TQS 2019 | RELATÓRIO TÉCNICO

Manual de Qualidade e Entrega Contínua

|  |  |
| --- | --- |
| Projeto: | **Criação de um sistema capaz de gerir listas de compras, VLoja** |
| Preparado por: | João Paulo Marques Fidalgo, 62243 |
| Data: | 26/09/2019 |

**Conteúdo do relatório**

[1- Accesso rápido aos recursos (*bookmarks* do projeto)](#_Toc18035300)

[2- Gestão e qualidade do código](#_Toc18035301)

[Guia para os colaboradores (*coding style)*](#_Toc18035302)

[SCM *workflow*](#_Toc18035303)

[Revisão de código](#_Toc18035304)

[Análise estática](#_Toc18035305)

[3- Integração contínua & entrega contínua (CI/CD)](#_Toc18035306)

[4- Testes](#_Toc18035307)

[Testes funcionais/aceitação](#_Toc18035308)

[Testes unitários](#_Toc18035309)

[Testes de sistema e de integração](#_Toc18035310)

# 1- Accesso rápido aos recursos (*bookmarks* do projeto)

Sistematização dos links para os recursos desenvolvidos no projecto:

* Acesso ao(s) projecto(s) de código:
  1. [Github](https://github.com/fidalgo91/VLoja/)
  2. [Swagger](https://app.swaggerhub.com/apis/Vloja/Vloja/1.0.0)
* Ambiente SQA :
  1. Análise estática ([Sonarqube](http://192.168.1.1)) todo
  2. Integração Contínua (Jenkins) todo

# 2- Gestão e qualidade do código

## Guia para os colaboradores (*coding style)*

Neste projeto foi usado o estilo de código recomendado pela Android.

Devido à forma prática como são apresentados os problemas e soluções, à sua extensão e ao fato de ser adotada por diversas empresas e ter uma grande reputação no ambiente Java.

Assim, este guião de recomendações aborda várias áreas de criação de código, particularmente, o tratamento de exceções, a adoção de comentários padrão do Javadoc, a formatação de código ao nível de acrónimos, regras para testes, entre outras.

Também importante referir uma linha final das diretrizes apresentadas: ser consistente.

Segue um [*link*](https://source.android.com/setup/contribute/code-style#dont-use-finalizers) sobre o estilo de código adotado.

## SCM *workflow*

Neste projeto, foi utilizado o SCM Github e o método de criação de código por *Feature Branch Workflow*.

A ideia principal de *Feature Branch Workflow* é que todas as novas funcionalidades da aplicação devem ter o seu próprio ramo. Esta encapsulação resulta numa maior facilidade de trabalhar com vários contribuidores sem que existam conflitos ou problemas no *codebase* principal. Também significa que o ramo principal nunca terá código por testar ou disfuncional.

Assim, é assumido um repositório central contendo vários ramos, sendo os principais: *master*, *develop*, *qa* e ramos singulares, designados como *feature branches*.

* No ramo *master* encontra-se o código a ser apresentado ao público, pelo que, antes de haver um envio de código, o mesmo deve ser analisado no processo de Code Review.
* No ramo *develop* encontra-se o código que foi aprovado a partir dos ramos singulares.
* No ramo *qa* encontra-se o código a ser testado pela equipa de testes, que deve primeiro passar pelo *develop.*
* Os ramos singulares são aqueles que contêm apenas uma funcionalidade e devem ser criados cada vez que é introduzida uma nova aplicabilidade. Estes ramos devem ter nomes únicos, como no caso do projeto, vloja-003, desta forma, é possível perceber o fluxo de criação de código e o momento em que cada ramo foi criado.

Sumariamente, os ramos singulares devem, após um processo de Pull Request e Code Review, ser acrescentados ao código principal.

## Revisão de código

Necessário?

[Descrição das práticas definidas no projeto para *code review* e recursos associados.]

