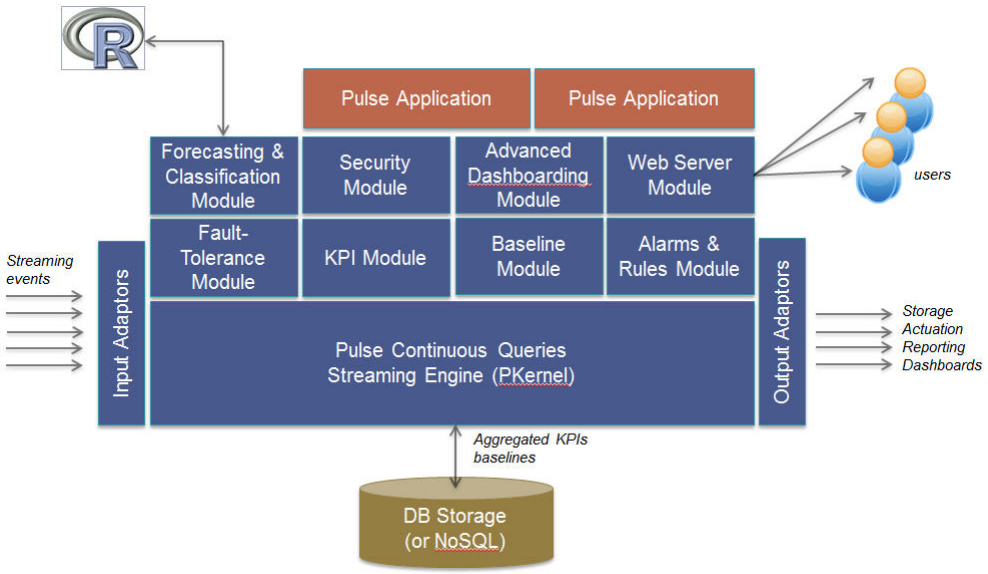

Figura 3.2: Arquitetura do Pulse. Fonte: Apresentação interna

JSON, etc. Os eventos são processados no *Pulse Application Server*, e no fim a informação pode ser encaminhada para outros sistemas ou disponibilizada utilizando uma interface web, através de um adaptador de saída *HTTP* [10]. Esta arquitetura é ilustrada na Figura 3.2.

O motor de processamento de eventos utiliza "queries contínuas" que produzem resultados constantamente de acordo com a informação a chegar em tempo real. Uma "query contínua" normalmente funciona através de uma janela temporal. O sistema mantém os eventos correspondentes à janela temporal em memória para os cálculos (por exemplo cálculo da média). Os novos eventos que entram no sistema são adicionados à janela e os que se tornam obsoletos são removidos desta janela. Se o sistema calcula a média, este só precisa de saber o número "total" e o número de eventos utilizados. Estes dois valores são actualizados com a entrada de novos eventos que vêm para a janela (incrementa o valor total e o contador) e com a saída dos dados expirados (decrementa o valor total e o contador). Ao contrário do que acontece no uso de uma base de dados, não há necessidade de rever todos os registos para o cálculo de novos resultados. Para além disso, os eventos que não são usados para os cálculos podem ser temporariamente enviados para o disco.

O *Pulse* é um produto *multi-module* sendo constituído pelos seguintes módulos principáis:

- **PKernel** é o motor de processamento de eventos. Permite executar as *queries* contínuas, escritas em *Pulse Query Language* e processá-las afim de produzir diversos resultados.
- **Rules Engine** é um motor de processamento de regras (usando o paradigma Event-Condition-Action). Permite detectar padrões e lançar acções para notificação.
- **Monitor** permite registar *listeners* sobre os ficheiros e pastas de forma centralizada.
- **Stats** permite recolher diferentes métricas do servidor *Pulse*.
- **Webapps** é um servidor *web* integrado no *Pulse* que permite disponibilizar as aplicações web no contexto do produto.
- **KPI** permite criar e configurar a política de armazenamento de indicadores de negócio.
- **Baseline** permite, com base nos valores de *KPI* ou em valores fixos, determinar valores aceitáveis para um determinado *KPI* no futuro próximo.
- **Security** permite gerir todos os recursos relacionados com a segurança, tais como utilizadores, permissões e filtros.

O processamento de eventos é um fluxo de dados em *stream*. Na prática, um fluxo pode ser um *TCP socket* através do qual os dados entram para o motor de processamento de dados. Por exemplo, na Figura 3.3 está representado um exemplo da aplicação de monitorização de tráfego de veículos que entram e saem da autoestrada. Um evento é frequentemente chamado de pacote de dados. Neste exemplo, a informação contida num evento pode ser a matrícula (*plate*), a velocidade (*speed*), o número de identificação(*sensor-id*) do veículo e o *timestamp* que contém a hora a que o evento ocorreu.

Quando um evento chega ao motor de processamento de eventos, é analisado numa série de procedimentos que produzem um resultado. No caso do *Pulse*, estes procedimentos são escritos numa linguagem específica, chamada *Pulse Query Language*. A Figura 3.4 mostra um exemplo de uma *query* que faz a filtragem de veículos que circulam com velocidade maior que `120 km/h`.

Neste exemplo, *speed-readings* é o fluxo de entrada que contém as leituras de velocidade dos veículos. Esta *query* resulta num novo fluxo chamado de *fast-readings*, no qual só os veículos com a velocidade superior a `120 km/h` são incluídos.

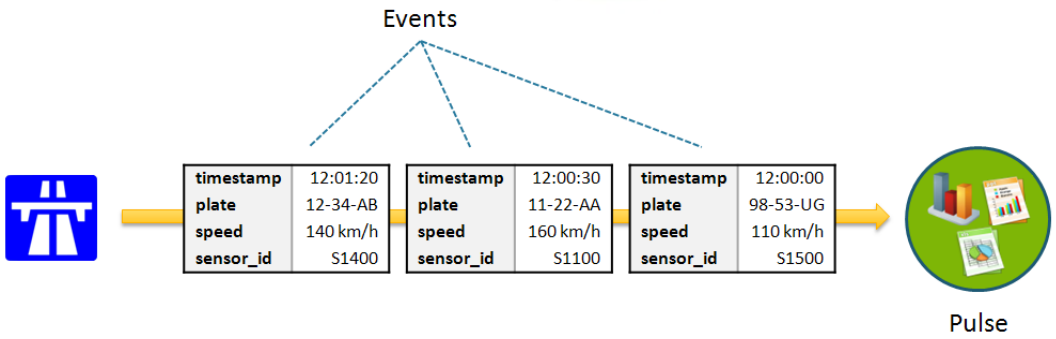

Figura 3.3: Um exemplo de um fluxo de eventos em *Pulse*.

```
fast_readings =
    from speed_readings
    where speed > 120;
```

Figura 3.4: Um exemplo da *Pulse Query Language*.

É importante realçar que embora estes procedimentos sejam chamados *queries*, são no entanto profundamente diferentes das *queries* tradicionais de uma base de dados. Isto acontece devido à natureza do processamento de eventos ser contínua. Pelo contrário, uma *query* de uma base de dados tradicional é aplicada a um conjunto finito de dados e o processo é terminado quando o resultado é produzido. Em aplicações de processamento de eventos, as *queries* são submetidas aquando do arranque de uma aplicação. Estas *queries* aguardam os novos eventos, processam-nos e produzem um novo resultado, voltando de seguida ao estado de espera até receberem os novos eventos. As *queries* são normalmente terminadas quando uma aplicação pára a sua execução. Por essa razão, estas *queries* são chamadas de *queries* contínuas.

3.2 Ciclo de release

No Pulse Software Review Board (SRB) [11] são decididas as datas de release de novas versões do produto e as novas funcionalidades a desenvolver para cada release. A reunião é presidida por Paulo Marques (CTO), sendo ainda membros Diogo Guerra (SVP Product Development), Pedro Barata (SVP Project Delivery), Pedro Bizarro (CSO), Nuno Pires (SVP Sales) e Nuno

Figura 3.5: Organização da release em sprints. Fonte: Apresentação interna

Sebastião (CEO e Product Placement).

Cada ciclo de release deve durar cerca de 4 meses, dependendo da decisão do SRB depois de considerar a estratégia da empresa, os requisitos do cliente/-produto e eventos comerciais.

O desenvolvimento segue uma metodologia *Scrum*. Em termos de *sprints*, cada versão deve ter quatro sprints mensais, sendo o último dedicado a consolidar o desenvolvimento, garantir a qualidade da release e preparar a entrega da release. No final de cada sprint uma versão com código distinto deve ser emitida, a qual vai testada pela equipa da garantia da qualidade durante o próximo sprint. Em cada sprint consequente é emitida uma nova versão que deverá passar pela garantia de qualidade. Na Figura 3.5 pode ser vista a organização da release em sprints.

No meio do último sprint, é feito um *Release Candidate* (RC). Nesta release são verificadas todas as funcionalidades implementadas com os testes respetivos. Se existirem problemas graves no RC, é criado um RC2. Os testes são repetidos e, se os problemas encontrados forem resolvidos, cria-se a versão *General Availability* (GA). Depois de criar GA, verifica-se se ficheiros binários, JSDoc, javadoc e conjuntos de dados estão situados corretamente e são acessíveis do exterior. No final, é feito o relatório de garantia de qualidade com os resultados de todos os testes efetuados para cada RC.

3.2.1 Scrum

O trabalho deste projeto segue a metodologia *Scrum*, que é "uma estrutura processual (framework) para o desenvolvimento e manutenção de produtos complexos. O *Scrum* é uma estrutura constituída por equipas Scrum, com as suas respectivas funções, por eventos, artefatos e regras. Cada componente desta estrutura serve um propósito específico e é essencial para o uso e sucesso do Scrum." [12, pp.3].

Três pilares sustentam qualquer implementação de processos empíricos: transparência, inspeção (verificação) e adaptação.

Transparência

Os aspetos significativos do processo devem ser visíveis para aqueles que são responsáveis pelos resultados. [12, pp.4] A transparência exige assim que esses aspetos sejam definidos com base num padrão comum. Desta forma, os diferentes observadores partilham um entendimento comum sobre o que está a ser observado.

Inspeção

Os utilizadores do Scrum devem, com frequência, inspecionar os artefactos Scrum e o progresso face ao objetivo final, de modo a detectar variações indesejáveis. [12, pp.4]

Adaptação

Se um inspetor verifica que um ou mais aspetos do processo são desviantes além de limites aceitáveis, e que por conseguinte o produto resultante será inaceitável, então ou o processo ou o objecto de processamento devem ser ajustados. [12, pp.4] Os ajustes devem ser realizados o mais rápido e mais cedo possível de forma a minimizar mais desvios.

3.2.2 Sistema de controlo de versão

Este produto usa o *GIT* [13], que é um sistema de controlo de versão. O modelo de ramificação adoptado para o Pulse contem cinco tipos de ramos:

- master - o ramo com a última versão estável.
- desenvolvimento - o ramo principal do desenvolvimento.
- funcionalidades - o ramo para novas funcionalidades.
- qualidade - o ramo para a equipa da garantia da qualidade.
- hot-fix - o ramo para correção de erros e pequenas releases.

Na Figura 3.6 são ilustrados estes cinco ramos. No início de cada *sprint* são criados diferentes *branches* para cada funcionalidade a desenvolver. Para além disso, faz-se *merge* do *branch* principal com o *branch* de Garantia de Qualidade para que a equipa possa implementar os teste funcionais e testes de regressão. Ao finalizar o desenvolvimento da nova funcionalidade, o

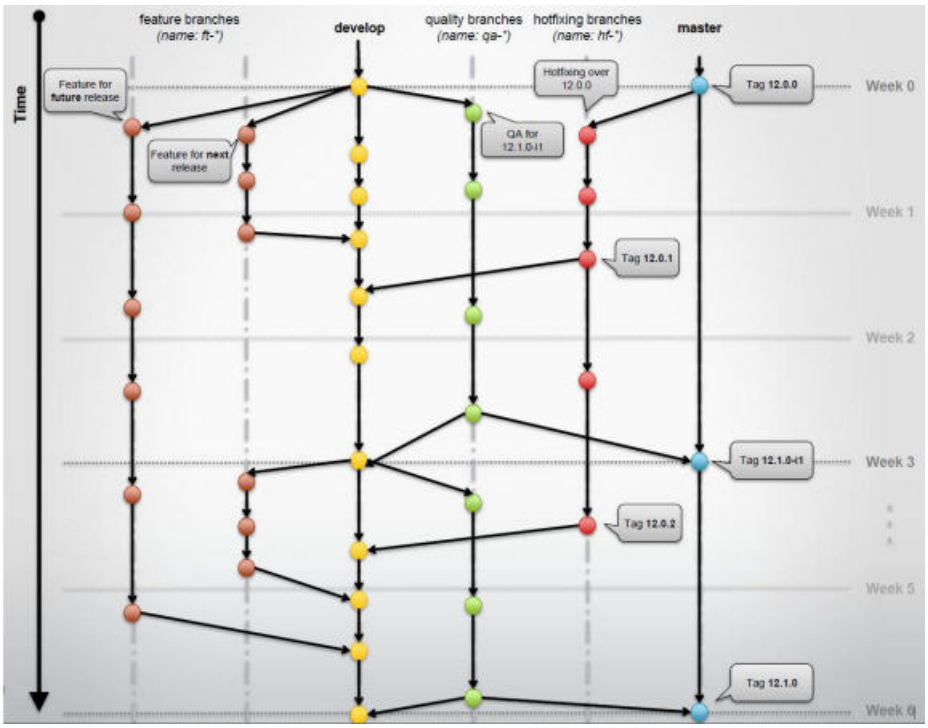

Figura 3.6: Modelo de ramificação do Git.

programador integra o branch criada com o *branch* principal. Após ao lançamento de uma nova versão do produto é criada um *branch hot-fix* para corrigir eventuais problemas.

3.3 Garantia da Qualidade

A garantia da qualidade de software não é algo em que se comece a pensar depois do código ser gerado. A Garantia de Qualidade é uma atividade que é aplicada ao longo de todo o processo de engenharia de software.

Na FeedZai, o departamento da garantia da qualidade é responsável por várias tarefas. Todos os dias se verificam então os seguintes pontos:

- a existência de problemas, ou com a execução dos testes, ou com a compilação do código no sistema de integração contínua, *Jenkins* [14] (ferramenta descrita na secção 3.4), sendo investigada a origem destes problemas com a solicitação do programador para a sua resolução.
- o desempenho do produto, utilizando a ferramenta PerfP [15] (também

descrita na secção 3.4).

- não existência dos problemas nas compilações e testes *NIGHTLY* no Jenkins. Todos os problemas encontrados e as suas soluções são registados no Confluence [16] (ferramenta descrita na seção **Ferramentas de apoio**).

No final de cada *sprint* é feita uma cópia do código para o ramo da qualidade.O departamento de Garantia de Qualidade é responsável por verificar o seguinte ponto:

- a criação de uma lista das funcionalidade a serem testadas.

Antes de criar o RC a Garantia de Qualidade tem de verificar:

- se todos os branches criados desde a primeira release foram movidos para o ramo de desenvolvimento.
- se os defeitos encontrados anteriormente foram realmente corrigidos. Caso possível, verificar se existem testes automáticos para estes problemas, que serão adicionados aos testes de regressão existentes.
- se existem problemas bloqueantes ou críticos na ferramenta da recolha das métricas de qualidade, *Sonar* (ferramenta descrita na secção **Ferramentas de apoio**).

Todos os bugs encontrados durante o desenvolvimento das novas funcionalidades e também nas fases *RC1*, *RC2* e *GA* são registados no *issue tracker Jira*. Depois da *issue* ser criada, esta está em estado *new*. Caso o programador necessite de alguma informação para a reproduzir, ele altera o estado da *issue* para *need info*. Depois de resolver a *issue*, esta é marcada como *waiting validation* e passa automaticamente para equipa de Garantia de Qualidade. Todas as *issues* resolvidas são verificadas pela equipa de Garantia de Qualidade e no caso não terem sido resolvidas, são reabertas. Depois da *issue* ser verificada pela Garantia de Qualidade, passa para o estado *waiting automation* para ser automatizada e incluída na testes de regressão. Depois da *issue* ser automatizada, se possível, passa para *resolved*. No final da release, todas as *issues* em estado *resolved* passam para o estado *closed*.

