1 - Qualidade do software (conceitos)

1.1 - Fatores de qualidade

1. **Apresentar uma definição de qualidade do software.**

Qualidade - Grau em que um componente, sistema ou processo atende requisitos especificados e/ou necessidades e expectativas do utilizador/cliente.
Qualidade de Software - A totalidade das funcionalidades e características de um produto de software que afetam a sua capacidade de satisfazer necessidades explícitas ou implícitas.

2. **Caracterizar fatores (funcionais e não-funcionais) que contribuem para o grau de qualidade de um sistema de software, de acordo com o standard ISO/IEC 25010.**

Modelo de Qualidade: Fatores e Funcionalidade	
Funcionais	Não-Funcionais
Conveniência: o software é adequado para as tarefas destinadas	Confiável: Quantidade de tempo que o software está disponível para uso
Precisão: resultados/saídas estão corretas e precisas	Usabilidade: avaliação da facilidade de interação do software
Interoperabilidade: capacidade de um componente de software que interaja com outros componentes ou sistemas	Eficiência: software utiliza os recursos do sistema
Segurança: resistente a acessos não intencionais ou modificações	Manutenção: Fácil de reparar/corrigir
	Portabilidade: facilidade de um software correr noutro ambiente

3. **Distinguir entre fatores de qualidade internos e externos, e exemplificar.**

Fatores Internos e Externos de Qualidade (S. McConnell)	
Externos (aqueles que os utilizadores estão cientes)	Internos (aqueles em que o programador está ciente)
Correção (construído livre de falhas)	Manutenção
Usabilidade	Flexibilidade + Portabilidade (adaptável a novos utilizadores ou ambientes)
Eficiência (uso de recursos)	Reutilização (partes podem ser reutilizáveis)
Confiável (longos tempos entre falhas)	Legibilidade e compreensão
Integridade (estado é sempre constante)	Testabilidade
Adaptabilidade (utilizável em novos contextos)	
Robustez (contra adversidades - recuperação de exceções)	

4. **Explicar, por palavra próprias, o "dilema da qualidade" (até onde investir/quando parar de testar?).**

5. **"Até 80% dos custos e esforços globais no ciclo de vida de desenvolvimento de software são gastos em manutenção" [Thumma2010]. A Maintainability deve ser considerada uma categoria importante durante o processo de desenvolvimento de software. Explique a natureza/contributo dos atributos de qualidade associados (ISO-25010): Analyzability, Modifiability, Modularity, Reusability, Testability.**

Analyzability: degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.

Modifiability: degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.

Modularity: degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.

Reusability: degree to which an asset can be used in more than one system, or in building other assets.
Testability: degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

6. **Identifique características de um modelo de qualidade relacionadas com a Usability (ISO-25010)**

Appropriateness recognisability: degree to which users can recognize whether a product or system is appropriate for their needs.
Learnability: degree to which a product or system enables the user to learn how to use it with effectiveness, efficiency in emergency situations.
Operability: degree to which a product or system is easy to operate, control and appropriate to use.
User error protection: degree to which a product or system protects users against making errors.
User interface aesthetics: degree to which a user interface enables pleasing and satisfying interaction for the user.
Accessibility: degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use.

7. **A adequação/rigor funcional é comumente apresentada como fator de qualidade (o produto faz o que deve). Como é que a functional suitability pode ser dividida (ISO-25010)?**

Functional Completeness: degree to which the set of functions covers all the specified tasks and user objectives.
Functional Correctness: degree to which the functions provides the correct results with the needed degree of precision.
Functional Appropriateness: degree to which the functions facilitate the accomplishment of specified tasks and objectives.

1.2 - Métricas e monitorização da qualidade

1. **Apresentar diferentes métricas usadas na avaliação do grau de cobertura.**

Function coverage – has each function (or subroutine) in the program been called?
Statement coverage – has each statement in the program been executed?
Edge coverage – has every edge in the Control flow graph been executed?
Branch coverage – has each branch (also called DD-path) of each control structure (such as in if and case statements) been executed? For example, given an if statement, have both the true and false branches been executed? This is a subset of edge coverage.
Condition coverage (or predicate coverage) – has each Boolean sub-expression evaluated both to true and false?

2. **Admita a utilização da métrica defeitos/kLOC para medir a densidade de defeitos num dado projeto. Critique o uso desta métrica, analisando, por exemplo, a sua eficácia ou facilidade de implementar. (kLOC = 1000 Lines Of Code).**

3. **Argumentar sobre o nível de cobertura de código que deve ser incluído num quality gate.**

80%

4. **Explicar/definir as métricas de qualidade usadas na dashboard do SonarQube.**

Code Smell: A maintainability-related issue in the code. Leaving it as-is means that at best maintainers will have a harder time than they should making changes to the code. At worst, they'll be so confused by the state of the code that they'll introduce additional errors as they make changes.
Bug: An issue that represents something wrong in the code. If this has not broken yet, it will, and probably at the worst possible moment. This needs to be fixed. Yesterday.
Vulnerability: A security-related issue which represents a potential backdoor for attackers.

5. **Caracterize a métrica maintainability (facilidade de manutenção), analisando os elementos que podem contribuir para a definir.**

Modularity: degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
Reusability: degree to which an asset can be used in more than one system, or in building other assets.
Analyzability: degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
Modifiability: degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
Testability: degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

6. **Que fatores objetivos podem influenciar a métrica de complexidade do código?**

Refactoring: Reduce complexity → easier to understand and evolve

7. **Explicar o conceito technical debt e o seu impacto na gestão da qualidade.**

Technical debt: the implied cost of additional rework caused by choosing an easy (limited) solution now instead of using a better approach that would take longer. Custo implícito de retrabalho adicional causado pela escolha de uma solução fácil agora, em vez de usar uma abordagem melhor que levaria mais tempo.

2 - Teste de software

2.1 - Conceitos e princípios

1. **Distinguir entre verificação e validação.**

Verificação e Validação	
Verificação	**Validação**
Estamos a desenvolver o sistema da maneira certa?	Estamos a desenvolver o sistema correto?
Verificar a consistência dos módulos (verificar se estamos a desenvolver as *features* corretas para resolver o problema)	Verificar se o sistema atende às expectativas do cliente

2. **Distinguir entre testes unitários, de integração, de sistema e funcionais/aceitação.**

Testes Unitários - Cada módulo faz o que é suposto fazer?