## Análise estática

A análise estática é uma forma de melhoria de código que tem várias aplicabilidades e formas de implementação, de forma sucinta, existe uma cotação que ajuda a definir este método:

“Static analysis tools compare favorably to manual reviews because they’re faster, which means they can evaluate programs much more frequently, and they encapsulate some of the knowledge required to perform this type of code analysis in a way that it isn’t require the tool operator to have the same level of expertise as a human auditor. “ - [An overview on the Static Code Analysis approach in Software Development, FE](https://paginas.fe.up.pt/~ei05021/TQSO%20-%20An%20overview%20on%20the%20Static%20Code%20Analysis%20approach%20in%20Software%20Development.pdf)

Ao longo do projeto, será utilizado o SonarQube para análise estática, assim como inspeção de código providenciado pela IDE STS3 (Sprint Tool Suit 3),

Com SonarQube, pretende-se corrigir o código produzido, tratando de *code smells,* vulnerabilidades ou bugs que possam ser detetados no processo de revisão.

Com a STS3, é possível usar várias funcionalidades, como extrair métodos, encapsulação de campos, converter variáveis locais para campos de classe, etc.

# 3- Integração contínua & entrega contínua (CI/CD)

A CI e CD são maneiras de automatizar e testar versões (*builds)* de código produzido antes de serem convergidas com a versão de produção. Enquanto CI apenas aborda as duas primeiras fases do processo de criação versões e testes, CD vai mais longe, criando testes de aceitação e fazendo o *deploy* para ser depois, de forma manual, serem aprovados pelo supervisor do projeto, sendo que, o último passo passa pela automatização de *smoke tests* e a junção ao código de produção.

Assim, existe uma série de boas práticas para a Integração Contínua

* Introduzir novas funcionalidades ao repositório de forma regular
* Mudanças introduzidas no SCM são revistas e devem ser, automaticamente, formas *builds*
* Feedback imediato sobre falha na compilação

Com base nas [recomendações de Fowler](https://martinfowler.com/articles/continuousIntegration.html), são retidas as seguintes práticas:

* Mater apenas um repositório.
* Automatizar a *build.*
* Criar uma build que seja *Self-Testing.*
* Garantir que a *build* seja rápida.
* Testar o código criado num ramo particular (clone do ambiente de produção),
* Facilitar a utilização da última *build*,
* Seja possível, à comunidade, ver o código produzido.
* Automatizar a implantação

De forma a ter um serviço leve e autónomo, serão implementados diversos *containers*, de forma a aceitar diferentes condições e cenários. Consequentemente, será possível testar o sistema em diferentes SO, browsers, etc, e, se possível, fazer testes de performance baseados em concorrência.

# 4- Testes

Esta secção explica diferentes técnicas de teste, com que ferramentas serão implementadas e providencia uma pequena descrição de como proceder e usá-las.

## Testes funcionais/aceitação

Para os testes funcionais, serão utilizadas três ferramentas, Selenium, Katalon Recorder e Junit. Com esta configuração, será testada a interface da aplicação, desde a ligação entre páginas à validação de conteúdo destas.

Usando esta configuração, pretende-se seguir uma abordagem de caixa preta na criação de testes, significa que não precisamos ter acesso ao código fonte para testar a aplicação. Uma das vantagens desta análise é a possibilidade de pessoas externas e sem ligação à criação de código, poderem testar o sistema devido aos baixos requisitos necessários para estes testes.

Segue assim uma citação que descreve este tipo de testes: “The main focus in black box testing is on the functionality of the system as a whole. The term ‘behavioral testing' is also used for black box testing. Behavioral test design is slightly different from the black-box test design because the use of internal knowledge isn't strictly forbidden, but it's still discouraged.” - [Black box testing: An in-dept tutorial with examples and techniques](https://www.softwaretestinghelp.com/black-box-testing/)

## Testes unitários

Para os testes unitários, será utilizado o Junit. Assim, pretende-se testar as funcionalidades da aplicação, ou seja, analisar as diversas operações do sistema. Será possível testar o sistema de *cache* ou, quando se pretende eliminar um produto, este não está a ser utilizado em qualquer lista de compras, o que inviabilizaria essa operação.

## Testes de sistema e de integração

Utilizando RestAssured, é possível testar os *endpoints* da API e entender se os dados solicitados são os esperados.

Com esta libraria, serão realizados testes de caixa branca e preta, ou seja, os testes aplicados à API podem ser feitos sem conhecimento de como as funções funcionam, mas também pode ser feito com técnicas de *mocking*, desta forma, podemos criar testes mais rápidos sem a necessidade de uma base de dados.

Esta abordagem é benéfica para os programadores no sentido de que podemos trabalhar com dados antes de configurar um banco de dados, mesmo que ele não dê um teste adequado, ele simula como a aplicação deve funcionar.