Depois da primeira RC, a equipa do departamento da qualidade executa o ciclo de testes, obtém os resultados dos testes de análise estática do código e verifica se não existem problemas com os direitos de propriedade. Caso não haja problemas críticos, o GA é feito. Nesta última fase, a Garantia de Qualidade efetua as últimas verificações dos ficheiros criados e da documentação, e cria o documento de qualidade onde é apresentada toda a informação

obtida dos testes realizados em comparação com as versões anteriores.

3.4 Ferramentas de apoio

A FeedZai utiliza várias ferramentas para diferentes necessidades, algumas das quais são descritas de seguida. Estas ferramentas permitem organizar a informação relativa ao produto desenvolvido e efetuar os diferentes tipos de testes.

3.4.1 Maven

Maven [17] é uma ferramenta para gestão e automatização de projetos em Java. Esta possui um modelo de configuração baseado no formato XML. O *Maven* utiliza uma construção conhecida como *Project Object Model* (POM), que descreve todo o processo de construção de um projeto de software, as suas dependências de outros módulos e componentes, e a sua sequência de construção. O *Maven* contém tarefas pré-definidas que realizam funções bem conhecidas, como compilação e empacotamento do código. Uma característica chave do *Maven* é a sua capacidade para trabalhar em rede, podendo obter plugins de um repositório remoto.

O *Maven* é baseado no conceito central de um ciclo de vida de construção. O ciclo de vida é representado na Figura 3.7.

Cada ciclo de vida de construção é definido por uma lista de diferentes fases de construção. As fases são executadas sequencialmente para completar o ciclo de vida, e são a seguir apresentadas:

- **validate** - avaliar se o projeto está correto e se todas as informações necessárias estão disponíveis
- **compile** - compilar o código fonte
- **test** - testar o código compilado usando uma framework para a execução de testes unitários
- **package** - compactar o código em formato de distribuição
- **integration-test** - processar e, caso seja necessário, distribuir o pacote no ambiente onde os testes de integração podem ser executados

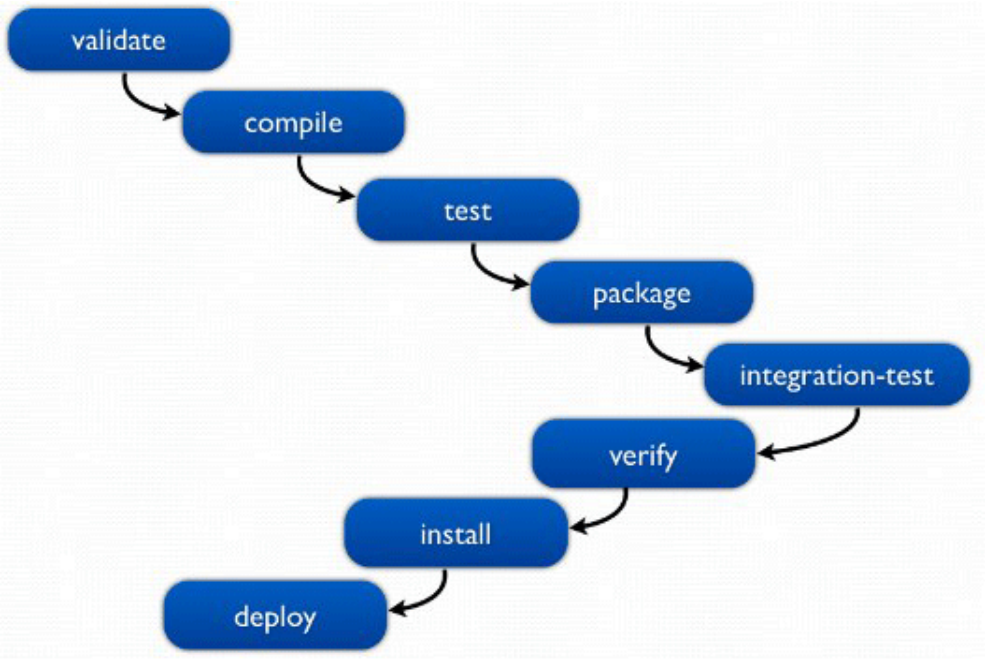

Figura 3.7: Maven. Ciclo de vida da construção.

- **verify** - executar todas as verificações para validar o pacote com a sua correspondência a todos os critéros da qualidade
- **install** - instalar o pacote no repositório local
- **deploy** - terminar a execução em ambientes de integração e de release, e copiar o pacote final para o repositório remoto para partilhar com os outros programadores e projetos

3.4.2 Nexus

Nexus [18] é uma ferramenta que serve de repositório de todos os tipos de artefactos. O seu principal objetivo é guardar todos os plugins utilizados pelo produto, diminuindo os acessos à Internet para obtenção destes plugins. Na Figura 3.8 é representado o funcionamento do Nexus.

Quando o programador solicita algum plugin que não se encontra no repositório do *Nexus*, este último faz um pedido via Internet para a "Central", a fim de obter o plugin solicitado. Nas próximas vezes, que o mesmo plugin for solicitado, o *Nexus* vai disponibilizá-lo ao programador sem realizar o pedido

Figura 3.8: Funcionamento do Nexus.

a Internet.

3.4.3 Jenkins

Jenkins é uma ferramenta escrita em *open source Java* que serve para a integração contínua. A integração contínua é uma pratica de desenvolvimento de software onde os programadores de um projeto integram o código desenvolvido num sistema de controlo de versão, para depois este ser analisado por um build automatizado. Esta ferramenta é utilizada em projetos de várias linguagens e plataformas como Java, .NET, Ruby, PHP, Groovy, etc. *Jenkins* é bastante robusto, extremamente flexível e extensível, e possui centenas de plugins *open source* que podem ser instalados automaticamente.

A FeedZai utiliza *Jenkins* para todos os componentes do Pulse, para compilar, testar e analisar continuamente o sistema.

3.4.4 Sonar

Sonar [19] é uma plataforma *open source* de qualidade de software composta por várias ferramentas de análise estática de código, que tem por objetivo extrair as métricas de software para melhorar a qualidade do código. Esta ferramenta tem algumas limitações:

- por defeito só é possível trabalhar com as linguagens de programação **Java** ou **JavaScript**, outras linguagens não tem suporte oficial.
- não há possibilidade de visualizar a evolução de uma determinada métrica ao longo do todo tempo.

- não permite a existência de várias linguagens de programação num só projecto do **Maven**.

3.4.5 Confluence

Confluence é a plataforma usada para organizar toda a documentação interna e pública sobre o produto, e contém alguma informação geral sobre a Feedzai. Para além disso a FeedZai utiliza esta ferramenta para especificar os requisitos das funcionalidades futuras. Ainda mais, esta ferramenta contém informação para as equipas de desenvolvimento como, por exemplo, regras para realizar um **commit** no repositório da FeedZai, explica o processo de desenvolvimento utilizado na empresa, e muito mais. Toda a informação que qualquer pessoa da FeedZai precisa, encontra-se nesta ferramenta.

3.4.6 Jira

Jira [20] é o "issue tracker" usado na FeedZai para acompanhar a evolução do relatório de *bugs*, desenvolvimento e planeamento de releases. A equipa de Garantia de Qualidade utiliza esta ferramenta não só para submeter os problemas encontrados ou sugestões de melhoramentos de usabilidade mas também para especificar o plano de testes para cada fase da release.

3.4.7 PerfP

Performance Evaluation tool for Pulse (PerfP) é uma ferramenta desenvolvida pela FeedZai com o objectivo de medir o desempenho do Pulse nos testes de stress usando diferentes **data sets**. O objetivo desta ferramenta é obter métricas sobre eventos injetados, utilização de CPU e memória, aquando da execução do Pulse.

3.4.8 PMD

PMD [21] é uma ferramenta com um conjunto de regras estáticas para analisar o código *Java* que identifica possíveis problemas, tais como:

- *Possible bug* - algum bloco vazio (try/catch/finally/switch)

- *Dead code* - variáveis locais, parâmetros, métodos privados, etc. não utilizados
- *Condições vazias*
- *Classes com alta complexidade ciclomática*
- *Código duplicado*

3.4.9 CheckStyle

Checkstyle [22] é uma ferramenta para a análise estática do código, usada no desenvolvimento de software para verificar se o código fonte *Java* está de acordo com as regras de estilo de código. Esta ferramenta permite adicionar novas regras por meio de adição do novos módulos criados por um programador. O principal objetivo é reforcar o uso dos *standards* definidos pela empresa consoante o código desenvolvido. *Checkstyle* trabalha com código fonte.

3.4.10 FindBugs

FindBugs [23] é uma ferramenta de análise estática que analisa as classes ou arquivos *JAR* procurando possíveis problemas combinando *bytecode* com a lista de padrões de bugs predefinidos. As ferramentas de análise estática permitem analisar *software* sem executar a aplicação. A forma e a estrutura das classes são analisadas para determinar a intenção da aplicação, muitas vezes usando o *pattern* visitante.

3.4.11 Testes de longa duração e testes de stress

É um conjunto de *scripts* que permitem automatizar testes de longa duração [24] e testes de stress. Estes testes são representados por *scripts* criados pelos programadores e pela equipa de qualidade com o objetivo de analisar o comportamento do software produzido durante longos períodos do tempo e com diferentes níveis de carga. A principal funcionalidade destes scripts consiste em:

- Preparar os ambientes remotos para execução dos testes.
- Recolher e analisar informação dos testes.

Os testes de longa duração e testes de stress são executados em sistemas operativos *Windows* e *Linux*.

3.4.12 Phabricator

Phabricator [25] é uma ferramenta *open source* que serve para a revisão de código [26]. Esta ajuda a melhorar a qualidade do código produzido analisando se o código corresponde às convenções definidas, fornece a visibilidade do produto que cada programador fornece, e ajuda a encontrar problemas nas revisões. O processo da revisão do código é o seguinte:

1. Na mensagem de submissão do código é feita uma descrição das alterações efetuadas.
2. As alterações são enviadas para o *Phabricator*.
3. As alterações são discutidas entres diferentes programadores.
4. Quando as alterações são aprovadas, estas são enviadas para o repositório remoto (GIT).

3.4.13 Selenium

Selenium é um *framework* para a automatização de testes de aplicações para *browsers* em diferentes plataformas, servindo para executar testes funcionais e testes de regressão. Como o Selenium foi integrado e está a ser utilizado na FeedZai, será descrito no próximo capítulo.

3.5 Conclusão

O processo de qualidade deve ter sempre uma melhoria contínua. Atualmente, a FeedZai tem o processo da qualidade bem formado, mas necessita de novos tipos de testes e de novas métricas. No próximo capítulo é descrito o trabalho realizado no âmbito deste estágio.

Capítulo 4

Testes Funcionais Automáticos

Este capítulo descreve o trabalho realizado e os resultados obtidos ao nível de automatização de testes funcionais para a interface gráfica (*web*) do *Pulse*.

4.1 Testes automáticos do PulseViews

Todo o processo de teste sobre o *frontend* do *Pulse* é feito com recursos à ferramenta *Selenium*. A análise das ferramentas existentes e a sua avaliação podem ser encontradas no **Anexo A** (*Testes funcionais e testes de regressão*). A FeedZai utiliza duas ferramentas para a realização dos testes funcionais e de regressão do PulseViews – *Selenium RC* e *Selenium Grid* [27], referidos anteriormente no ponto *3.4.12*. Os testes são desenvolvidos pela equipa de Garantia de Qualidade com base nas funcionalidades desenvolvidas (requisitos) e problemas encontrados (*bugs*). O objetivo é executar testes diários em diferentes browsers e diferentes sistemas operativos, de forma automática, integrando o *Selenium* na plataforma de integração contínua (*Jenkins*), referida no ponto *3.4.3*. O *Selenium* possui uma arquitectura de sistema própria que será usada na execução dos testes funcionais e de regressão.

4.1.1 Arquitetura

Para entender o funcionamento do *Selenium Grid* é necessário perceber como o *Selenium* funciona. Fundamentalmente, o *Selenium* é uma ferramenta que permite lança o *browser* programaticamente, manipulá-lo (abrir *urls*, inserir dados nos campos, clicar em *links*) e verificar o seu estado (por exemplo se

uma *widget* está ativa ou um texto está presente na página). A componente mais importante do *Selenium*, que controla o *browser* por meio do envio de pedidos *HTTP* através do protocolo específico *Selenese*, é chamada *Selenium RC*.

Na Figura 4.1 é demonstrada a configuração tradicional do *Selenium* para o caso da FeedZai. Esta configuração funciona muito bem para alguns testes, mas com o aumento do número de *RCs*, as limitações tornam-se mais claras:

- O *Selenium RC* torna-se muito lento na manipulação do *browser* (no caso de haver vários *RCs* no mesmo computador).
- O número de testes que se podem executar no mesmo *RC* sem afetar a sua estabilidade é limitado. Isto deve-se ao aumento da carga, e por sua vez da latência, o que leva a que aplicações *web* assíncronas não respondam como esperado.
- Os testes podem ser executados em vários *Selenium RCs* para contornar a limitação do número de testes que podem ser executados simultaneamente, mas isto não é escalável.

Devido a todas estas limitações, os testes do *Selenium* correm sequencialmente e, nalguns casos, podem ser executados em paralelo numa só máquina, o que faz com que passam demorar várias horas. Isto não é conveniente quando se pretende obter resultados rapidamente.

Selenium Grid

O principal objetivo do *Selenium Grid* é tornar possível a execução dos testes remotamente e paralelamente em várias máquinas. Por outras palavras, esta ferramenta permite executar por exemplo 10 testes diferentes ao mesmo tempo em diferentes máquinas em vez de executar estes 10 testes sequencialmente numa só máquina.

O *Selenium Grid* baseia-se na configuração tradicional do *Selenium* e tem as seguintes vantagens:

- A aplicação a testar e o *RC* não precisam de ser alocados na mesma máquina.
- O *Selenium RC* não precisa de ser associado aos testes de uma aplicação específica, mas pode ser partilhado por várias aplicações e projetos.

Para resolver o problema da escalabilidade anteriormente referido, é necessário haver uma componente capaz de executar os testes em paralelo. Esta componente é responsável por:

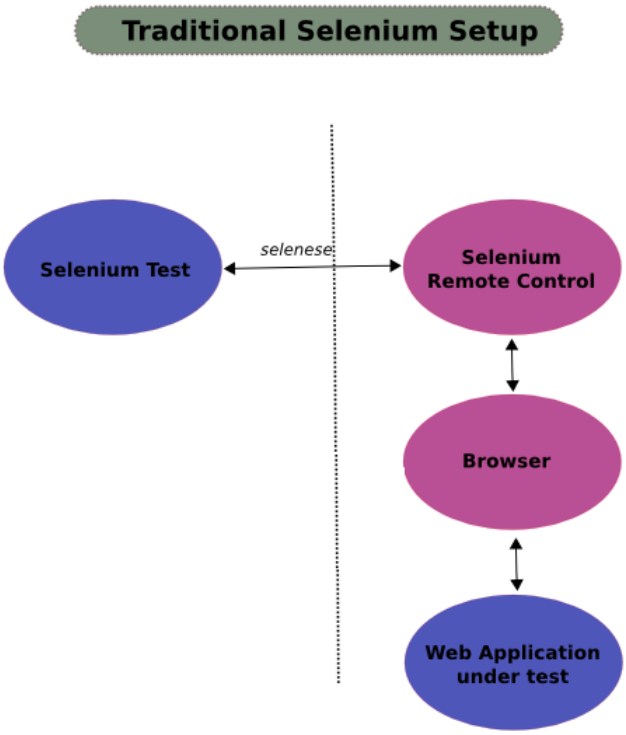

Figura 4.1: Configuração tradicional para o caso da FeedZai.

- Atribuir de modo transparente um *RC* a um teste específico
- Limitar a execução simultânea de testes num *Selenium RC*

No *Selenium Grid* esta componente chama-se *Selenium HUB*:

- O *Selenium HUB* é uma interface externa exatamente igual ao *RC* tradicional, o que significa que um conjunto de testes pode ter por destino um *RC* ou um *Selenium Hub* sem alteração do código. O *Selenium HUB* só tem que ter um endereço *IP* diferente dos outros *RCs*.
- O *HUB* aloca um *RC* a cada teste e é responsável por reencaminhar os pedidos *Selenese* para cada *RC*, bem como pela manutenção do controlo das sessões de teste.
- Quando recebe um novo teste, o *HUB* coloca-o na lista de espera até haver um *RC* disponível com as configurações adequadas. Assim que um *RC* adequado se tornar disponível, o *HUB* retira o teste da lista de espera e envia-o para esse *RC*.