- Fase em que se testam as menores unidades de software desenvolvidas (métodos, classes, pequenos trechos de código). O objetivo é encontrar falhas de funcionamento dentro de uma pequena parte do sistema funcionando independentemente do todo.

Testes de Integração - O resultado é o esperado quando as partes são colocadas juntas?

- Na fase de teste de integração, o objetivo é encontrar falhas provenientes da integração interna dos componentes de um sistema. Geralmente os tipos de falhas encontradas são de transmissão de dados (exemplo: um componente A pode estar aguardando o retorno de um valor X ao executar um método do componente B; porém, B pode retornar um valor Y, gerando uma falha)
- **Objetivo**: expor os defeitos nas interfaces e nas interações entre componentes integrados no sistema

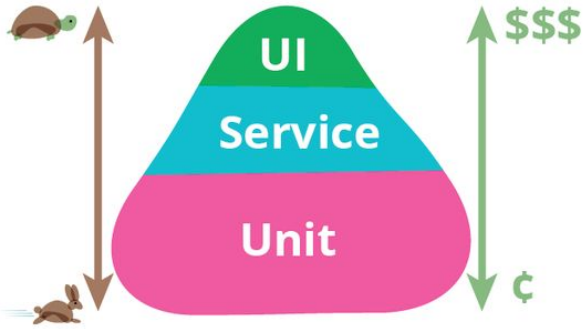
Testes de Aceitação/Validação - O programa satisfaz os requisitos?

- Geralmente, os testes de aceitação são realizados por um grupo restrito de utilizadores finais do sistema, que simulam operações de rotina do sistema de modo a verificar se seu comportamento está de acordo com o solicitado, de maneira a determinar a satisfação dos critérios reclamados pelo cliente.

Testes de Sistema - Todo o sistema funciona como esperado?

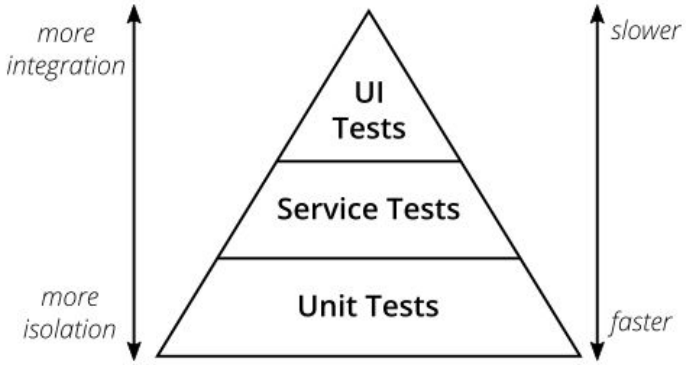
- **Objetivo**: Executar o sistema sob ponto de vista de seu utilizador final, percorrendo todas as funcionalidades em busca de falhas em relação aos objetivos originais.

3. **Interpretar o modelo da "pirâmide dos testes", explicando a razão da existência de camadas verticais e dos diferentes "tamanhos" de cada nível.**



- Write tests with different granularity
- The more high-level you get the fewer tests you should have

4. **Associar tipos de teste aos níveis da pirâmide dos testes e aos papéis na equipa.**



5. **O que são testes de desempenho (performance) e de carga (load)? O que medem?**

Testes de Desempenho - Os testes de desempenho são idênticos aos testes de carga mas com o intuito de testar o software a fim de encontrar o seu limite de processamento de dados no seu melhor desempenho. No teste é avaliada a capacidade de resposta em determinados cenários e configurações, determinando assim a sua escalabilidade e confiança.

Testes de Carga - Identifica os níveis máximos os quais um sistema/aplicação pode realizar com sucesso em termos de carga de trabalho e número de utilizadores virtuais.

6. **O que são testes “flacky”? Qual o papel das estratégias de “mocking” neste contexto?**

Flaky test: is an analysis of web application code that fails to produce the same result each time the same analysis is run. Whenever new code is written to develop or update computer software, a web page or an app, it needs to be tested throughout the development process to make sure the application does what it's supposed to do when it's released for use. Logically, when put through the same test over and over, the code will produce the same result -- the application will either work properly every time, thus passing the test, or fail to work properly every time, thus failing the test.
Mocking: Objetos Mock, objetos simulados ou simplesmente Mock em desenvolvimento de software são objetos que simulam o comportamento de objetos reais de forma controlada. São normalmente criados para testar o comportamento de outros objetos

7. **Justificar a necessidade de, por regra, os ambientes de teste de desempenho suportarem a execução distribuída.**

8. **Explicar a necessidade de os programadores suplementar o seu código de produção com código de verificação, relacionando com práticas de entrega contínua.**

9. **Relacionar os diferentes níveis de abstração nos testes com o âmbito dos objetos/assuntos sob teste. Discutir a forma como esta relação é observada no V-Model e nos métodos ágeis.**

No **V-Model** a execução dos processos acontece de forma sequencial em forma de V. Também é conhecido como um modelo de verificação e validação. O V-Model é uma melhoria do modelo em cascata e baseia-se na associação de uma fase de testes para cada fase de desenvolvimento correspondente. Isto significa que, para cada fase do ciclo de desenvolvimento, há uma fase de teste diretamente associada. Este é um modelo altamente disciplinado e a próxima fase começa apenas após a finalização da fase anterior.
BDD: É uma técnica de desenvolvimento ágil que encoraja a colaboração de developers, setores de qualidade e pessoas não técnicas. O desenvolvimento deve ser orientado aos comportamentos que o sistema deve apresentar. Desta

forma, um comportamento (requisito/especificação) é priorizado em relação ao teste unitário, o que não exclui a execução do fluxo TDD neste processo.
TDD: É uma técnica de desenvolvimento ágil onde o desenvolvimento deve ser orientado a testes, onde cada teste unitário deve ser escrito antes que uma funcionalidade do sistema o seja. O objetivo é fazer com que o teste passe com sucesso, significando que assim a funcionalidade está pronta e conta com garantia de qualidade.