Na Figura 4.2 é apresentada a configuração tradicional do *Selenium Grid* para o caso da FeedZai.

Solicitação de um ambiente específico

No início da sessão, cada teste especifica o tipo de *browser* e o sistema operativo onde tem de ser executado. Esta especificação é feita através de uma cadeia de caracteres pré-definida, tal como *"chrome"* e *"firefox"* ou *"Linux"* e *"Windows"*. O *Selenium HUB* garante que um teste é executado apenas no controlo remoto *Selenium Controls*, proporcionando o ambiente solicitado. Cada *RC* é responsável por se registar no *Selenium HUB* com as configurações definidas pela equipa de Garantia de Qualidade.

Na Figura 4.3 é demonstrado o modo como é feito o registo de cada *RC* no *HUB* e o reencaminhamento de um teste para o ambiente correto.

Arquitetura dos testes (1ª versão)

Cada teste do *Selenium* deve possuir as seguintes características:

- Ser isolado.
- Ser determinístico.
- Eliminar todo o estado gerado.

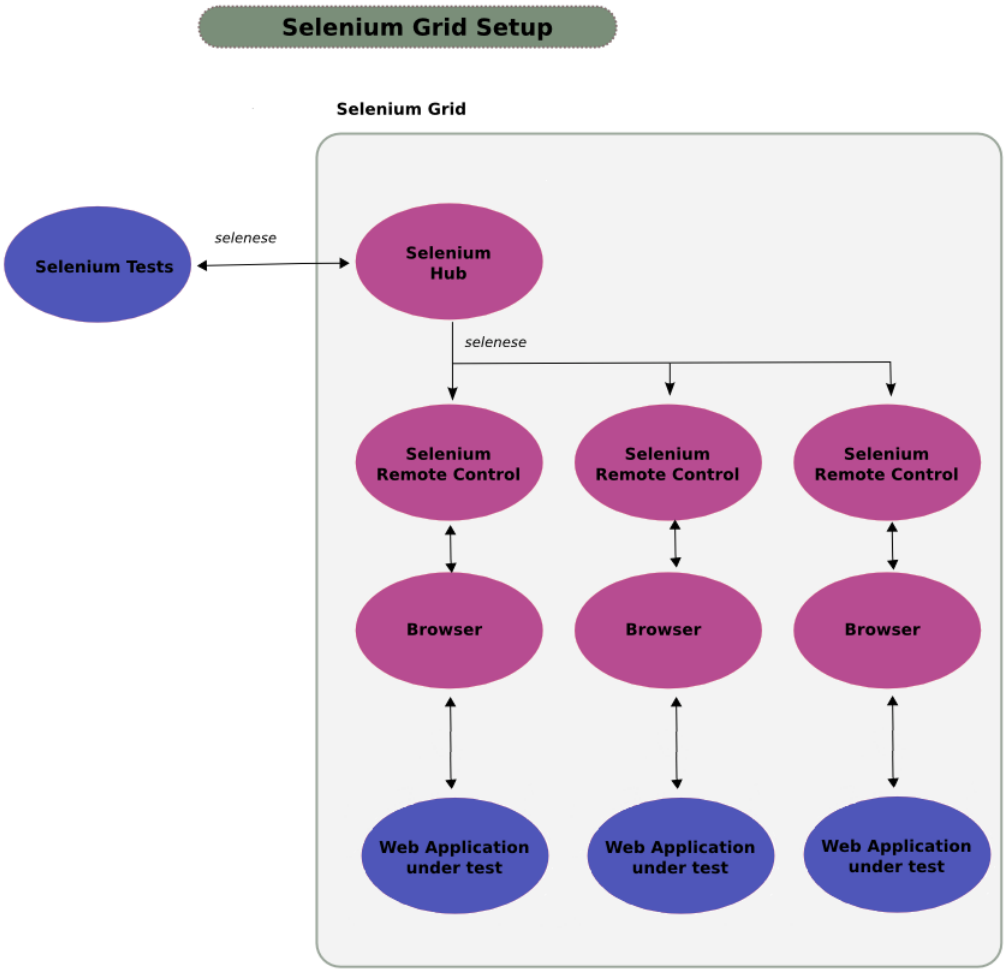

Figura 4.2: Configuração do Selenium Grid para o caso da FeedZai.

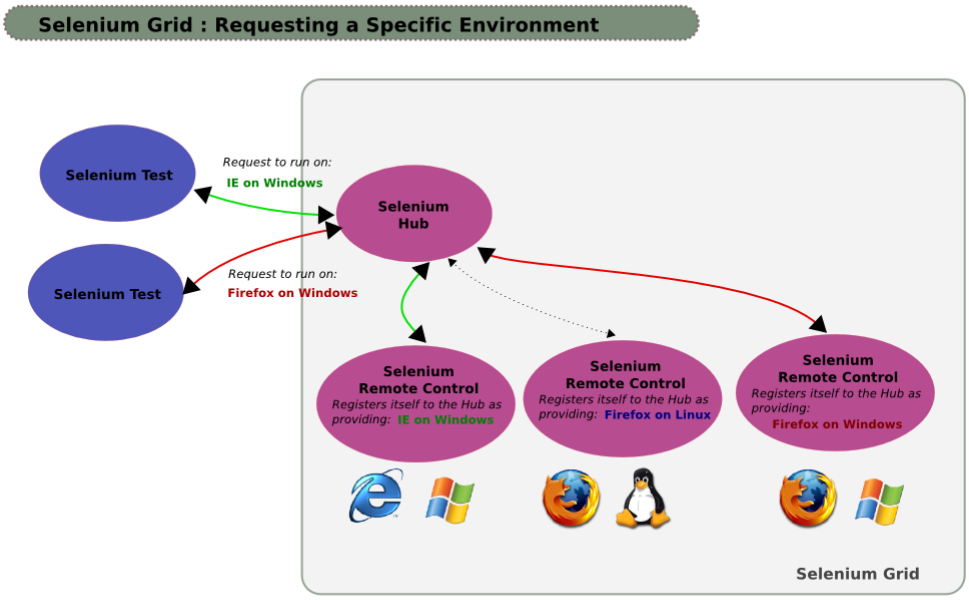

Figura 4.3: Solicitação de ambiente especifico no caso da FeedZai.

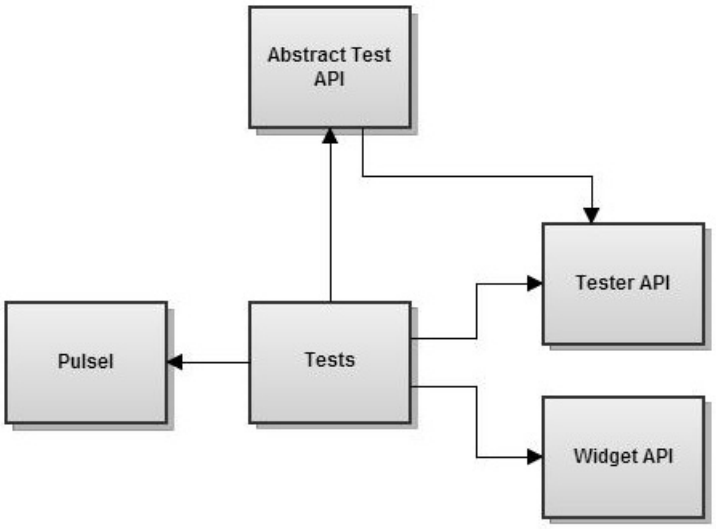

Figura 4.4: Arqutetura dos testes.

No Figura 4.4 é ilustrado o modo como os testes do *Selenium* foram organizados, para automatizar os testes sobre o *PulseViews*.

- **Tests** - módulo que contém um conjunto de classes divididas por funcionalidades. Cada classe contém um conjunto de casos de testes que testam um determinado ecrã.
- **Abstract Test API** - contém métodos pré-definidos para facilitar a criação dos casos de testes.
- **Tester API** - módulo que contém um conjunto de classes que disponibilizam a *API* para testar um determinado ecrã.
- **Widget API** - módulo que contém um conjunto de classes que disponibilizam a *API* para testar uma determinada widget.
- **Pulsel** - módulo responsável por iniciar/finalizar o *RC* e executar comandos *JQuery*.

Todos os testes são divididos em três categorias:

- ***ManagementTest** - classes que testam diferentes ecrãs (por exemplo, *Stream*, *KPI*, *Baseline*, etc)
- ***BuilderTest** - classes que testam diferentes widgets (por exemplo, *Plot Widget*, *Bullet Widget*, *Gauge Widget*, etc)

- ***RenderTest** - classes que testam a consistência dos dados apresentados nos gráficos.

Arquitetura dos testes (2ª versão)

Com o desenvolvimento de novas funcionalidade tornou-se claro que *Pulsel* e *Tester API* devem ser divididos com o objetivo de facilitar o uso do *API*. Na figura 4.5 é demonstrada esta divisão. De seguida só serão descritos os novos módulos.

- **Pulsel** - permite obter a instância do *tester*.
- **PulselTester** - contém todos os métodos comuns usado por todos testes.
- **PulselJavascriptChecker** - responsável por verificar os erros do *javascript* durante a execução de cada teste.
- **TesterCRUD** - permite realizar operações *create, read, update, delete* para cada funcionalidade.
- **ErrorTester** - permite verificar as mensagens de erro no ecrãs.
- **TesterBaseView** - permite dividir o ecrã por *views*.

Categorização dos testes

Como foi referido anteriormente, a FeedZai utiliza o *Selenium* para a execução dos testes funcionais e de testes de regressão sobre a interface gráfica. Uma vez que o *PulseViews* é altamente configurável e muito flexível, surgiu a necessidade de realização de testes de integração rápidos. Por exemplo, quando o programador faz alguma alteração numa das componentes do *PulseViews*, ele quer garantir que esta alteração não afeta o funcionamento das outras componentes. Dada esta necessidade, foram criados vários *tags* que ajudam os programadores a realizar testes rápidos usando o *Selenium* no próprio computador, escolhendo um dos perfis do *Maven* descritos na Tabela 4.1.

4.1.2 Configuração do ambiente

Antes de executar os testes, o ambiente tem de estar corretamente configurado. A configuração das máquinas remotas encontra-se especificada no ficheiro *POM* utilizado pelo *Maven*, que contém vários perfis para cada tipo de configuração. Para configurar as máquinas remotas são usados dois plugins para *Maven*:

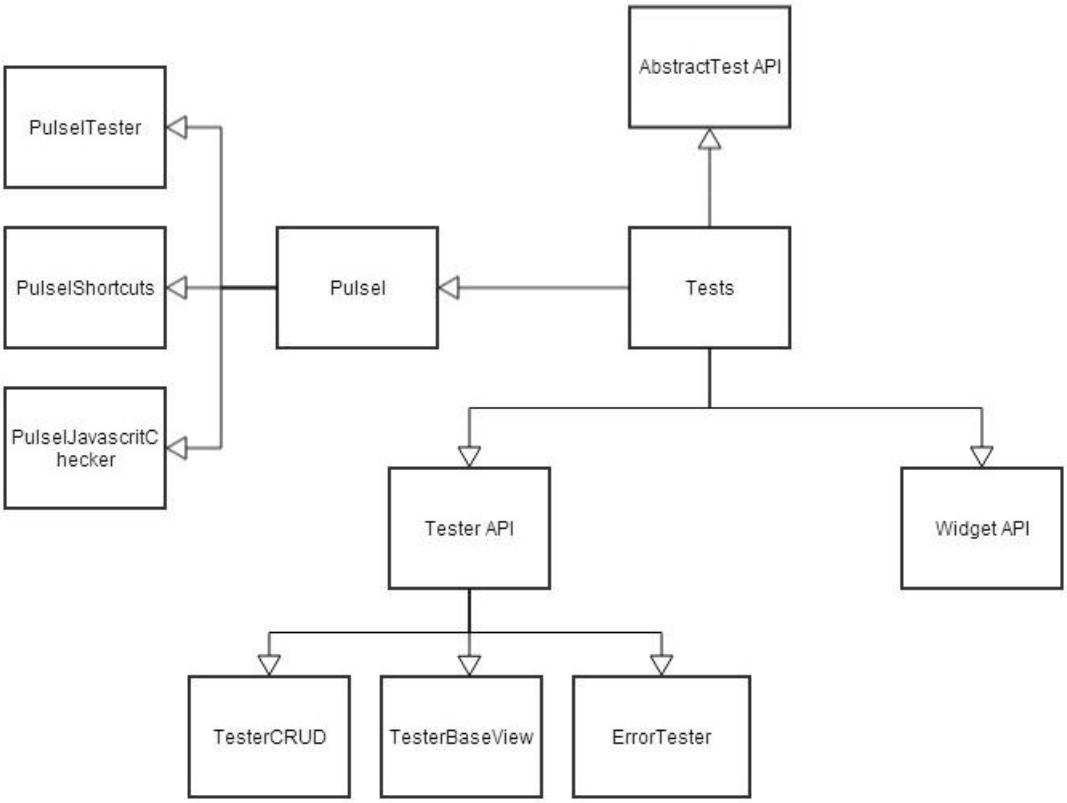

Figura 4.5: Arqutetura dos testes (2ª versão).

ID	Descrição
`admin`	Executa os testes no ecrã do administrador (cria vários utilizadores, diferentes permissões, etc).
`base`	Executa os testes de importação e exportação de aplicações, *CRUD* das aplicações e navegação em todas as *tabs* existentes no *PulseViews*.
`stream`	Executa testes *CRUD* de streams, mapeamento e filtro de dimensões e hierarquias.
`hierarchy`	Executa testes *CRUD* de hierarquias, kpi e alertas.
`kpi`	Executa testes *CRUD* de kpi, alertas.
`alert`	Executa testes *CRUD* de alertas.
`datasource`	Executa testes *CRUD* de conexões a vários tipos de base de dados e fonte de dados.
`input_adapter`	Executa testes *CRUD* de adaptadores (base de dados, ficheiros, sockets e binários).
`plot`	Configura *widget plot* e executa *CRUD* testes de mapeamento de dimensões, baselines e kpi.
`bullet`	Configura *widget bullet* e executa *CRUD* testes de mapeamento de dimensões, baselines e kpi.
`single_value`	Configura *widget single value* e executa *CRUD* testes de mapeamento de dimensões, baselines e kpi.
`map`	Configura *widget map* e executa *CRUD* testes de mapeamento de dimensões, baselines e kpi.
`tree`	Configura *widget tree* e executa *CRUD* testes de hierarquias.
`alert_icon`	Configura *alert icon* e executa *CRUD* testes de alertas.
`alert_log`	Configura *alert log* e executa *CRUD* testes de alertas.
`gauge`	Configura *widget gauge* e executa *CRUD* testes de kpis.
`text`	Configura *widget text*.
`list`	Configura *widget list* e executa *CRUD* testes de fonte de dados.
`date`	Configura *widget date*.
`button`	Configura *widget button*.
`image`	Configura *widget image*.
`table`	Configura *widget table* e executa *CRUD* testes de fonte de dados.
`tab`	Configura *widget tab*.
`banking`	Cria aplicação *banking* - uma aplicação de demonstração da FeedZai.

Tabela 4.1: Perfis do Maven para execução dos testes de integração.

- *copy-maven-plugin* - permite copiar vários ficheiros para máquinas remotas
- *sshexec-maven-plugin* - permite a execução de comandos *bash* em maquinas remotas

4.1.3 Maven plugin para o Selenium

O uso dos *plugins* referidos na secção anterior não é muito conveniente pelas seguintes razões:

- O ficheiro de configuração torna-se muito complexo.
- A adição de novas máquinas ao *Selenium Grid* torna-se muito complicada, uma vez que estas têm que ser inseridas em diferentes perfis do *Maven*.
- Falta do controlo do estado das máquinas. Caso não haja máquinas disponíveis, os testes não devem ser executados.

Devido aos problemas acima referidos, foi implementado pelo antes um *plugin* para o *Maven*, capaz de tornar a configuração das máquinas remotas mais abstrata. O resumo dos requisitos globais é descrito a seguir, sendo a sua análise detalhada apresentada no **Anexo B**.

- O *Plugin* deve permitir a configuração do *Selenium HUB* antes de configurar as máquinas de teste. As máquinas de testes não devem ser configuradas e os testes não devem ser executados caso não seja possível configurar o *Selenium HUB*.
- Deve existir suporte para a configuração de máquinas remotas com sistemas operativos *Windows* e *Linux*
- Só podem ser configuradas simultaneamente máquinas do mesmo sistema operativo
- O *Plugin* deve permitir de criar/remover ficheiros/directorias nas máquinas remotas
- O *Plugin* deve ser capaz de compactar os ficheiros num arquivo zip na maquina local.
- O *Plugin* deve ser capaz de descompactar os ficheiros dum arquivo zip nas máquinas remotas.