10. **Porque é que os testes unitários devem, em geral, ser independentes? Em que condições deixam de o ser?**

Independent: Not depend or interfere with other tests. You cannot rely upon one unit test to do the setup work for another unit test. Not guaranteed to run in a particular order

11. **Porque é importante testar de forma "isolada", relativamente aos testes unitários?**

Unit testing assumes using units in isolation Unit tests verify the "local" contract. The purpose is to Tests components individually. Answer the question: does the component function properly, in isolation?

2.2 - Processo e práticas de teste e qualidade de software

1. **Explicar o sentido do princípio metodológico "testar tão cedo quanto possível" (early testing).**

Para prevenir erros tardios e custos inesperados, usar metodologias TDD, BDD podem ajudar nisto

2. **Confrontar a abordagem proposta no V-Model com uma aproximação Behaviour-driven development, caracterizando os aspetos distintivos de cada qual.**

No **V-Model** a execução dos processos acontece de forma sequencial em forma de V. Também é conhecido como um modelo de verificação e validação. O V-Model é uma melhoria do modelo em cascata e baseia-se na associação de uma fase de testes para cada fase de desenvolvimento correspondente. Isto significa que, para cada fase do ciclo de desenvolvimento, há uma fase de teste diretamente associada. Este é um modelo altamente disciplinado e a próxima fase começa apenas após a finalização da fase anterior.
BDD: É uma técnica de desenvolvimento ágil que encoraja a colaboração de developers, setores de qualidade e pessoas não técnicas. O desenvolvimento deve ser orientado aos comportamentos que o sistema deve apresentar. Desta

forma, um comportamento (requisito/especificação) é priorizado em relação ao teste unitário, o que não exclui a execução do fluxo TDD neste processo.

3. **Relacionar os conceitos de TDD e BDD (e os respetivos modelos de processo).**

BDD: É uma técnica de desenvolvimento ágil que encoraja a colaboração de developers, setores de qualidade e pessoas não técnicas. O desenvolvimento deve ser orientado aos comportamentos que o sistema deve apresentar. Desta forma, um comportamento (requisito/especificação) é priorizado em relação ao teste unitário, o que não exclui a execução do fluxo TDD neste processo.
TDD: É uma técnica de desenvolvimento ágil onde o desenvolvimento deve ser orientado a testes, onde cada teste unitário deve ser escrito antes que uma funcionalidade do sistema o seja. O objetivo é fazer com que o teste passe com sucesso, significando que assim a funcionalidade está pronta e conta com garantia de qualidade.

4. **Explicar as implicações do modelo "tradicional" (Debug Later) e do modelo TDD no tempo necessário para a descoberta e correção de falhas.**

We've all done it—written a bunch of code and then toiled to make it work. Build it and then fix it. Testing was something we did after the code was done. It was always an afterthought, but it was the only way we knew.

We would spend about half our time in the unpredictable activity affectionately called *debugging*. Debugging would show up in our schedules under the disguise of test and integration. It was always a source of risk and uncertainty. Fixing one bug might lead to another and sometimes to a cascade of other bugs. We'd keep statistics to help predict

5. **Confrontar as seguintes estratégias para realizar testes que dependem do comportamento de frameworks/serviços remotos: utilização de mocks para resolver dependências vs utilização de embedded containners.**

Mocks: Objetos pré-programados com as expectativas, isto é, a especificação das chamadas que são esperadas para receber. Verificação compara as chamadas recebidas com as expectativas
In-Container Testing: Containers são ativados para ativar o ambiente de teste. Requer mecanismos para implementar e executar testes num container

6. **Explicar o sentido da frase "os critérios para a aceitação do sistema devem ser executáveis"**

Cucumber

7. **Explicar ao padrão Page Objetc Model no contexto dos testes funcionais (sobre a camada web).**

Page Object model is an object design pattern in Selenium, where web pages are represented as classes, and the various elements on the page are defined as variables on the class.

8. **Tendo em conta que há erros de runtime imprevisíveis, os testes relativos a exceções devem ser considerados para a verificação do contrato de um componente?**

Runtime Exceptions: indicam erros de programação. indicam pré-condições foram violadas. todas as "unchecked throwables" implementadas devem estender *RunTimeException*

9. **Tradicionalmente, o desenvolvimento é bottom-up (dos elementos da infraestrutura para os de apresentação), respeitando a ordem natural das dependências. Há alguma vantagem em desenvolver numa lógica top-down? Como é que a estratégia de testes pode ajudar/deve ser orientada?**