- O *Plugin* deve ser capaz de copiar ficheiros da máquina local para as máquinas remotas.
- O *Plugin* deve permitir executar comandos bash nas máquinas remotas.
- O *Plugin* deve permitir a configuração do *Selenium RC* nas máquinas remotas com um só comando.

Uma possível estrutura da configuração das máquinas remotas pode ser visualizada na Figura 4.6.

4.1.4 Verificação dos dados em *dashboards*

O *PulseViews* apresenta os dados por meio de *dashboards*. Os testes funcionais não permitem verificar se os valores dos gráficos estão corretos, uma vez que os *dashboards* são apresentados em *canvas*. A apresentação em *canvas* tem a grande vantagem de não permitir o acesso aos valores representados nos gráficos.

Usando o *Selenium* foi possível criar um conjunto de testes para verificar os valores dos gráficos. Estes testes geram os *dashboards* com diferentes configurações, por exemplo, o eixo dos Y pode aparecer do lado esquerdo bem como do lado direito, e a imagem do gráfico é capturada e guardada numa pasta externa. Na primeira execução dos testes os valores dos gráficos foram verificados manualmente com o objetivo de garantir que estão corretos. Nas execuções posteriores a verificação dos valores é feita por comparação das imagens capturadas com as tiradas anteriormente. A comparação das imagens é feita através da comparação de *byte arrays*. As imagens dos gráficos serão as mesmas, uma vez que os dados de entrada também o são.

4.2 Conclusão

Durante o estágio foram desenvolvidos 683 testes funcionais/de regressão para o *PulseViews* pela equipa da Garantia da Qualidade, usando o *Selenium*. Na Tabela 4.2 estão apresentados os testes para cada um dos componentes do *PulseViews*.

O projeto *Selenium* foi integrado com o *Jenkins* com o objetivo de executar os testes diariamente. Todos os problemas relacionados com a instabilidade dos testes foram resolvidos. Os problemas mais críticos encontrados foram os seguintes:

```
<configuration>
    <linux>
        <path>...</path>
        <username>...</username>
        <password>...</password>
        <nodes>
            ...
        </nodes>
        <testBehaviour>
            <copyFiles>
                ...
            </copyFiles>
            <commands>
                ...
            </commands>
            <nodeLauncher>...</nodeLauncher>
        </testBehaviour>
    </linux>
    <windows>
        <path>...</path>
        <username>...</username>
        <password>...</password>
        <nodes>
            <node>...</node>
            <node>...</node>
            <node>...</node>
        </nodes>
        </testBehaviour>
            <copyFiles>
                ...
            </copyFiles>
            <commands>
                ...
            </commands>
            <nodeLauncher>...</nodeLauncher>
        </testBehaviour>
    </windows>
</configuration>
```

Figura 4.6: Possível configuração das maquinas remotas.

- Os testes falhavam devido aos pedidos assíncronos, que são feitos por AJAX. Este problema foi resolvido verificando o número de pedidos pendentes antes de executar o próximo comando, através de execução do comando *return $.active == 0*. O próximo comando não poderá ser executado enquanto existirem pedidos pendentes.
- Os testes falhavam devido às limitações do servidor *Pulse*, em que aparece a janela de exceção, aquando occore um erro inesperado durante a execução dos testes. Caso o projeto *Selenium* não consiga executar qualquer comando, este verifica se a janela de exceção está presente e a exceção é guardada nos *logs* posterior análise da origem do problema.
- Os testes falhavam devido ao comportamento incorreto da funcionalidade do *PulseViews*. Neste caso o *screenshot* é capturado e é criado uma *issue* na plataforma *Jira*.

Administração	
Admin	1
Security Filter	7
Security Permissions	4
Security Projections	7
Security Roles	9
Security Users	16
Security Services	4
Ecrãs	
Alert	45
Baseline	45
Connection	36
Dashboard	45
Data source	41
Dimension Mapping	27
Hierarchy	24
Input adapter (binary)	8
Input adapter (file)	16
Input adapter (Database)	73
Input adapter (Socket)	10
KPI	43
Menus	19
Notifications	14
Stream	38
Widgets	
Alert icon	7
Alert log	7
Bullet	13
Button	4
Date	7
Gauge	14
Image	9
Label	6
List	6
Map	15
Plot	26
Single Value	10
Tab	13
Table	9
Tree	5

Tabela 4.2: Número de testes para cada componente de PulseViews.

Capítulo 5

Melhorias do Processo de Qualidade

Esta seção descreve as melhorias efetuadas ao nível do processo de qualidade, especialmente em ferramentas de *backend*. A criação de suporte para testes de mutação para a linguagem de programação *Scala*, bem como o suporte para o cálculo da cobertura para esta linguagem foram as melhorias efetuadas.

5.1 Testes de mutação

Para criar testes de mutação foi escolhida a ferramenta *PIT*. A análise detalhada das ferramentas existentes, bem como a razão da escolha da ferramenta em causa pode ser encontrada no **Anexo C**.

Atualmente *PIT* suporta dez tipos diferentes de mutações. Destes, sete são ativados por *default* e podem ser reconfigurados ou desativados. As mutações são feitas na base de código compilado e não na base dos ficheiros de código fonte. Esta abordagem tem a vantagem de permitir uma análise muito mais rápida dos ficheiros compilados, bem como a injeção das mutações nestes últimos.

O *PIT* permite gerar mutações consoante os testes unitários existentes. Por exemplo, o *PIT* analisa cada teste para identificar as zonas do código executadas a fim de gerar mutações exatamente nestas zonas.

Condição original	Condição mutante
<	<=
<=	<
>	>=
>=	>

Tabela 5.1: Conditional boundary mutator.

```
public void test(int a, int b)
{
    //por exemplo
    if(a > b)
    {
        //qualquer coisa
    }
    //será mutado para
    if(a >= b)
    {
        //qualquer coisa
    }
}
```

Figura 5.1: Exemplo de mutação do operador relacional.

5.1.1 Tipos de mutações

Segue abaixo uma breve descrição com exemplos concretos das mutações suportadas por *PIT*.

5.1.1.1 Conditional Boundaries

A mutação *conditional boundaries* substitui os operadores relacionais $<$, $<=$, $>$, $>=$ por outros operadores conforme se mostra na Tabela 5.1.

Um exemplo desta mutação é apresentado na Figura 5.1.

Condição original	Condição mutante
==	!=
!=	==
<=	>
>=	<
<	>=
>	<=

Tabela 5.2: Negate Conditionals Mutator.

```
public void test(int a, int b)
{
    //por exemplo
    if(a == b)
    {
        //qualquer coisa
    }
    //será mutado para
    if(a != b)
    {
        //qualquer coisa
    }
}
```

Figura 5.2: Exemplo da condição negativa.

5.1.1.2 Negate Conditionals Mutator

A mutação *Negate Conditionals Mutator* permite substituir diferentes condições por outras contrárias às mesmas, de acordo com a tabela 5.2.

Um exemplo da mutação *negate conditionals mutator* é apresentado na Figura 5.2.

5.1.1.3 Mutação matemática

A mutação matemática substitui cada operação aritmética por outra de acordo com a tabela 5.3.

Um exemplo da mutação matemática é representado na figura 5.3.

Condição original	Condição mutante
+	-
-	+
*	/
/	*
&	|
|	&
<<	>>
>>	<<
>>>	<<

Tabela 5.3: Mutação matemática.

```
public void test(int a, int b, int c)
{
    //por exemplo
    a = b + c;

    //será mutado para
    a = b - c;
}
```

Figura 5.3: Exemplo da mutação matemática.

```
public int test(int i)
{
    i++;
    return i;
}

//será mutado para
public int test(int i)
{
    i--;
    return i;
}
```

Figura 5.4: Exemplo de *increment mutator*.

```
public int test(int i)
{
    return i;
}

//será mutado para
public int test(int i)
{
    return -i;
}
```

Figura 5.5: Exemplo de *invert negatives mutator*.

5.1.1.4 Increments Mutator

A mutação de incrementação e decrementação é apenas aplicada às variáveis locais. As incrementações são substituídas por decrementações, e vice-versa. Na Figura 5.4 é demonstrado um exemplo deste tipo de mutação.

5.1.1.5 Invert Negatives Mutator

A mutação *Invert Negatives Mutator* inverte o sentido dos números inteiros e números de ponto flutuante como é apresentado na Figura 5.5.

```
public int test()
{
    int i = 42;
    return i;
}

//será mudado para
public int test()
{
    int i = 43;
    return i;
}
```

Figura 5.6: Exemplo de *inline constant mutator*.

Tipo do constante	Mutação height
boolean	substitui o valor *true* por *false* e vice versa.
integer, byte short	incrementa o valor, por exemplo *1* será substituído por *2*.
long	incrementa o valor, por exemplo *6* será substituído por *7*.
float	incrementa o valor, por exemplo *2.0* será substituído por *3.0*.
double	incrementa o valor, por exemplo *-2.0* será substituído por *-1.0*.

Tabela 5.4: Alteração do valor (*Inline constant*).

5.1.1.6 Inline Constant Mutator

Mutação *Inline Constant Mutator* permite alterar o valor atribuído a uma variável local, como se pode observar na Figura 5.6.

Inline Constant Mutator são efetuadas de acordo com a Tabela 5.4.

5.1.1.7 Return Values Mutator

Return Values Mutator permite alterar o valor do retorno do método consoante o tipo de retorno. O processo desta alteração consta na Tabela 5.5.

Na Figura 5.7 é mostrado um exemplo desta mutação.

Tipo do constante	Mutação
boolean	substitui o valor *true* por *false* e vice versa.
integer, byte short	altera o valor *0* por *1* e vice versa. Incrementa o valor por 1 caso este seja diferente do 0 ou 1
long	altera o valor x para $x+1$.
float, double	altera o valor x para $x+1$. Caso o valor seja *NAN* altera para *0* e vice versa.
object reference	substitui *non null* por *null* e vice versa.

Tabela 5.5: *Return value mutator*.

```
public Object test(int i)
{
    return new Object();
}

//será mutado para
public Object test(int i)
{
    return null;
}
```

Figura 5.7: Exemplo de *return value mutator*.

```java
public void method(int i)
{
    //qualquer operação com variável i
}
public int test(int i)
{
    int i = 10;
    method(i);
    return i;
}

//será substituido por...
public void method(int i)
{
    //qualquer operação com variável i
}
public int test(int i)
{
    int i = 10;
    return i;
}
```

Figura 5.8: Exemplo de *void method call mutator*.

5.1.1.8 Void Method Call Mutator

Void Method Call Mutator permite eliminar a invocação de métodos *void*, como é exemplificado na figura 5.8.

5.1.1.9 Non Void Method Call Mutator

Non Void Method Call Mutator permite substituir a invocação de métodos que retornam valores ou objetos por outro valor ou objeto fixo como pode ser observado na Figura 5.9. Caso o valor de retorno do método seja um objeto, esse valor será substituído por *null*. Este tipo de mutação não é aplicado aos construtores.

```
public int method(int i)
{
    return 10;
}
public void test(int i)
{
    int i = method(i);
}

//será mutado para
public int method(int i)
{
    return 10;
}
public void test(int i)
{
    int i = 0;
}
```

Figura 5.9: Exemplo de *non void method call mutator*.

```
public Object method()
{
    Object o = new Object();
    return o;
}

//será mutado para
public Object method()
{
    Object o = null;
    return o;
}
```

Figura 5.10: Exemplo de *constructor call mutator*.

5.1.1.10 Constructor Call Mutator

A mutação *Constructor Call Mutator* faz com que as chamadas aos construtores sejam substituídas por *null*. Um exemplo concreto é apresentado na Figura 5.10

5.1.2 Conclusão

Nesta secção foram descritos vários tipos de mutações, que permitem alterar o código fonte com o objetivo de avaliar a qualidade dos testes unitários. Os testes de mutação assemelham-se a qualquer outro tipo de testes, tendo as suas vantagens e desvantagens. As vantagens são as seguintes:

- Permitem avaliar a qualidade dos testes unitários existentes.
- Permitem detetar falhas na aplicação.
- Permitem melhorar a qualidade do *software*.

Este tipo de testes tem uma grande desvantagem que consiste o elevado consumo de recursos e tempo. No caso da *FeedZai*, para executar os testes de mutação para todos os módulos e com todas as mutações possíveis é necessário uma semana. É possível reduzir o tempo da execução diminuindo o número de mutações para cada classe.

Depois de integrar a ferramenta *PIT* no projeto *Pulse* e realizar várias execuções dos testes de mutação concluiu-se que:

- Com uma cobertura maior as alterações introduzidas por mutações têm um efeito direto. Por exemplo, com uma cobertura de 20% serão detetadas menos mutações no código que com uma cobertura de 90%.
- As ferramentas atuais não permitem detetar todos os problemas dos testes unitários. Na Figura 5.11 é apresentado um exemplo de um teste unitário. Este teste tem um problema que não será detetado com testes de mutação. Um *Thread.sleep* não garante que o sistema processou o valor injetado e que o resultado está pronto para a leitura.
- Em algumas das vezes, as mutações criadas no código ajudam de facto a avaliar a qualidade do processo de realização dos testes unitários. Na Figura 5.12 é mostrado um exemplo da não verificação do parametro *NIF* pelo teste do objeto *Person*. Se a ferramenta introduzir mutações no método *getNIF*, estas não serão detetada por este teste. Por outro lado, se existir outro teste que verifique todos os campos do objeto *Person*, a mutação será detetada.

A execução dos testes de mutação é independente das execuções anteriores, pelo que os seus resultados não podem ser comparados. Disto decorre que o resultado de cada execução deve ser analisado separadamente, tendo em atenção que nem todas as mutações criadas serão necessariamente ser detetadas. Os testes de mutação permitiram detetar vários testes unitários que não efetuavam a verificação completa.

5.2 Modificação do projecto *Pulse*

Esta seção descreve as modificações efetuadas ao nível do *Pulse*, tais como a extração da cobertura do código, a adição do suporte para a linguagem de programação *Scala*, e a extração da cobertura do código para esta última.

5.2.1 Estrutura do projeto

O produto *Pulse* é um projeto *multi-module* em que são usadas diferentes linguagens de programação, tais como *Java*, *Javascript* e *Scala*. Como foi mencionado na secção 3.4.4, *FeedZai* utiliza o *Sonar* para extrair diferentes métricas de qualidade. O projeto *Pulse* é compilado e os testes unitários são

```
@Test
public void badTest()
{
    InputSocketAdapter issa = Instances.getInputSocketAdapter();
    int [] valuesToBeInjected = {1, 2, 3, 4, 5};
    //inject values into engine
    for(int i : valuesToBeInjected)
    {
        issa.injectValues(i);
        Thread.sleep(1000);
        assertEquals("Read value is correct?", i, issa,readVaules(i));
    }
}
```

Figura 5.11: Teste unitário com defeito.

executados na plataforma de integração contínua *Jenkins*. Na Figura 5.13 são apresentados os módulos principais do produto da *FeedZai*. Depois de compilação cada módulo é analisado pelo *Sonar*. O *Sonar* entende cada módulo como sendo um projeto independente e não consegue juntar todos módulos num só. Uma das principais desvantagens desta estrutura é não haver uma visão global do estado do projeto, mas sim, de cada módulo separadamente.

A estrutura geral da definição dos módulos pode ser vista na Figura 5.14. Para conseguir ter uma visão global do projeto é necessário compilar o projeto e executar os testes unitários a partir do módulo principal - *root*. Para tal, foi necessário mover todas as dependências usadas em cada módulo para o módulo principal.

Estas dependências consistem em:

- Configuração do *plugin PMD* referida no secção 3.4.8.
- Configuração do *plugin Findbugs* referida no secção 3.4.9. Os ficheiros de configuração que especificam as classes incluídas e excluídas foram movidos para a pasta *root* do projeto.
- Configuração do *plugin Checkstyle* referida na secção 3.4.10. Os ficheiros de configuração que especificam as classes incluídas e excluídas foram movidos para a pasta *root* do projeto.