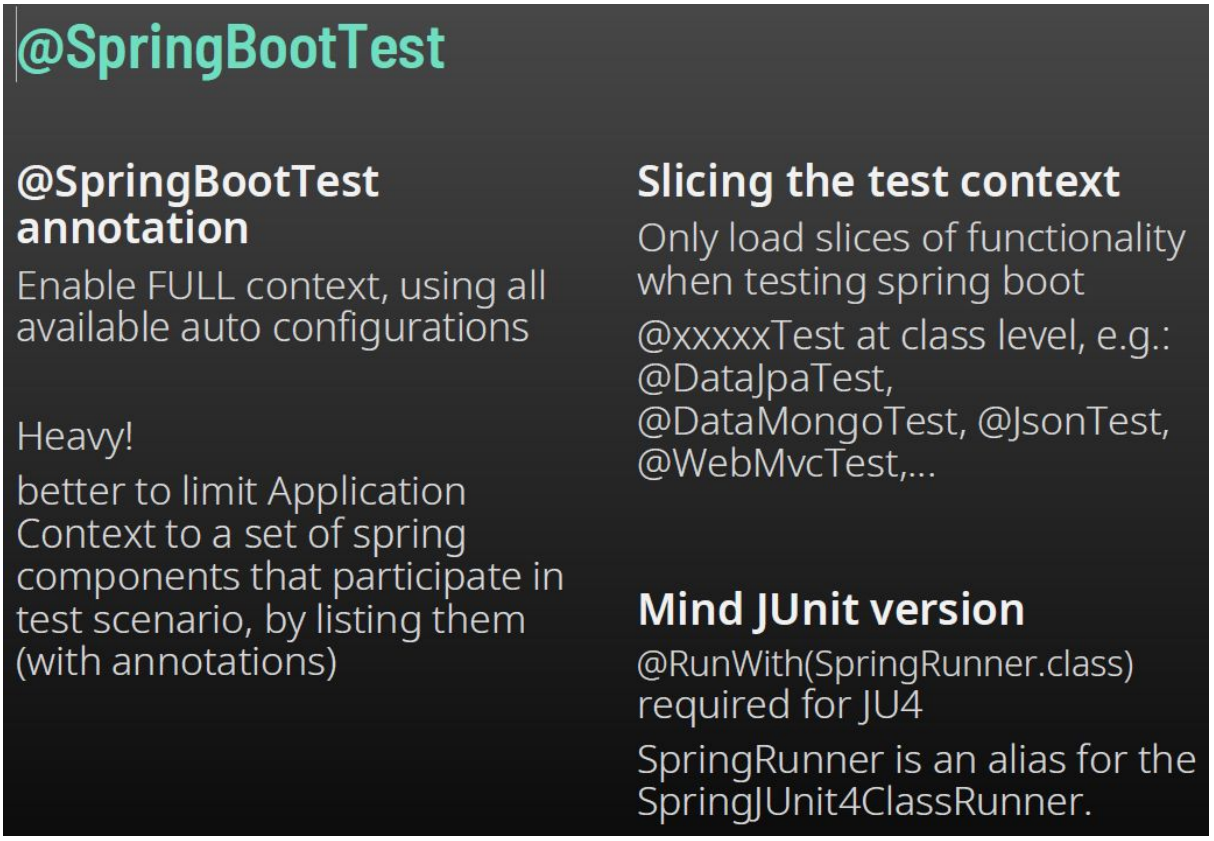
A **top-down approach** is essentially the breaking down of a system to gain insight into its compositional sub-systems in a reverse engineering fashion. In a top-down approach an overview of the system is formulated, specifying, but not detailing, any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", which makes it easier to manipulate. However, black boxes may fail to clarify elementary mechanisms or be detailed enough to realistically validate the model. Top down approach starts with the big picture. It breaks down from there into smaller segments.
A **bottom-up approach** is the piecing together of systems to give rise to more complex systems, thus making the original systems sub-systems of the emergent system. Bottom-up processing is a type of information processing based on incoming data from the environment to form a perception. From a cognitive psychology perspective, information enters the eyes in one direction (sensory input, or the "bottom"), and is then turned into an image by the brain that can be interpreted and recognized as a perception (output that is "built up" from processing to final cognition). In a bottom-up approach the individual base

elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, by which the beginnings are small but eventually grow in complexity and completeness. However, "organic strategies" may result in a tangle of elements and subsystems, developed in isolation and subject to local optimization as opposed to meeting a global purpose.

10. **Qual a vantagem de usar browsers sem interface gráfica (headless) em testes funcionais, sobre a camada de apresentação?**

Headless testing is when you run a UI-based browser test without showing the browser UI. It's running a test or running a script against a browser but without the browser, UI starting up. ... Using a headless browser might not be very helpful for browsing the Web, but for automating tasks and tests it's awesome.

11. **Explique a relação entre diferentes tipos de testes do Spring Boot, denotados com anotações como @DataJpaTest, @WebMvcTest, @SpringBootTest, etc.**



2.3 - Trechos de código e tipos de teste no Exame

À semelhança de exames anteriores, é possível que se peça, no exame, para interpretar um conjunto de testes simples, e as partes de código associados. Nesse contexto, tenha presente, em particular, a escrita de:

1. **Testes unitários, com JUnit, com ou sem mocking de dependências;**
2. **Diversos tipos de teste em aplicações SpringBoot, com e sem mocking de serviços/componentes;**
3. **Testes de integração, para verificação de API, utilizando o TestRestTemplate e MockMvc da SpringBoot.**

3 - Garantia de qualidade no processo de engenharia de software

3.1 - Práticas gerais

1. **A partir de textos correntes relativos a oportunidades de emprego, identificar e caracterizar tecnologias e ferramentas próprias do processo de SQA.**
2. **Caracterizar o perfil (competências) de um Engenheiro de Qualidade de software e caracterizar um elenco de ferramentas que devem constar do seu portfólio.**
3. **"Maybe even your team is one of those teams that have good intentions on testing, but it always gets put off or forgotten as the projects get rolling. Why is testing so hard to do consistently? Testing benefits are well-known, and yet, why is it so often overlooked?" (Vuk Skobalj). Apresente argumentos para justificar a falta de adesão a uma prática generalizada de testes. Apresente uma solução metodológica para garantir que os testes sejam mais que "boas intenções" e tão importantes como o código de produção.**

3.2 - Práticas de equipa

1. **Distinguir os três estados em que se podem encontrar os ficheiros, num sistema de controlo de versões git.**

Unmodified: These are any files that haven't been modified since the last commit. They will still be included in the next commit, but remain as is.
Modified: These are files that have been modified since the last commit (we probably modified these as part of bug fixes). These files will be included in the next commit, but will be included in their respective new form.
Staged: These are files that are either not present in the last commit (e.g. newly created files) or are "modified" files that we tell git to include in the next commit. Files are added to the staging using git's "add" command.

2. **Explicar o workflow "feature-branch", na utilização do Git (e variações associadas, e.g..GitHub flow e GitLab flow)**

A **feature branch** is a source code branching pattern where a developer opens a branch when she starts working on a new feature. She does all the work on the feature on this branch and integrates the changes with the rest of the team when the feature is done.

3. **Explicar o processo de "merge request" (ou "pull request") na integração de contributos num projeto open source e o papel das ferramentas que podem facilitar esse workflow.**

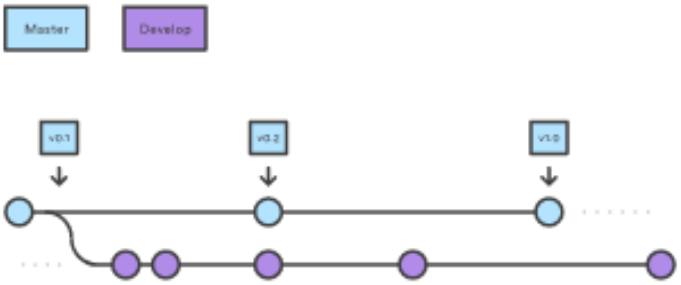
Merge requests allow you to visualize and collaborate on the proposed changes to source code that exist as commits on a given Git branch. A Merge Request (MR) is the basis of GitLab as a code collaboration and version control platform. It's exactly as the name implies: a request to merge one branch into another.
A **pull request** (PR) is a method of submitting contributions to an open development project. It occurs when a developer asks for changes committed to an external repository to be considered for inclusion in a project's main repository after the peer review.

4. **Para que serve um guia de boas práticas num projeto de desenvolvimento de código? Que tipo de informação deve conter?**

Para informar os colaboradores de como colaborar, oq fazer, oq não fazer, etc

5. **Explique o sentido de cada uma das práticas partilhadas preconizada no "Android open source project"**

6. **Como é que se caracteriza o modelo de trabalho do "GitFlow workflow"?**



3.3 - Revisão de código

1. **Apresentar e discutir as principais vantagens da adoção de práticas de revisão de código.**

Code review helps to maintain consistent coding style across the company. Teaching and sharing knowledge. During review team members gain better understanding of the code base and learn from each other. Consistent design and implementation.

2. **Como é que a revisão de código é incluída no processo Extreme Programming (que utiliza Pair-Programming)?**

Pair Programming: Dois autores desenvolvem código em conjunto. Revisão de código contínua. Não posso verificar o código que "eu" fiz. Uma forma de "revisão de código contínua"

3. **O código dos testes também deve ser objeto de revisão pelos pares?**

4. **Que tipo de defeitos são mais suscetíveis de serem revelados numa revisão de código?**

Deviations from standards: either internally defined and managed or regulatory/legally defined
Requirements defects: e.g.: the requirements are ambiguous, or there are missing elements.
Design defects: e.g.: too much coupling; fail to use known patterns
Insufficient maintainability: e.g.: the code is too complex to maintain
Incorrect interface specifications

5. **Encontrar defeitos num processo de revisão de código é algo de bom ou de mau? Explique à luz da cultura que deve ser adotada.**

Bom

3.4 - Análise estática de código

1. **Que tipo de problemas são mais suscetíveis de serem reveladas num processo sistemático (e automático) de análise estática de código?**

- Analysis of code patterns, without running the code
- Examples of issues found in SA:
- Referencing a variable with an undefined value
- Variables that are never used
- Unreachable (dead) code
- Programming standards violations
- Security vulnerabilities
- Internationalization (i18n) issues

2. **Apresentar exemplos de ferramentas viáveis para a análise estática de código (para projetos Java).**

Codacy, Sonarcube

3. **Explique o sentido da afirmação "a análise estática de código contribui para encontrar defeitos invisíveis".**

O olho humano falha mas o computador nao

4. **Exemplifique situações que podem ser sinalizadas como vulnerabilidade de segurança na análise estática.**

Vulnerability: A security related issue which represents a potential backdoor for attackers.

3.5 - Redesenho (refactoring)

1. **Apresentar exemplos comuns de transformações de código enquadradas no conceito de refactoring.**

- Adição de parâmetros
- Substituir condições por polimorfismo
- Extrair interface
- Extrair (código duplicado dentro de um) método

- **Extract method**: Seleciona parte do método para formar um novo, substituindo a seleção chamando para o novo
- **Extract interface**: Cria uma nova interface usando alguns métodos de uma classe, que seguidamente, irá implementar a nova interface
- **Encapsulate field**: Cria os métodos get e set para os campos e usa apenas aqueles para aceder ao mesmo

2. **Explicar em que consistem as transformações suportadas nas operações de refactoring disponíveis em IDE populares (e.g., ItelliJ IDEA).**

3. **A linguagem Java usa o mecanismo de exceções para o tratamento de erros, distinguindo entre exceções do tipo checked e unchecked. Como é que se devem usar as exceções e, em particular, estes dois subtipos?**

4. **Identificar más práticas na utilização das Exceções em Java e propor refactoring adequados.**

Esperar exceptions.

5. **Que fatores podem influenciar positiva ou negativamente a complexidade cognitiva de um trecho de código?**

6. **Exemplifique anti-padrões (bad-smells) recorrentes e oportunidades de refactoring associadas.**

Code Smell: A maintainability-related issue in the code. Leaving it as-is means that at best maintainers will have a harder time than they should making changes to the code. At worst, they'll be so confused by the state of the code that they'll introduce additional errors as they make changes.
https://sourcemaking.com/refactoring

3.6 - Integração contínua

1. **Explicar cada uma das práticas enumeradas por Fowler no contexto da implementação de um sistema de integração contínua.**

1. Manter num único repositório
2. Automatizar o Build
3. Tornar o Build automático a nível de testes
4. Todos fazem commit todos os dias
5. Cada commit deve construir uma linha principal na máquina de integração
6. Manter a Build rápida
7. Testar num ambiente de produção clone
8. Torná-lo acessível a todos
9. Todos podem ver o que está a acontecer
10. Automatizar a implementação

2. **Descrever o fluxo de trabalho (do desenvolvimento em equipa) quando se aplica uma abordagem de integração continua.**

1. Checkout / atualizar do SCM
2. Codificar um novo recurso
3. Executar automaticamente na máquina local
    - Repetir os pontos 2 e 3 até os testes passarem
4. Juntar a cópia local com as mais recentes mudanças do SCM
    - Reparar e reconstruir até os testes passarem
5. Commit
6. Executar numa máquina "clean"
    - Correção de bugs e problemas de integração imediatos

3. **O que é a cultura de integração continua? Como é que isso "está para além das ferramentas"?**

4. **Distinguir ente Continuous integration, Continuous Delivery e Continuous Deployment.**

Integração Contínua: Consiste em construir um projeto regular e automático com execução de testes automatizados, testes de qualidade, mecanismos de integração e implementação de binários em máquinas de execução ou de repositórios partilhados. CI torna o processo de desenvolvimento mais suave, mais previsível e menos arriscado.
Continuous Delivery: sw development practice in which you build software in such a way that it can be released to production at any time. You're doing

continuous delivery when: Focus on quality of working software. Your software is deployable throughout its lifecycle. Your team prioritizes keeping the software deployable over working on new features. Anybody can get fast, automated feedback on the production readiness