Depois de efetuar estas alterações, na plataforma de integração contínua passa a existir um só projeto. Este projeto executa todos os módulos e

```
@Test
public void badTest()
{
    Person p = new Person(1, "Andrey", 2555555);
    assertEquals("Id is correct?", 1, p.getId());
    assertEquals("Name is correct?", "Andrey", p.getName());
}

class Person
{
    int id;
    String name;
    int nif;

    public Person(int id, String name, int nif)
    {
        this.id = id;
        this.name = name;
        this.nif = nif;
    }

    public int getNIF()
    {
        return this.nif;
    }
    public int getId()
    {
        return this.id;
    }
    public int getName()
    {
        return this.name;
    }
}
```

Figura 5.12: Teste sem todas as verificações.

Figura 5.13: Estrutura do projecto no *Jenkins*.

submódulos mencionados anteriormente. O *Sonar* ao analisar este novo projeto da plataforma de integração contínua, consegue juntar todos os módulos do projeto e agrupar os resultados de cada métrica extraídos de todos os módulos. Na Figura 5.15 é monstrada a nova estrutura do projeto no sistema *Sonar* onde o módulo principal *pulse-root trunk* contém todos os módulos do projecto.

5.2.2 Cobertura do código

Atualmente a *FeedZai* usa *Cobertura* [28] como ferramenta para calcular a cobertura do código dos testes unitários mas, entretanto, foi proposto utilizar uma outra ferramenta, chamada *JaCoCo*. As principais razões para uso de *JaCoCo* são as seguintes:

- Suporte para *Java 7*. A atualização para esta versão estava prevista no planeamento do produto *Pulse*.
- Instrumentação das classes é feita *on the fly* - uma das principais vantagens desta ferramenta. O uso deste modo de instrumentação será descrito na secção **JaCoCo Maven Plugin para *Scala***.

De facto, com a atualização da versão de Java 6 para Java 7, a *Cobertura* deixou de funcionar corretamente, uma vez que não suporta esta versão de *Java*.

A análise das ferramentas existentes e as conclusões para a escolha da ferramenta *JaCoCo* podem ser encontradas em **Anexo D**.

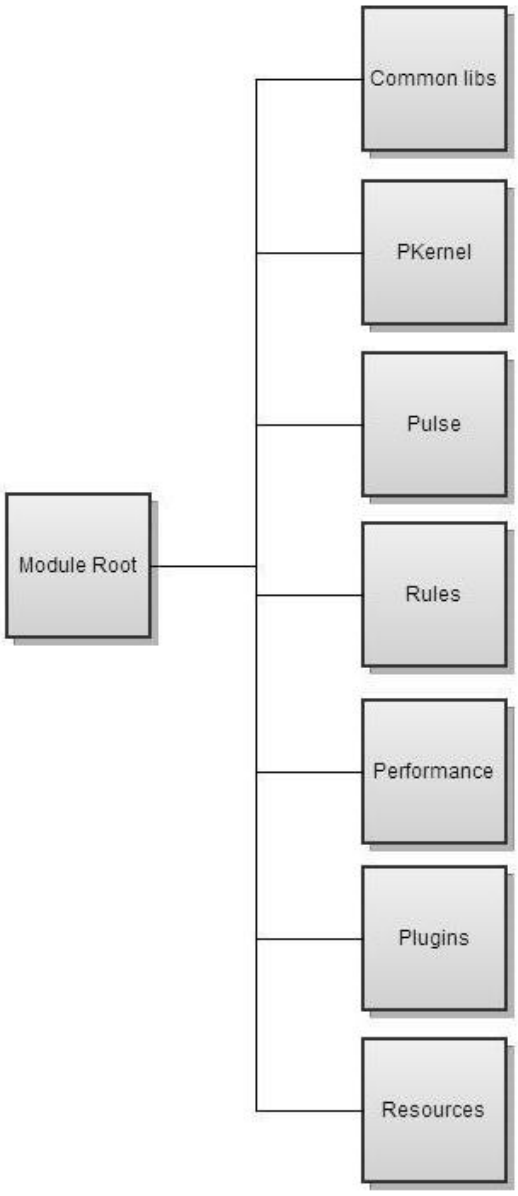

Figura 5.14: Estrutura do projecto.

Name	Lines of code	Coverage	Comments (%)	Public documented API (%)	Unit tests	Unit tests success (%)	Skipped unit tests	Unit tests errors	Unit tests failures	Pulse Hot Ev Rate
pulse-root trunk	147.769	31,6%	19,9%	70,0%	6.527	100,0%	27	0	0	

Name	Lines of code	Coverage	Comments (%)	Public documented API (%)	Unit tests	Unit tests success (%)	Skipped unit tests	Unit tests errors	Unit tests failures	Pulse Hot Ev Rate
common-libs trunk	12.029	58,2%	19,8%	95,2%	791	100,0%	6	0	0	
performance-tests trunk					0					
PKernel trunk	23.556	7,6%	13,1%	39,9%	3.345	100,0%	4	0	0	
plugins trunk	69	0,0%	11,5%	100,0%	0					
pulse trunk	93.822	29,6%	21,0%	77,0%	1.547	100,0%	17	0	0	
resources trunk					0					
rules-engine trunk	18.293	79,3%	22,4%	72,0%	844	100,0%	0	0	0	

Figura 5.15: Nova estrutura do projecto.

5.2.2.1 Integração do *JaCoCo*

O projeto *Pulse* é compilado usando o *Maven*, descrito na seção 3.4.1. Esta ferramenta também serve para fazer *deploy* dos artefatos para o *Nexus*, descrito na secção 3.4.2, e para executar os testes unitários usando o *Maven Surefire Plugin*. Este último *plugin* é utilizado na fase de testes.

Para injetar JaCoCo na execução dos testes usando *Maven Surefire Plugin* foi necessário:

1. Definir a dependência para conseguir obter o *plugin* do repositório remoto.
2. Instrumentar as classes antes de executar os testes e restaurá-las depois da execução.
3. Injetar o *JaCoCo java agent* em *JVM* segundo duas possibilidades:
   (a) Injetar o *JaCoCo java agent* como o argumento para o *plugin*, indicando o caminho físico para este.
   (b) Injetar *JaCoCo java agent* como propriedade do sistema. Este modo é mais correto uma vez que não há necessidade de man-

```
<dependencies>
    <dependency>
        <groupId>org.jacoco</groupId>
        <artifactId>org.jacoco.agent</artifactId>
        <classifier>runtime</classifier>
        <version>0.6.3-SNAPSHOT</version>
    </dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>0.6.3-SNAPSHOT</version>
            <executions>
                <execution>
                    <goals>
                        <goal>instrument</goal>
                        <goal>restore-instrumented-classes</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <configuration>
                <skipTests>false</skipTests>
                <systemPropertyVariables>
                    <jacoco-agent.destfile>jacoco.exec</jacoco-agent.destfile>
                </systemPropertyVariables>
            </configuration>
        </plugin>
```

Figura 5.16: Configuração do plugin *JaCoCo* no projecto *Pulse*.

ter o ficheiro com o *JaCoCo java agent* fisicamente no sistema operativo.

No final da execução, dos testes o *JaCoCo* gera um ficheiro *jacoco.exec* que contém a cobertura do código executado. A configuração atual do *JaCoCo* no ficheiro *POM* pode ser visualizada na figura 5.16.

5.2.3 Suporte para a linguagem *Scala*

Sonar é composto por vários *plugins* que permitem a extração das diferentes métricas. Por omissão o *Sonar* só suporta a linguagem *Java*, mas existe

suporte (desenvolvido pela equipa de desenvolvimento do *Sonar*) para a linguagem *Javascript*. A linguagem de programação *Scala* não é suportada oficialmente pelo *Sonar*, uma vez que os algoritmos para calcular as métricas não estão preparados para esta linguagem.

Em Julho de 2012, Felix Müller desenvolveu um *plugin* para a linguagem de programação *Scala*. Este *plugin* permite calcular as métricas *básicas*, tais como:

- Número de linhas de código físicas.
- Número total de linhas de código.
- Número de *statements*.
- Número de ficheiros.
- Número de classes.
- Número de *packages*.
- Número de métodos.
- Percentagem de comentários no código.
- Percentagem de comentários de métodos de *API*.
- Complexidade ciclomática.

Por defeito, este *plugin* permite analisar os módulos do *Maven* escritos apenas em *Scala*, e não suporta módulos escritos em mais do que uma linguagem de programação, como por exemplo *Java* e *Scala* - como é o caso do *Pulse*.

Tendo em conta esta situação, o *Scala plugin* foi alterado com o objetivo de suportar várias linguagens de programação num só módulo. Esta modificação não exigiu muito tempo, uma vez que a solução para este problema era simples. Assim, foi necessário analisar os ficheiros com extensão *.scala* e não verificar outros ficheiros.

Para poder analisar o projeto com a linguagem de programação *Scala*, na plataforma de integração contínua *Jenkins* é necessário indicar explicitamente o parâmetro *-Dsonar.language=scala* e também reconfigurar o caminho para as pastas com ficheiros de código fonte e os ficheiros dos testes. Na Figura 5.17 são apresentadas duas janelas. A primeira monstra as métricas calculadas para o modulo *PKernel* para a linguagem de programação *Scala*, e a segunda demonstra as métricas calculadas para o mesmo módulo para a linguagem de programação *Java*.

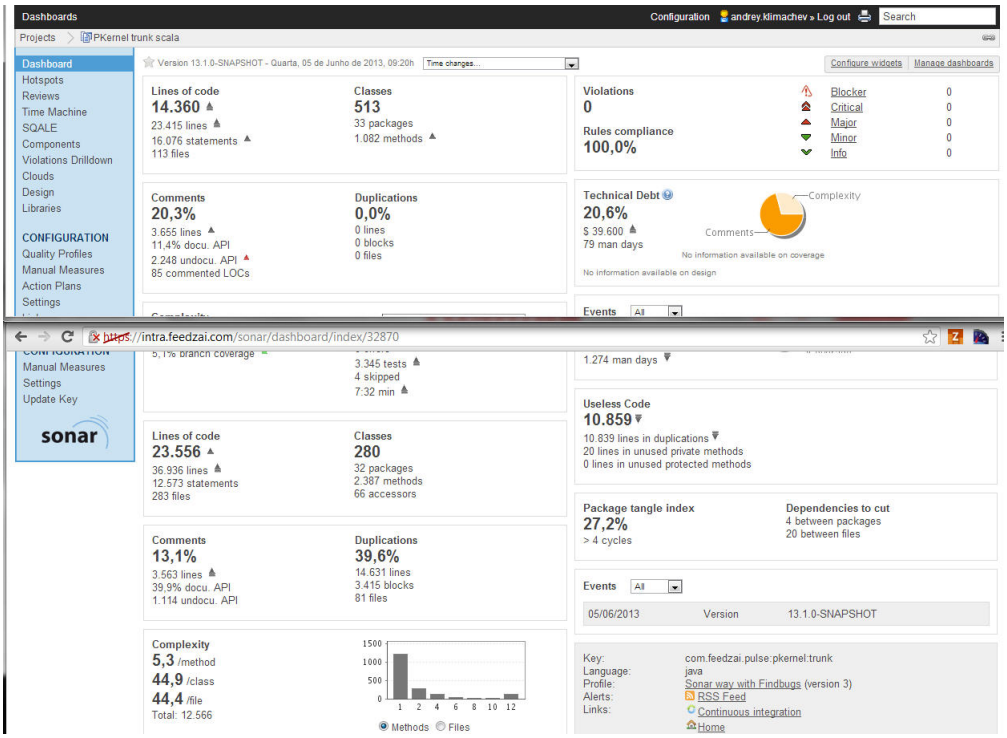

Figura 5.17: Demonstração da divisão dos projetos com diferentes linguagens de programação.

5.2.4 JaCoCo Maven Plugin para Scala

A *FeedZai* utiliza *Maven Surefire Plugin* para executar testes unitários de *Java* e *ScalaTest Maven Plugin* para executar testes unitários de *Scala*. A vantagem deste *plugin* é o facto dos testes poderem ser executados em *JVM*, o que permite injetar o agente *JaCoCo* para instrumentar as classes *on the fly*, o que por sua vez permite extraír a cobertura dos testes unitários.

Atualmente, o *PKernel*, é o único modulo (composto por vários submódulos) do produto da FeedZai escrito em duas duas linguagens de programação – *Java* e *Scala*. A estrutura do *PKernel* não é uma estrutura perfeita para extrair a cobertura dos testes unitários. Na Figura 5.18 são monstradas as dependências dos seus módulos. Como se pode verificar, o modulo *PKernel Driver* depende de três módulos - *PKernel API*, *PKernel Frontend* e *PKernel Backend*. Para calcular a cobertura dos testes do módulo *PKernel Driver* foi necessário copiar o código fonte dos módulos dependentes, para poder instrumentar as classes e incluí-las no relatório da cobertura.

Com estas manipulações nos módulos foi possível calcular a cobertura correta de cada módulo, mas não foi possível analisá-la com o *Sonar*, uma vez que o *Sonar plugin* suporta apenas as métricas básicas descritas na secção **Suporte para linguagem de programação *Scala***. Uma vez que a cobertura extraida não foi guardada, foi proposto desenvolver o *JaCoCo Scala Maven Plugin*. O objetivo deste *plugin* é guardar a informação detalhada de cada modulo numa base de dados externa. A extração da cobertura é feita segundo as seguinte fases:

- Executar os testes unitários de cada módulo do *PKernel* com a injeção do agente *JaCoCo*.
- Executar *JaCoCo Scala Maven Plugin* com as seguintes configurações:
    1. Indicar o ficheiro com os dados recolhidos da cobertura, que por defeito é *jacoco.xml*.
    2. Indicar a informação da base de dados (*url*, nome do utilizador e palavra passe) em que a informação irá ser guardada.
    3. Efetuar o *parsing* dos ficheiros com os resultados da cobertura.
- Apagar todos os ficheiros copiados de cada módulo. A razão para copiar os ficheiros foi indicada anteriormente.

Na Figura 5.19 pode ser visualizada a configuração base do *JaCoCo Scala Maven Plugin*.

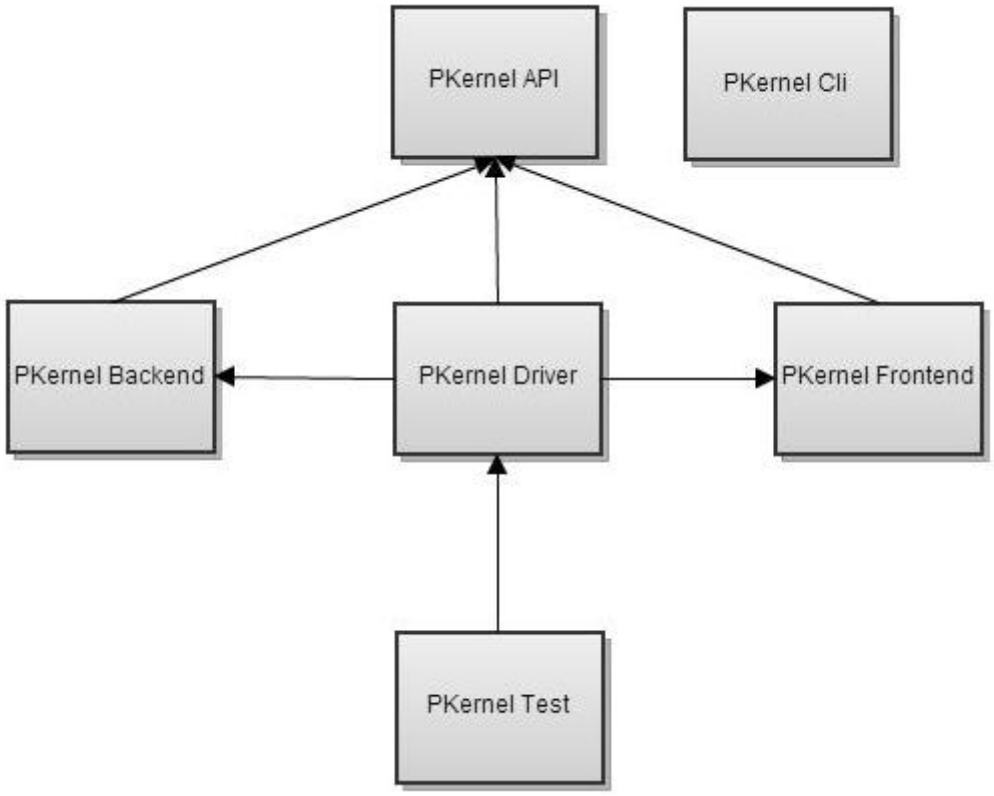

Figura 5.18: Dependência dos modulos do PKernel.

```
<plugin>
    <groupId>com.feedzai</groupId>
    <artifactId>jacoco-scala-maven-plugin</artifactId>
    <version>1.0.1</version>
    <configuration>
        <filename>jacoco.xml</filename>
        <jdbcDriverClassName>com.mysql.jdbc.Driver</jdbcDriverClassName>
        <jdbcUrl>jdbc:mysql://localhost.zai:3306/quality-reports</jdbcUrl>
        <jdbcUsername>root</jdbcUsername>
        <jdbcPassword>password</jdbcPassword>
    </configuration>
</plugin>
```

Figura 5.19: Configuração do *JaCoCo Scala Maven Plugin*.

5.2.5 Conclusão

O projeto *Pulse* foi modificado com o objetivo de continuar a ter a cobertura do código, usando a nova ferramenta *JaCoCo*. Para além da cobertura de código de *Java* foi possível analisar os módulos escritos em *Scala* e também se tornou possível calcular a cobertura dos módulos para esta linguagem.

5.3 Conclusão

As alterações efetuadas no projeto foram necessárias para melhorar o processo da qualidade. Além disso, a equipa de Garantia de Qualidade deve tentar prever os problemas. Por exemplo, foi previsto que a ferramenta *Cobertura* não suporta *Java 7*, e por isso foram realizadas as alterações no processo de medição da cobertura do código.

As alterações efetuadas no *Scala plugin* para o *Sonar* permitiram incluir a avaliação da qualidade do código produzido em *Scala*. O novo *JaCoCo Scala Maven Plugin* permitiu calcular a cobertura do código para esta linguagem de programação.

Todas as alterações efetuadas têm o mesmo objetivo – melhorar o processo da qualidade. Para além disso, estas alterações preparam a estrutura para visualizar a evolução de cada métrica do projecto *Pulse* ao longo do tempo usando o servidor da qualidade, que será descrito no próxima secção.

Capítulo 6

Servidor de Qualidade

Esta secção descreve os objetivos da implementação do servidor da qualidade, bem como a sua arquitetura, o modo de funcionamento e os relatórios de qualidade gerados.

6.1 Enquadramento

Até agora as alterações efetuadas ao nível do projecto da *FeedZai* são independentes mas têm o mesmo fim em vista – melhorar o processo da qualidade. O objetivo de implementação de um servidor de qualidade para a *FeedZai* foi complementar o servidor *Sonar*. Este último tem várias desvantagens:

- Não suporta várias métricas, como por exemplo, a cobertura do código para linguagem de programação *Scala*.
- Não permite visualizar a evolução contínua de cada métrica.
- Não permite comparar as métricas de diferentes versões do produto.
- Suporta somente as métricas ao nível do código fonte.
- Não é extensível para a adição das novas métricas.

Pelas razões acima indicadas, foi necessário criar um servidor de qualidade para a *FeedZai*. As principais funcionalidades deste são as seguintes:

- Extrair as métricas dos diferentes recursos - *Sonar*, *Jenkins*, *Git* e *Jira*.
- Mostrar a evolução ao longo do tempo de cada métrica.

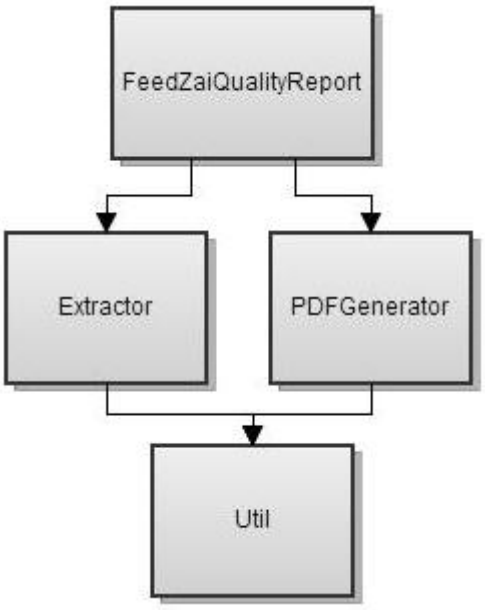

Figura 6.1: Arquitetura do servidor da qualidade em alto nível.

- Gerar diferentes tipos de relatórios de qualidade.
- Permitir a visualização das diferentes métricas em tempo real nos ecrãs nas salas de desenvolvimento, com o objetivo de visualizar a evolução da qualidade do produto.

6.2 Arquitetura

A arquitetura global do servidor composto, por duas componentes principais, é ilustrada na Figura 6.1.

- **Extractor**. Permite extrair valores de diferentes recursos, tais como *Sonar*, *Jira*, *Jenkins* e *Git*.
- **PDFGenerator**. Permite gerar dois diferentes tipos de relatórios - relatório completo e relatório curto.
- **Util**. É um conjunto dos métodos usados pelas componentes *Extractor* e *PDFGenerator*. Permite manusear as matrizes com as métricas.

O *Extrator* é o módulo principal do servidor e tem por objetivo extrair frequentemente diferentes métricas. A periodicidade da extração é definida

manualmente e depende da frequência com que os resultados são atualizados nas plataformas. Atualmente a periodicidade é de um dia. Este módulo pode ser visto em detalhe na Figura 6.2.

- **QRDB**. É a base de dados onde todas as métricas são guardadas.
- **DatabaseEngine**. Permite obter, atualizar e guardar as métricas.
- **Util**. Um conjunto dos métodos que permitem manipular as matrizes das métricas (ordenar, filtrar, etc).
- **GitMetric**. Representa a estrutura do objeto para a métrica do *Git*. Contém informação da hora do *commit* e autor.
- **SourceCode**. Representa a estrutura do objeto para guardar o código fonte.
- **GeneralMetric**. Representa a estrutura do objeto geral de uma métrica. Contém nome, descrição, tipo da métrica (inteiro, percentagem, *string*, etc) e linguagem de programação desta métrica.
- **Project**. Representa a estrutura do objeto geral do projecto em que cada componente é considerado como um projeto (*class, package e module*).
- **MetricValue**. Representa a relação entre os objetos **Project** e **GeneralMetric**.
- **Violation**. Representa uma violação no código fonte.
- **Sonar API**. É a *API* do *Sonar* a partir da qual algumas métricas da qualidade do código são obtidas.
- **Jenkins API**. É a *API* do *Jenkins* a partir da qual é obtida a informação do estado de cada projeto na plataforma de integração contínua.
- **Jira API**. É a *API* do *Jira* a partir da qual é obtida a informação dos *bugs*.
- **Git API**. É a *API* do *Git* a partir da qual é obtida a informação dos *commits* efetuados.
- **SonarExtractor**. Extrai a informação das métricas do *Sonar* através de *Sonar API*.
- **JiraExtractor**. Extrai a informação das métricas do *Jira* através do *JQL* (*Jira Query Language*).

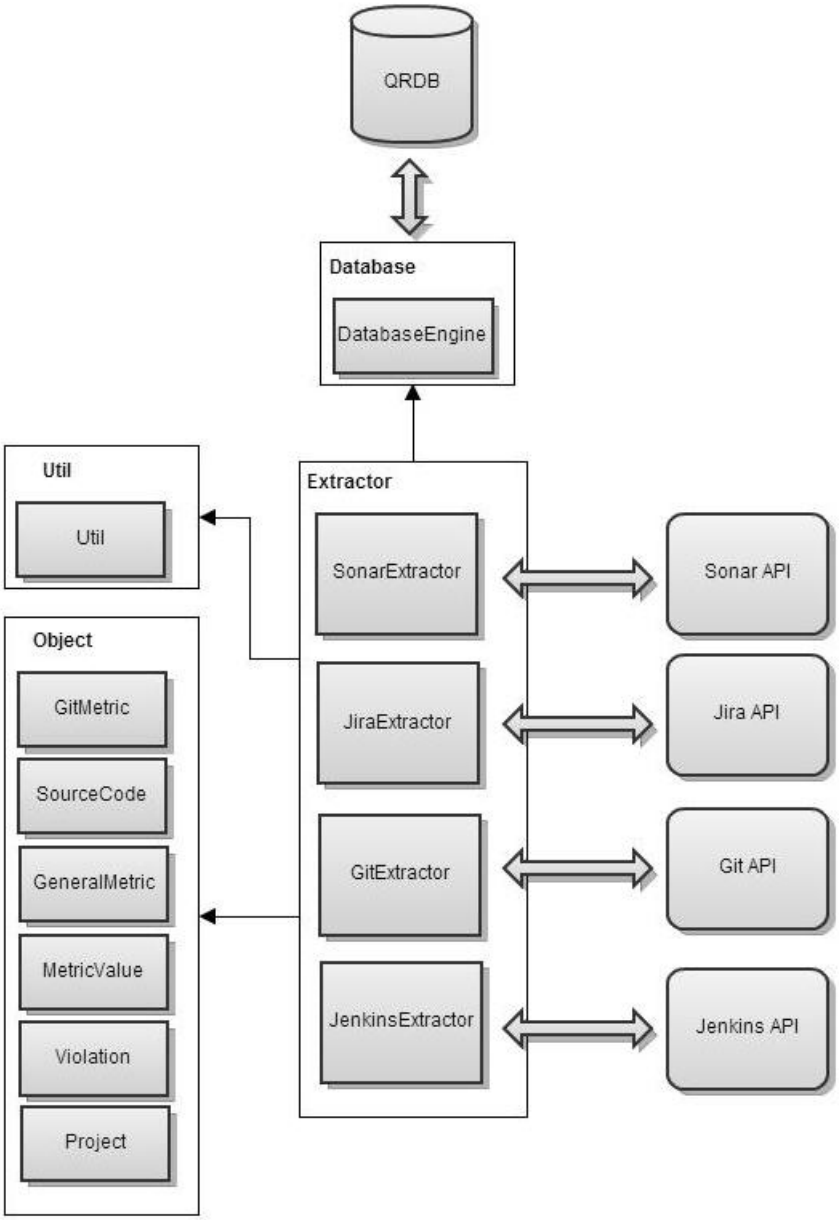

Figura 6.2: Arquitetura da extração das métricas.

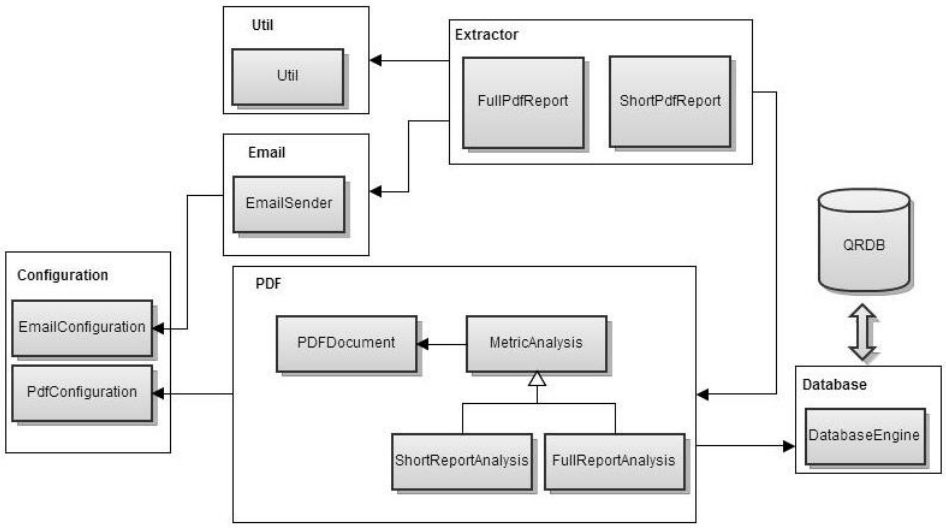

Figura 6.3: Arquitetura da geração dos relatórios.

- **JenkinsExtractor**. Extrai a informação das métricas do *Jenkins* através do *REST API*.
- **GitExtrator**. Extrai a informação das métricas do *GIT* através de *API*, usando a biblioteca *JGit*.

As métricas são extraídas por ordem fixa, pois algumas das métricas correspondem à qualidade do código e outras à qualidade do projeto. Assim sendo, em primeiro lugar, são extraídas as métricas do *Sonar* e de seguida são extraídas as métricas dos restantes recursos (*Jira*, *Jenkins*, *Git*) que são atribuídas aos módulos principais, ou seja, aos projetos.

As métricas atualmente suportadas pelo servidor da qualidade podem ser consultadas no **Anexo E**.

A extração das métricas é feita de modo periódico para permitir a demonstração da evolução de cada métrica. No futuro, o intervalo para extração das métricas será diminuído para ter melhor resolução.

O *PDF Generator* é o segundo módulo do servidor, que permite gerar dois tipos de relatórios – curtos ou completos, gerados com diferentes intervalos de tempo. A arquitetura deste módulo pode ser visualizada na Figura 6.3.

- **QRDB**. É a base de dados onde todas as métricas são guardadas.
- **Util**. É um conjunto dos métodos que permitem manusear as matrizes das métricas (ordenar, filtrar, etc).

- **DatabaseEngine**. Permite obter os valores anteriores e atuais das métricas.
- **FullPdfReport**. Classe principal que gere o relatório completo.
- **ShortPdfReport**. Classe principal que gere o relatório curto.
- **ShortMetricAnalysis**. Contém os métodos necessários para extrair os valores para o relatório curto.
- **FullMetricAnalysis**. Contém os métodos necessários para extrair os valores para o relatório completo.
- **MetricAnalysis**. Contém os métodos comuns ao *ShortMetricAnalysis* e ao *FullMetricAnalysis*.
- **EmailSender**. É responsável pelo envio dos *emails* com os relatórios.
- **EmailConfiguration**. Contém a configuração necessária para enviar os *emails*.
- **PdfConfiguration**. Contém informação sobre os estilos usados para cada tipo de relatório, bem como os métodos necessários para posicionar os valores das métricas.

Como foi dito anteriormente, atualmente existem dois tipos de relatórios – curto e completo, que serão descritos a seguir.

6.3 Relatório de qualidade curto

O relatório curto é enviado semanalmente para a lista interna da *FeedZai* com o objetivo de mostrar a evolução da qualidade do produto. Nas Figuras 6.4 e 6.5 é dado um exemplo de relatório curto, que também pode ser visto no **Anexo F**.

Este relatório é composto no máximo por uma página, de forma a facilitar a leitura. Caso existam violações com alta prioridade, estas serão visualizadas nas páginas seguintes (violações com alta prioridade tem severidade *BLOCKER*). Na primeira página são mostradas as métricas mais relevantes.

O relatório curto é composto por quatro secções diferentes. Cada secção, exceto a última - *Performance*, - é dividida em três linguagens de programação diferentes, *Java*, *JavaScript* e *Scala*.

Primeira secção - *Static Analysis*, mostra os resultados da análise estática do código para cada linguagem de programação. Cada conjunto de métricas, composto por pelo menos uma métrica é apresentado num *dashboard*. Por exemplo, o *dashboard Comments* corresponde ao grupo de "documentação" e contém três métricas - a percentagem de comentários no código, a percentagem de métodos públicos da *API* documentados e número dos métodos não documentados.

A segunda secção - *Test Analysis*, - apresenta a informação relativa aos testes unitários e contém a cobertura dos mesmos, bem como a informação relativa ao estado dos testes.

A terceira secção - *Coding Rules Violations*, - apresenta a percentagem de conformidade quanto às regras definidas pelas ferramentas *checkstyle*, *findbugs* e *pdm* descritas anteriormente.

Quarta secção - *Performance*, - apresenta a evolução do desempenho do produto e contém as médias da memória, do *CPU* e do *throughput*.

Na segunda página do relatório curto e seguintes, é apresentada a informação detalhada das violações com prioridade *BLOCKER*. Cada violação é representada pela parte do código onde esta ocorreu, indicando a linha exata com a mensagem de erro a fim de facilitar a correção da violação.

Para cada métrica, é apresentado o nome, valor atual, a diferença entre o valor atual e o valor anterior e o seu sentido. Por exemplo, aumento das violações é indesejado, sendo por isso o valor da diferença representado com a cor vermelha. Por outro lado, o aumento da percentagem da conformidade da regras é desejável, sendo por isso o valor da diferença representado com a cor verde.

Algumas das métricas têm o valor a zero ou *N/A*. Por exemplo, atualmente não existem ferramentas que permitam analisar o código *Scala* para detetar violações com base em regras (*Checkstyle*, *PMD*, *FindBugs*).

6.4 Relatório de qualidade completo

Em comparação com o relatório curto, o relatório completo contém informação mais aprofundada. O exemplo do relatório completo pode ser visto no **Anexo G**.