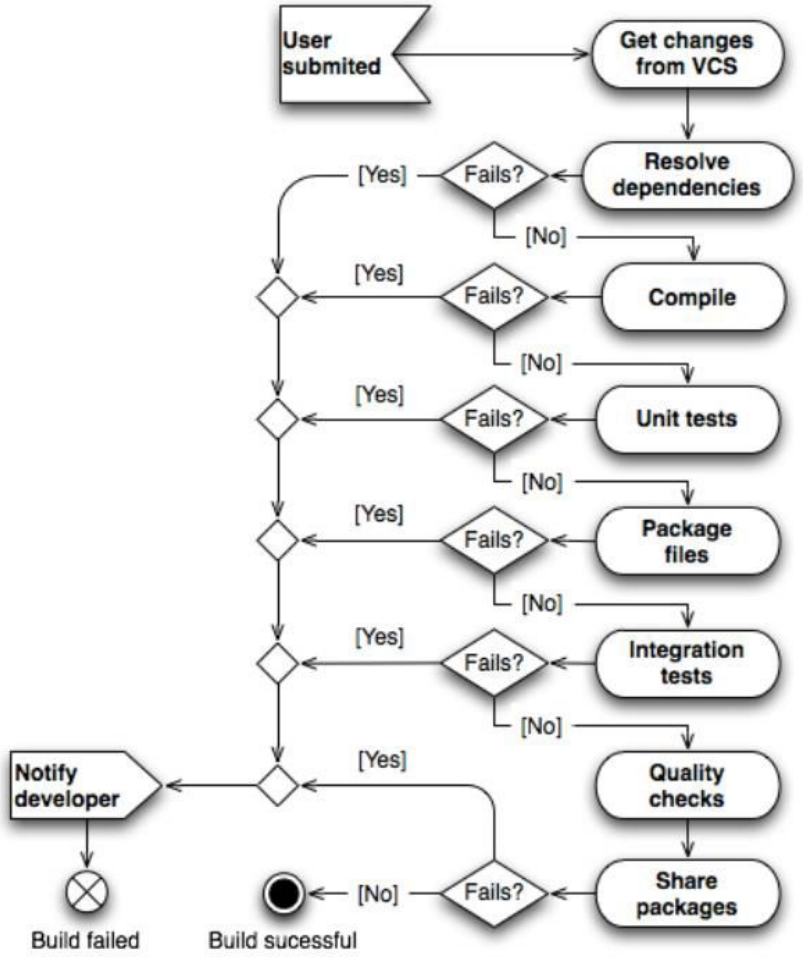
Continuous Deployment/release: every change goes through the pipeline and automatically gets put into production. Focus on speed and agility to deploy to production

5. **O feedback é crucial num processo de CI. De que tipo de feedback se trata?**

Continuous feedback: Errors are easier to detect in an earlier stage, near the point where they have been introduced: The detection mechanism of such bugs becomes simpler because the natural step in diagnosing the problem is to check what was the latest submitted change. problems followed by atomic commits are easiest to correct than to fix several problems at once, after bulk commits. There must be an effective mechanism that automatically informs programmers, testers, database administrators and managers about the status of the build Feedback → generate reaction in a more accurate and prompter way

6. **Porque é que a integração é chamada de contínua?**

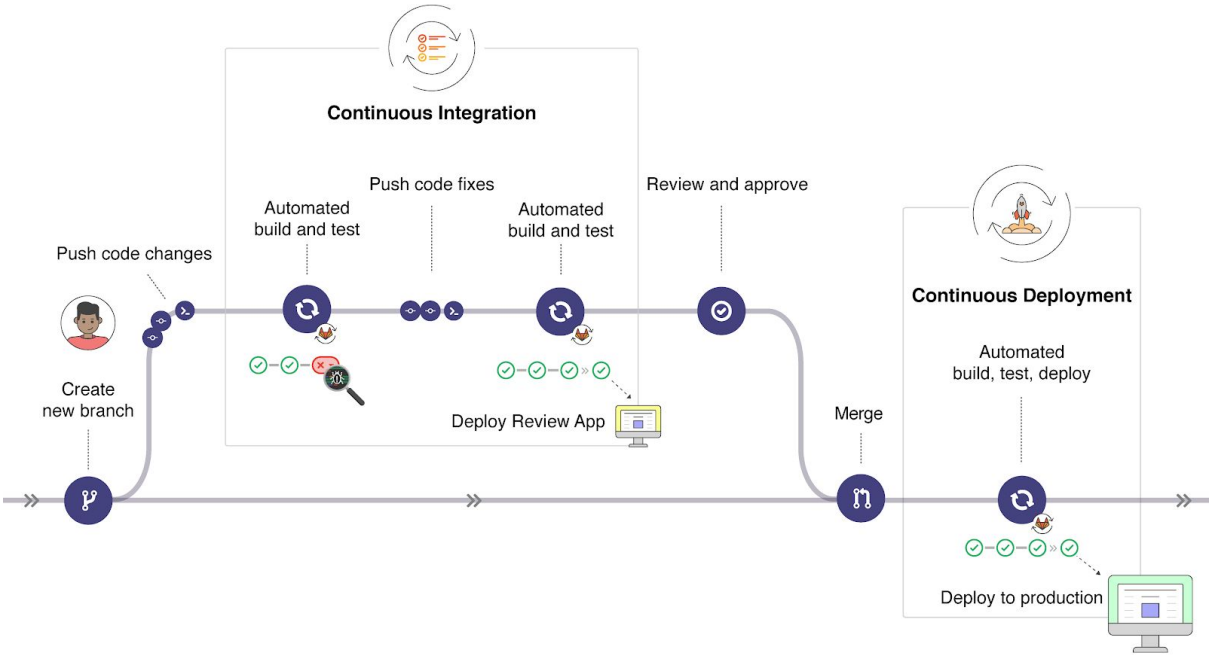
7. **Distinguir os conceitos de "compilar" (compile) e "construir" (build process).**