O objetivo deste relatório é apresentar a evolução da qualidade do produto

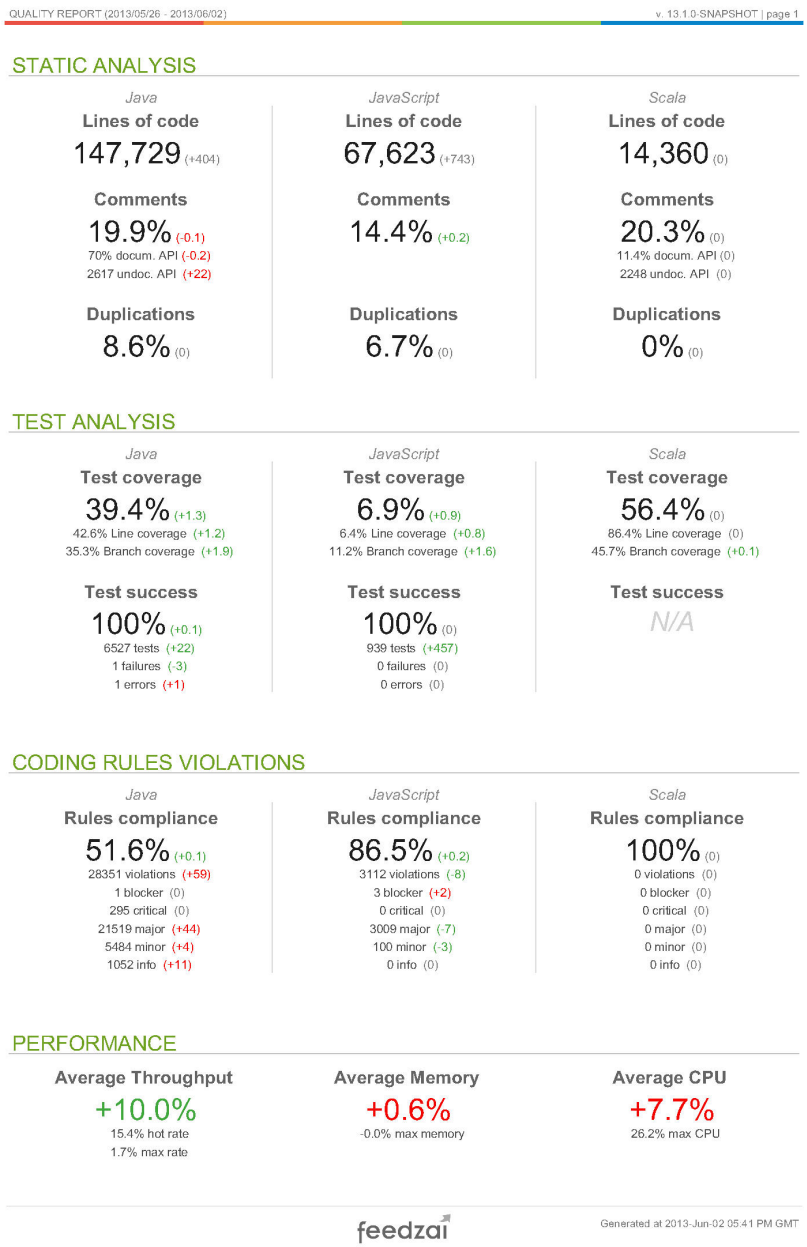

Figura 6.4: Página 1 do relatório curto. Apresenta as métricas mais relevantes.

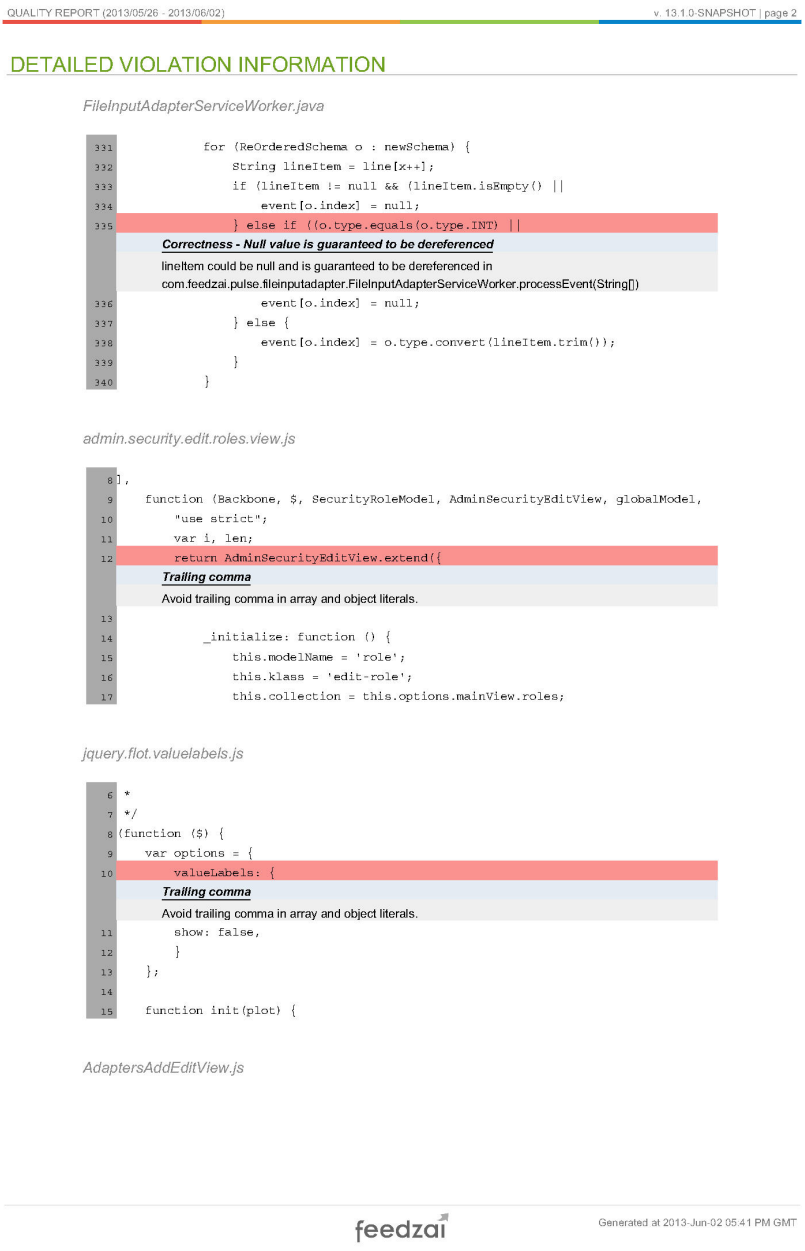

Figura 6.5: Página 2 do relatório curto. Apresenta as violações com alta prioridade.

num período superior a uma semana. Normalmente, este período é de três meses, o que, no entanto, pode ser configurado. A estrutura deste relatório também é diferente da do relatório curto. As seções são apresentadas em páginas separadas. A apresentação das métricas é feita em tabelas de modo a facilitar a leitura do relatório, uma vez que a quantidade da informação em cada secção é maior que no relatório curto.

As métricas são calculadas da mesma maneira que no relatório curto, com a exceção da secção *Performance*. Nesta, a evolução é apresentada para cada *data set* e não através da média como está no relatório curto.

A secção *Other analysis*, ausente no relatório curto, contém a seguinte informação:

- Informação do estado dos *builds* na plataforma de integração contínua *Jenkins*.
- O número de *issues* na plataforma *Jira*.
- O número de *commits* feitos para o *branch* principal, bem como o gráfico dos *commits* efetuados por hora.

6.5 Validação

Esta secção descreve a validação das funcionalidades do servidor da qualidade. Em fase de desenvolvimento foi criado um conjunto de testes unitários. Na fase de validação do sistema foram efetuados os testes do sistema pela equipa de Garantia de Qualidade.

6.5.1 Testes unitários

Para criar os testes unitários destinados ao servidor da qualidade foi escolhida a *framework JUnit*. Esta *framework* é usada pela equipa de desenvolvimento da *FeedZai* para criar os testes unitários para produto *Pulse*. O projecto é compilado e os testes unitários são executados usando o *Maven*. Para executar os testes unitários foi escolhido o plugin *Maven Surefire Plugin*. Este *plugin* é executado na fase de testes e serve precisamente para executar os testes unitários e os testes de integração.

Na tabela 6.1 são apresentados os módulos testados. No total foram criados 145 testes unitários. Para o cálculo da cobertura do código foi usada a

Modulo	#testes - #numero do métodos testados / #total número dos métodos
DatabaseEngine	69 - 41/45
EmailSender	2 - 1/1
Extractor	1 - 1/1
GitExtractor	3 - 4/4
JenkinsExtractor	2 - 2/2
JiraExtractor	2 - 3/3
SonarExtractor	2 - 6/6
GitMetric	2 - 7/11
Metric	9 - 17/20
MetricValue	4 - 8/10
Project	23 - 18/20
SourceCode	2 - 11/12
Violation	3 - 17/22
MetricAnalysisFullReport	7 - 15/15
MetricAnalysisShortReport	3 - 8/8
PdfCreator	2 - 3/3
PdfDocument	8 - 35/40
Util	1 - 1/25

Tabela 6.1: Testes unitários do servidor da qualidade.

ferramenta *Jacoco*. A cobertura total do código fonte é de 78%.

6.5.2 Testes de sistema

Os testes do sistema foram executados pela equipa de Garantia de Qualidade. Durante o período de um mês, ao fim de cada semana foi emitido um relatório curto e um relatório completo. Todos os valores das métricas foram comparados com os valores constantes na base de dados do servidor da qualidade e com os valores das ferramentas usadas.

6.6 Eating your own dog food

Eating your own dog food também é conhecido como *dogfooding*. A premissa base por detrás desta técnica consiste na suposição que, se a empresa espera

que os seus clientes fiquem satisfeitos com o produto, os seus próprios funcionários também o devem estar. Às vezes é difícil encontrar utilização para o próprio produto ou serviço internamente, mas esta técnica tem as seguintes vantagens:

- Detetar as limitações do produto ou serviço.
- Melhorar a usabilidade do produto ou serviço.

Para além das vantagens, esta técnica também tem uma grande desvantagem: o seu uso pode ter um impato negativo sobre a moral dos funcionários.

Uma das tarefas futuras do servidor da qualidade é permitir a visualização das diferentes métricas em tempo real nos ecrãs nas salas de desenvolvimento, com o objetivo de tornar presente a evolução da qualidade do produto. O *PulseViews* foi configurado para usar as métricas extraídas pelo servidor de qualidade a fim de visualizar as métricas em tempo real. Vários *dashboards* foram criados usando as diferentes *widgets* disponíveis no *PulseViews*, que podem ser vistos nas figuras 6.6, 6.7, 6.8, 6.9 e 6.10. Com a entrada de novos dados na base de dados do servidor da qualidade, estes serão imediatamente visualizados nos *dashboards* configurados. As figuras com uma maior resolução podem ser encontradas no **Anexo E**.

O objetivo desta integração é permitir a qualquer programador da *FeedZai* ser capaz de criar o seu próprio *dashboard* para visualizar a informação de uma ou várias métricas da qualidade.

6.7 Conclusão

Todas as alterações efetuadas ao nível do projeto da *FeedZai* serviram não só para melhorar o processo em geral, mas também para permitir a criação dos novos processos. O servidor de qualidade acompanha a evolução de cada métrica em diferentes intervalos de tempo:

- Os relatórios curtos com a informação resumida da qualidade do produto são enviados semanalmente para a equipa de desenvolvimento.
- Os relatórios completos com a informação detalhada sobre a qualidade do produto são enviados a cada três meses para a direção da *FeedZai*.

Usando a técnica *Dogfooding* foi possível integrar o servidor da qualidade com o *PulseViews*, com o objetivo de permitir a visualização das diferentes métricas do produto em qualquer altura.

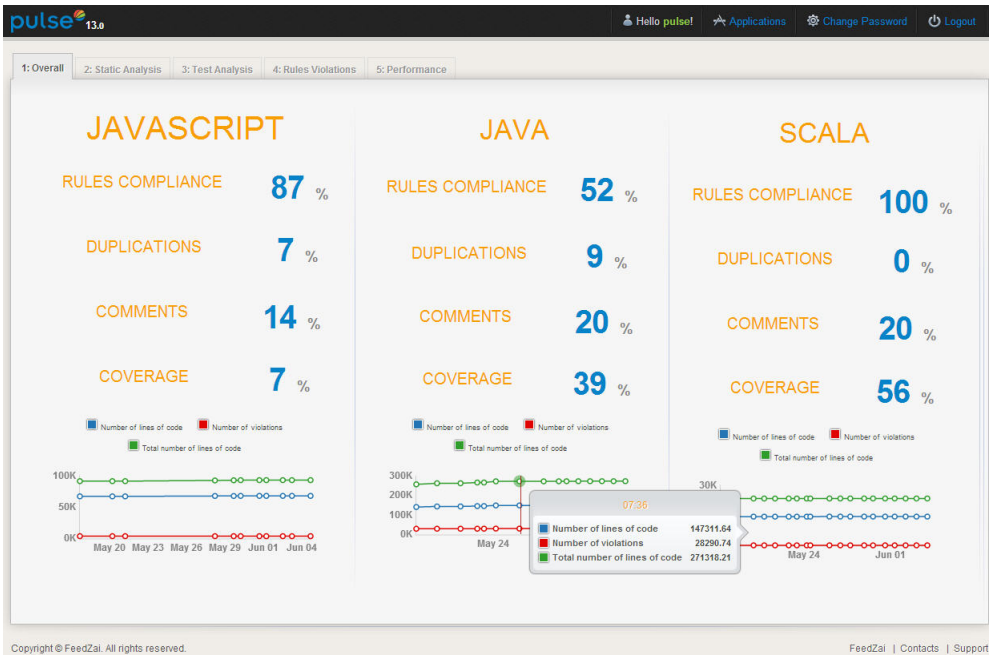

Figura 6.6: Estado global da qualidade do código.

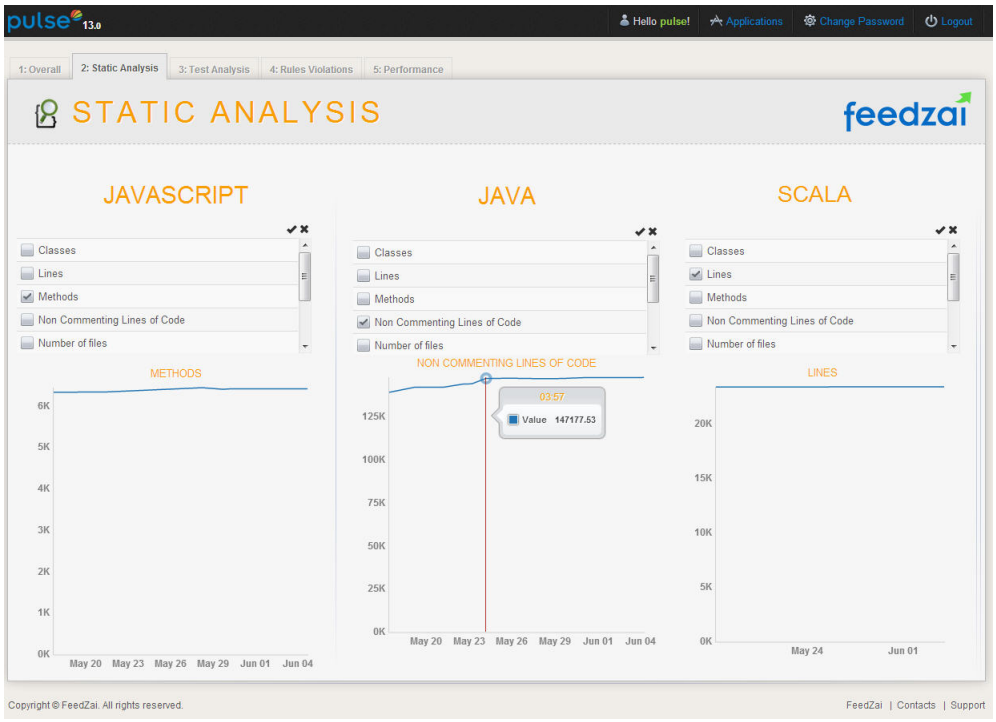

Figura 6.7: Resultados da análise estática do código.