Build process: A build has several stages (goals in Maven terms). A successful build implies success in code correctness and quality checks. Automatic build tools run quality checks (e.g.: unit testing, code inspections)

3.7 - Entrega contínua

1. **Apresentar o workflow típico de CD**



2. **Comentar as ideia expressas neste trecho: "The pattern that is central CD is the deployment pipeline. A deployment pipeline is an automated implementation of your application's build, deploy, test, and release process. Every organization will have differences in the implementation of their deployment pipelines, depending on their value-stream for releasing software, but the principles that govern them do not vary." [Humble]**

3. **Explicar qual o papel de cada ferramenta num processo de CD, designam ente: Jenkins, repositório de artefactos (e.g.: Artifactory), Docker e ferramentas associadas.**

4. **Apresentar argumentos a favor e cenários de Continuous Delivery vs. Continuous Deployment.**

Continuous Delivery: means that you are ready and able to deploy any version to any supported platform at any time.
Continuous Deployment: means that you are engaging in actual deployment.

5. **Relacionar ferramentas como Jenkins, GitLab CI, GitHub Actions, Travis CI no contexto de CI/CD.**

6. **Relacionar o papel de estratégias baseadas em containers, locais (e.g.: Docker) ou em larga escala (e.g.: Kubernetes), com os processos de entrega contínua e coexistência de diferentes ambientes de qualidade.**

4 - SQA e métodos ágeis

1. **No contexto dos métodos ágeis, o que é uma (user) story?**

(User) Story: as the basic unit of functionality, and therefore of delivery. Captures a feature of the system defines the scope of the feature and its acceptance criteria. They are also used as the basis for estimation when we come to do our planning. Can be mapped on outcomes, requirements

2. **Apresentar, no contexto no projeto do grupo, exemplos concretos de stories, incluindo a sua descrição (âmbito funcional e critérios de aceitação) de acordo com o esquema de documentação proposto nos métodos ágeis.**

3. **Relacionar BDD com testes funcionais de um sistema.**

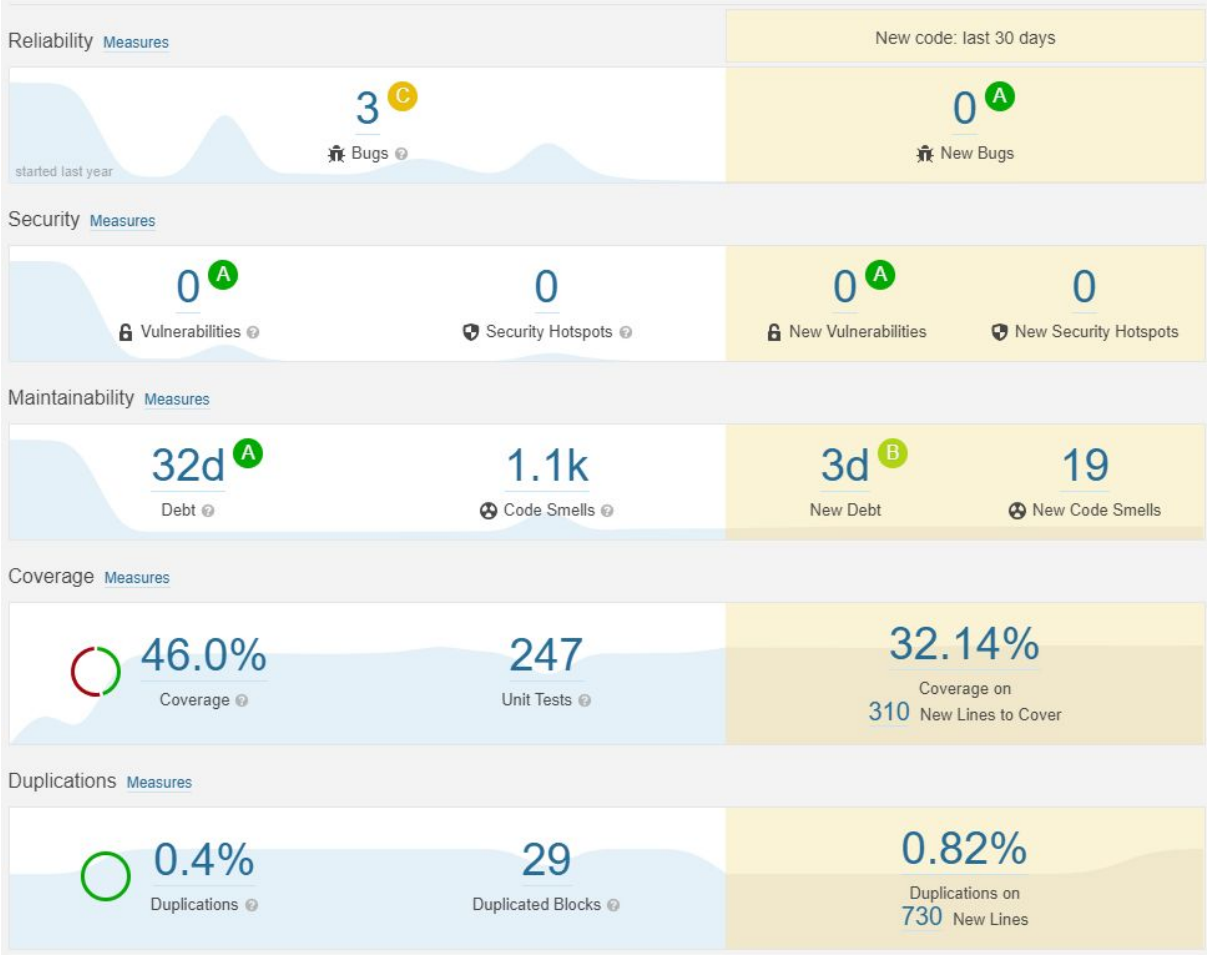
4. **Como é que as user stories podem ser vistas como um instrumento conversacional para recolher e evoluir os requisitos do sistema?**

5. **Como é que a user story é usada como unidade elementar de funcionalidade do produto (na gestão do trabalho, verificação de qualidade, entrega contínua)?**

BDD: we drive that development by first stating the requirements as, well, requirements. The form of those requirements is fairly rigid, allowing them to be interpreted by a tool that can execute them in a manner that is similar to unit tests.