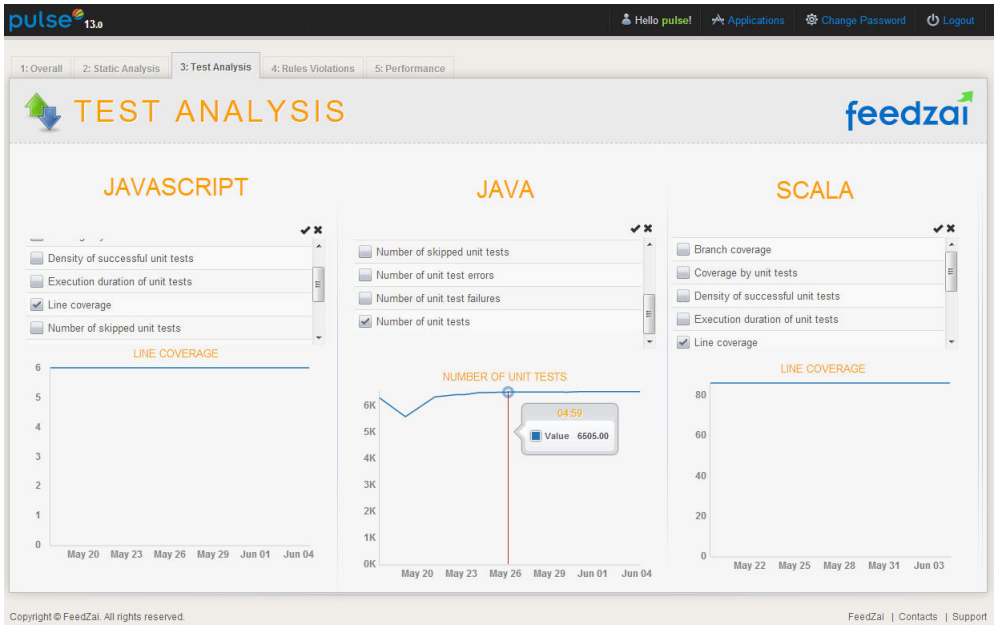

Figura 6.8: Rsultados dos testes unitários, incluindo a cobertura do código.

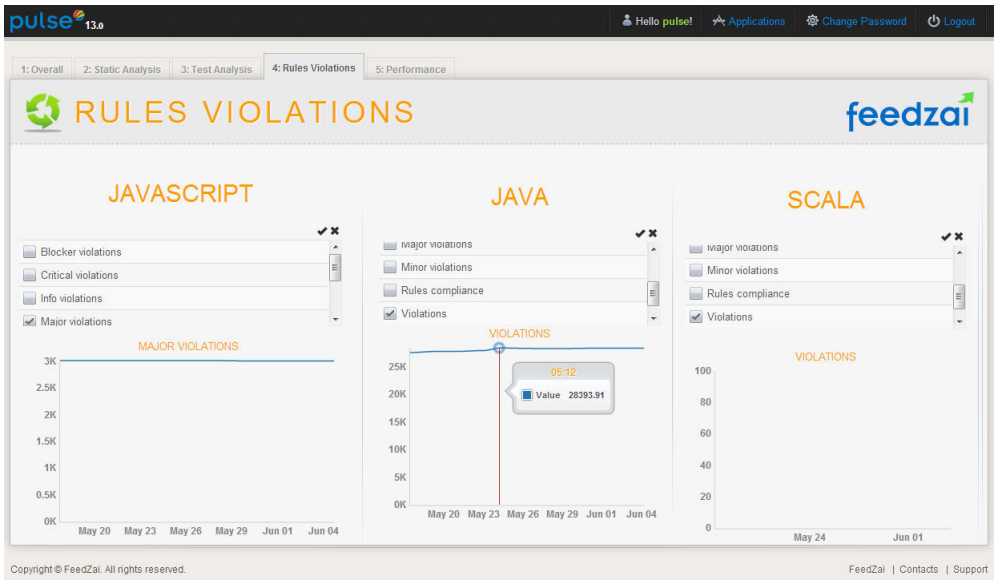

Figura 6.9: Resultados da análise de conformidade das regras.

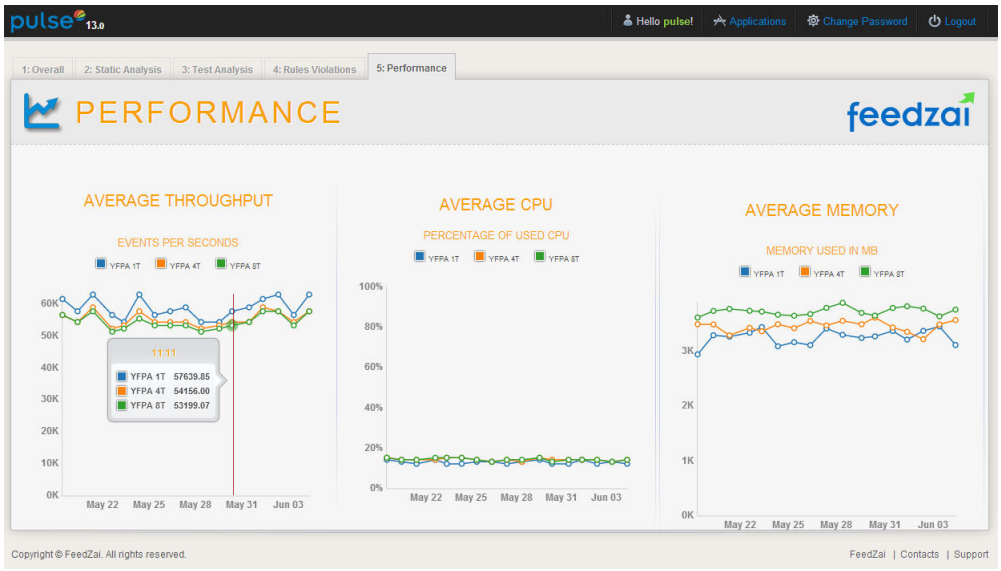

Figura 6.10: Visualização do desempenho do produto com diferentes *data sets*.

Capítulo 7

Plano do estágio

Esta secção descreve o planeamento do primeiro e segundo semestres, com os problemas encontrados e as tarefas não contempladas no planeamento inicial.

7.1 Primeiro semestre

No início do estágio foi elaborado o plano para o 1º semestre do estágio. Este plano presuma que todas as tarefas seram executadas sequencialmente sem desvios maiores. O plano referido pode ser visto na figura 7.1.

A realização das tarefas do plano, incluindo os desvios ao mesmo, está ilustrada na figura 7.2. Como se pode verificar, a maior parte do plano foi cumprida, havendo no entanto quatro exceções:

- O levantamento do estado de arte, que também incluída a análise dos processos da *FeedZai* demorou um pouco mais do que estava previsto

Figura 7.1: Plano do estágio para o 1º semestre.

Figura 7.2: O plano do progresso para o 1º semestre.

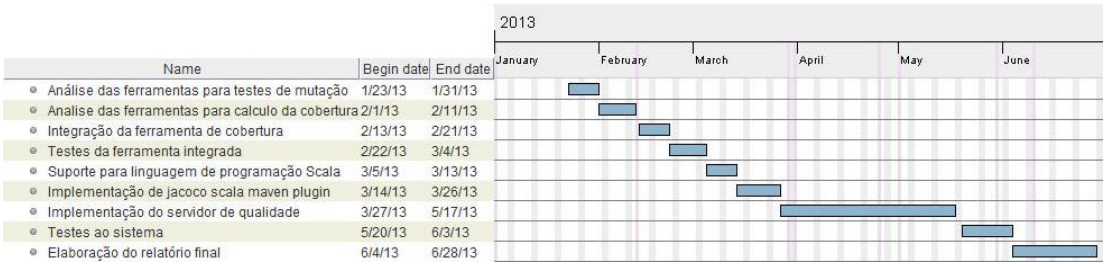

Figura 7.3: Plano do estágio para o 2º semestre.

(capítulos 2 e 3)

- A integração do *Selenium Grid* no *Jenkins* demorou mais tempo devido à instabilidade dos testes do *Selenium* (Capítulo 4)
- Foi desenvolvido o *plugin* para o *Maven*, cuja motivação foi explicada na seção 4.1.3
- A elaboração do relatório intermédio começou mais cedo devido ao volume da informação recolhida na tarefa "Levantamento do estado de arte"

7.2 Segundo semestre

No final do primeiro semestre foi elaborado o plano a cumprir na segunda parte do estágio, que pode ser visto na Figura 7.3.

A realização das tarefas do plano, incluindo os desvios ao mesmo, está ilustrada na figura 7.4.

Como se pode verificar, a maior parte do plano foi cumprida, com as seguintes

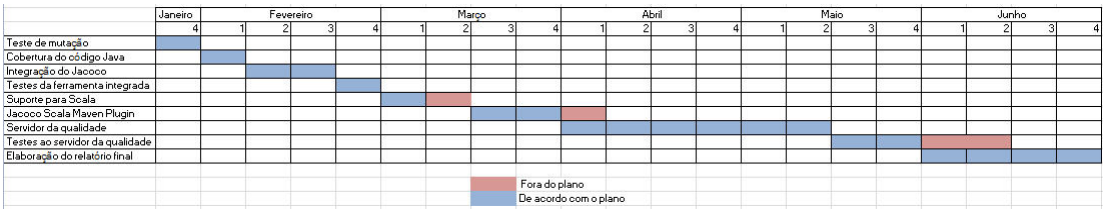

Figura 7.4: Plano do progresso para o 2º semestre.

exceções:

- A elaboração do suporte para a linguagem de programação *Scala* demorou um pouco mais do que estava previsto (Capítulo 5).
- A criação do *JaCoCo Scala Maven Plugin* demorou mais uma semana devido à elaboração da estratégia de arquivo dos resultados.
- Para a realização dos testes ao servidor da qualidade foi necessário emitir os relatórios todos os dias durante um mês, para conseguir comparar os valores das métricas do código fonte.

7.3 Metodologia

Este estágio seguiu a metodologia *Scrum*, abordada na secção **3.2.1**. As **regras** consistem em *Product Owner* - Paulo Marques (CTO) e *Scrum Master* - Diogo Guerra (SVP Product Development); os **eventos** consistem em *sprints* e os **artefactos** em relatórios de 15/5 que podem ser consultados no **Anexo I**. Estes relatórios semanais que contêm uma breve descrição das tarefas realizadas durante a semana, os problemas encontrados durante a realização das tarefas, a lista das tarefas a realizar na semana a seguir e a lista da bibliografia consultada foram enviados para Diogo Guerra, Paulo Marques e Carlos Fonseca.

7.4 Conclusão

Os desvios ao plano não afetaram o trabalho realizado durante o estágio, pois todos os objetivos estabelecidos foram atingidos. Para além disso foi feita

uma demonstração funcional das *features* futuras do servidor da qualidade, como por exemplo, Dogfodding.

Capítulo 8

Conclusões e Trabalho futuro

Olhando para todo o trabalho realizado pode-se concluir que o projeto foi levado a cabo com sucesso. A *FeedZai* passou a ter um conjunto amplo de testes funcionais e de regressão para *PulseViews*. Para além disso, todas as alterações efetuadas ao nível do projeto, incluindo a adição do suporte para linguagem de programação *Scala*, vieram permitir implementar servidor de qualidade para a *FeedZai* para visualizar a evolução das métricas da qualidade ao longo do processo de desenvolvimento.

Este estágio foi uma excelente oportunidade para experimentar o espírito de trabalho que define a maioria das *startups* de sucesso, e para aprender novas técnicas de testes do *software*. Mais do que isso, o estágio transmitiu toda uma nova perspectiva sobre o desenvolvimento de *software*, onde os possíveis defeitos se tornam tão ou mais importantes que a funcionalidade implementada.

Em relação ao trabalho futura pretende-se:

- Usar a técnica *Dogfodding* ou implementar um novo sistema de *dashboards* para visualizar as métricas de qualiidade em tempo real.
- Introduzir novos tipos de testes, por exemplo, testes de segurança.
- Implementar suporte para testes de desempenho para cada módulo principal do *Pulse*.
- Implementar mecanismos para medir a cobertura dos testes do *Selenium*.

Capítulo 9

Bibliografia

1. SELENIUM. Selenium. http://seleniumhq.org. Acesso em 12 de Janeiro de 2013.

2. FEEDZAI. PulseViews. https://docs.feedzai.com/display/pulse121/Pulse+Views. Acesso em 8 de Janeiro de 2013.

3. SOURCE. PIT. http://pitest.org/. Acesso em 29 de Janeiro de 2013.

4. SOURCEFORGE. Jacoco. http://www.eclemma.org/jacoco/. Acesso em 2 de Fevereiro de 2013.

5. PORTO EDITORA, Dicionário da Língua Portuguesa, 2013.

6. K. NAIK, P. TRIPATHY, "Software Testing and Quality Assurance. Theory and Practice". Wiley, 2008.

7. S. PRESSMAN, "Software Engineering - A Practitioner's Approach", McGrawHill, 2005.

8. L. Shuping "The Research of V Model in Testing Embedded Software", Computer Science and Information Technology, August 2008.

9. V. Chen, S. Conte, H. Dunsmore, "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support", IEEE Transactions on Software Engineering, vol. 9, pp. 155 - 165, March 1983.

10. FEEDZAI. HTTP Output Adapter. https://docs.feedzai.com/display/pulse/HTTP+Output+Adapter. Acesso em 8 de Janeiro de 2013.

11. FEEDZAI. Pulse Software Review Board.
https://docs.feedzai.com/display/PULSEDEV/9.+Pulse+Software+Review+Board.
Acesso em 8 de Janeiro de 2013.

12. K. SCHWABER, J. SUTHERLAN, The Scrum Guide.
http://www.scrum.org/Portals/0/Documents/Scrum Guides/Scrum Guide
- Portuguese European.pdf, 2011.

13. SOFTWARE FREEDOM CONSERVANCY. Git. http://git-scm.com.
Acesso em 10 de Janeiro de 2013.

14. SUN MICROSYSTEMS. Jenkins. http://jenkins-ci.org. Acesso em 10
de Janeiro de 2013.

15. FEEDZAI. PerF. https://docs.feedzai.com/display/intra/PerfP+-
+A+performance+evaluation+tool+for+Pulse. Acesso em 10 de Janeiro de
2013.

16. ALTASSIAN. Confluence. http://www.atlassian.com/confluence.
Acesso em 11 de Janeiro de 2013.

17. APACHE SOFTWARE FOUNDATION. Maven.
http://maven.apache.org. Acesso em 10 de Janeiro de 2013.

18. SONATYPE. Nexus. http://www.sonatype.org/nexus. Acesso em 11
de Janeiro de 2013.

19. SONARSOURCE. Sonar. http://www.sonarsource.org. Acesso em 11
de Janeiro de 2013.

20. ALTASSIAN. Jira. http://www.atlassian.com/confluence. Acesso em
11 de Janeiro de 2013.

21. SOURCEFORGE. PMD. http://pmd.sourceforge.net. Acesso em 11 de
Janeiro de 2013.

22. SOURCEFORGE. CheckStyle. http://checkstyle.sourceforge.net.
Acesso em 11 de Janeiro de 2013.

23. SOURCEFORGE. FindBugs. http://findbugs.sourceforge.net. Acesso
em 11 de Janeiro de 2013.

24. FEEDZAI. Longrun Automation.
https://docs.feedzai.com/display/PULSEDEV/8.1.+Longrun+Automation.
Acesso em 12 de Janeiro de 2013.

25. FACEBOOK. Phabricator. http://phabricator.org/. Acesso em 12 de
Janeiro de 2013.

26. FEEDZAI. Code Review.
https://docs.feedzai.com/display/PULSEDEV/1.8.5+-+Code+Review.
Acesso em 12 de Janeiro de 2013.

27. SELENIUM. Selenium Grid. http://selenium-grid.seleniumhq.org.
Acesso em 13 de Janeiro de 2013.

28. SOURCEFORGE. Cobertura. http://cobertura.sourceforge.net/.
Acesso em 1 de Fevereiro de 2013.

Anexos

- Anexo A - Analise das ferramentas para testes funcionais e testes de regressão.
- Anexo B - Requisitos para *Selenium Maven Plugin*.
- Anexo C - Analise das ferramentas para testes de mutação.
- Anexo D - Analise das ferramentas para cobertura do código para *Java*.
- Anexo E - As métricas suportadas pelo servidor da qualidade.
- Anexo F - Exemplo do relatório curto da qualidade.
- Anexo G - Exemplo do relatório completo da qualidade.
- Anexo H - Ecrãs do *PulseViews*.
- Anexo I - Relatórios 15/5.