5 - Ferramentas

1. **Interpretar, a partir de um exemplo concreto, a informação apresentada no painel de monitorização (dashboard) do SonarQube.**



2. **Escrever testes unitários para cenários simples, utilizando o framework JUnit. (v5).**

3. **Escrever testes unitários simples com a utilização de "mock objects" para obter comportamento previsível de módulos externos (com Mockito).**

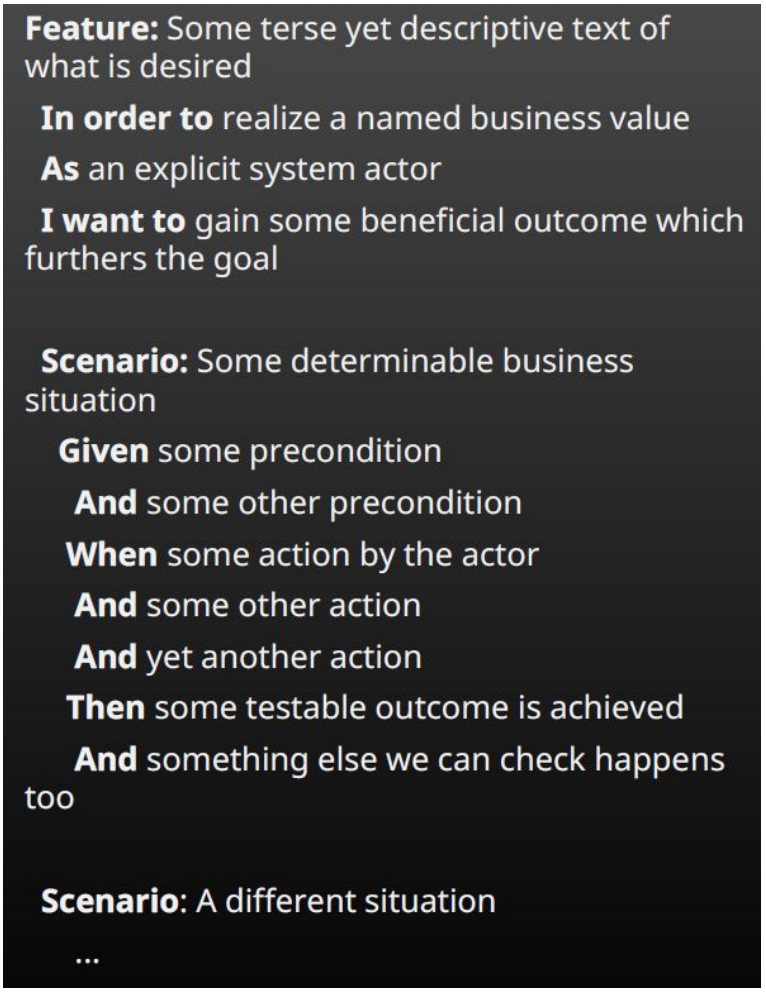
4. **Reconhecer a utilização das primitivas do Mockito em excertos de código e a explicar a respetiva ação.**

5. **Explicar o papel da interface WebDriver (do framework Selenium) na automação de testes de aceitação, em Java.**

6. **Interpretar e explicar por palavras próprias um teste funcional escrito com recurso a JUnit e Selenium WebDriver.**

7. **A ferramenta maven define um conjunto de objetivos progressivos (maven goals) para a construção do projeto. Caraterize os objetivos mais comuns.**

8. **Escrever (a especificação de) uma feature para ser usada no framework Cucumber dado um cenário de teste de uma story.**



9. **Descrever, ilustrar com um digrama, e concretizar com comandos, o workflow principal de colaboração quando se utiliza um repositório Git para integração dos contributos numa equipa.**

10. **Interpretar uma página de resultados do painel de monitorização (dashboard) do Jenkins, para um projeto.**

11. **Interpretar ficheiros de configuração do Jenkins (Jenkinsfile) e argumentar quanto às vantagens de usar "pipelines as code".**

12. **Como é que as tecnologias de virtualização baseada em containers (e.g.: Docker) estão relacionadas com a prática de Continuous Deployment?**

13. **Apresentar estratégias para realizar testes de integração sobre serviços REST (na ótica de quem os desenvolve).**

14. **Distinguir e interpretar diferentes tipos de teste (unitários, integração) no contexto das boas práticas em SpringBoot.**

15. **Explicar um processo de testes e ferramentas necessárias para implementar abordagem BDD sobre um projeto full-stack em Java EE.**

16. **O JMeter permite a criação de testes modulares com a inserção de vários Samplers, Timers e Listners. Explique o papel dos elementos designados.**

Sampler	An action that causes a request.
Config	Additional configuration.
Timer	Add a predefined delay.

Results reporting (by Listeners)

17. **Distinguir o papel de @Mock e @Spy (no framework Mockito) e os respetivos casos de utilização.**

The most used widely used annotation in Mockito is **@Mock**. We can use @Mock to create and inject mocked instances without having to call Mockito.mock manually.

@Spy: A field annotated with @Spy can be initialized explicitly at declaration point. Alternatively, if you don't provide the instance Mockito will try to find zero argument constructor (even private) and create an instance for you. But Mockito cannot instantiate inner classes, local classes, abstract classes and interfaces. For example this class can be instantiated by Mockito :

Annotation	Purpose
@Mock	create and **inject mocked instances** (without having to call Mockito.mock() "manually") when()/given() to specify how a mock should behave
@Spy	**partial mocking**, real methods are invoked but still can be verified and stubbed. Every call, unless specified otherwise, is delegated to the object.

18. **Relacionar o @InjectMocks com o padrão de software inversão do controlo (IoC)**

Design patterns in action: Inversion of Control

Applying the IoC pattern to a class means removing the creation of all object instances for which this class isn't directly responsible and passing any needed instances instead. The instances may be passed using a specific constructor, using a setter, or as parameters of the methods needing them. It becomes the responsibility of the calling code to correctly set these domain objects on the called class.[2]

One last point to note is that if you write your test first, you'll automatically design your code to be flexible. Flexibility is a key point when writing a unit test. If you test first, you won't incur the cost of refactoring your code for flexibility later.