Tópicos de estudo para o exame

Atualizado em: 2020-06-14

Os tópicos apresentados servem como elementos de autodiagnóstico e de preparação do exame e não constituem, nem limitam, as perguntas do teste.
(alterações desde a versão *draft* marcadas a amarelo; denota apenas modificações, e não importância relativa...)

Conteúdos:

1 Qualidade do software (conceitos) 1
1.1 Fatores de qualidade 1
1.2 Métricas e monitorização da qualidade 2

2 Teste de software 2
2.1 Conceitos e princípios 2
2.2 Processo e práticas de teste e qualidade de software 2
2.3 Trechos de código e tipos de teste no Exame 3

3 Garantia de qualidade no processo de engenharia de software 3
3.1 Práticas gerais 3
3.2 Práticas de equipa 3
3.3 Revisão de código 4
3.4 Análise estática de código 4
3.5 Redesenho (*refactoring*) 4
3.6 Integração contínua 4
3.7 Entrega contínua 4

4 SQA e métodos ágeis 5
5 Ferramentas 5

1 Qualidade do software (conceitos)

1.1 Fatores de qualidade

- Apresentar uma definição de qualidade do software.
- Caraterizar 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.
- Distinguir entre fatores de qualidade internos e externos, e exemplificar.
- Explicar, por palavra próprias, o "dilema da qualidade" (até onde investir/quando parar de testar?).
- "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*.
- Identifique caraterísticas de um modelo de qualidade relacionadas com a *Usability (*ISO-25010*)*
- A adequação/rigor funcional é comummente apresentada como fator de qualidade (o produto faz o que deve). Como é que a *functional suitability* pode ser dividida *(*ISO-25010*)*?

1.2 Métricas e monitorização da qualidade

- Apresentar diferentes métricas usadas na avaliação do grau de cobertura.
- 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).
- Argumentar sobre o nível de cobertura de código que deve ser incluído num *quality gate*.
- Explicar/definir as métricas de qualidade usadas na *dashboard* do SonarQube.
- Caraterize a métrica *maintainability* (facilidade de manutenção), analisando os elementos que podem contribuir para a definir.
- Que fatores objetivos podem influenciar a métrica de complexidade do código?
- Explicar o conceito *technical debt* e o seu impacto na gestão da qualidade.

2 Teste de software

2.1 Conceitos e princípios

- Distinguir entre verificação e validação.
- Distinguir entre testes unitários, de integração, de sistema e funcionais/aceitação.
- Interpretar o modelo da "pirâmide dos testes", explicando a razão da existência de camadas verticais e dos diferentes "tamanhos" de cada nível.
- Associar tipos de teste aos níveis da pirâmide dos testes e aos papéis na equipa.
- O que são testes de desempenho (*performance*) e de carga (*load)*? O que medem?
- O que são testes "flacky"? Qual o papel das estratégias de "mocking" neste contexto?
- Justificar a necessidade de, por regra, os ambientes de teste de desempenho suportarem a execução distribuída.
- Explicar a necessidade de os programadores suplementarem o seu código de produção com código de verificação, relacionando com práticas de entrega contínua.
- 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.
- Porque é que os testes unitários devem, em geral, ser independentes? Em que condições deixam de o ser?
- Porque é importante testar de forma "isolada", relativamente aos testes unitários?

2.2 Processo e práticas de teste e qualidade de software

- Explicar o sentido do princípio metodológico "testar tão cedo quanto possível" (*early testing*).
- Confrontar a abordagem proposta no V-Model com uma aproximação *Behaviour-driven development*, caraterizando os aspetos distintivos de cada qual.
- Relacionar os conceitos de TDD e BDD (e os respetivos modelos de processo).
- Explicar as implicações do modelo "tradicional" (*Debug Later*) e do modelo TDD no tempo necessário para a descoberta e correção de falhas.
- 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*.
- Explicar o sentido da frase "os critérios para a aceitação do sistema devem ser executáveis"
- Explicar ao padrão Page Objetc Model no contexto dos testes funcionais (sobre a camada web).
- 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?

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

- Testes unitários, com JUnit, com ou sem *mocking* de dependências;
- Diversos tipos de teste em aplicações SpringBoot, com e sem *mocking* de serviços/componentes;
- 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

- A partir de textos correntes relativos a oportunidades de emprego, identificar e caraterizar tecnologias e ferramentas próprias do processo de SQA.
- Caraterizar o perfil (competências) de um Engenheiro se Qualidade de software e caraterizar um elenco de ferramentas que devem constar do seu portfolio.
- *"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

- Distinguir os três estados em que se podem encontrar os ficheiros, num sistema de controlo de versões *git*.
- Explicar o *workflow* "*feature-branch*", na utilização do Git (e variações associadas, e.g..GitHub flow e GitLab flow)
- 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*.
- Para que serve um guia de boas práticas num projeto de desenvolvimento de código? Que tipo de informação deve conter?
- Explique o sentido de cada uma das práticas partilhadas preconizada no "Android open source project"
- Como é que se carateriza o modelo de trabalho do "GitFlow workflow"?

3.3 Revisão de código

- Apresentar e discutir as principais vantagens da adoção de práticas de revisão de código.
- Como é que a revisão de código é incluída no processo Extreme Programming (que utiliza P*air-Programming*)?
- O código dos testes também deve ser objeto de revisão pelos pares?
- Que tipo de defeitos são mais suscetíveis de serem revelados numa revisão de código?
- Encontrar defeitos num processo de revisão de código é algo de bom ou de mau? Explique à luz da cultura que deve ser adotada.

3.4 Análise estática de código

- Que tipo de problemas são mais suscetíveis de serem reveladas num processo sistemático (e automático) de análise estática de código?
- Apresentar exemplos de ferramentas viáveis para a análise estática de código (para projetos Java).
- Explique o sentido da afirmação "a análise estática de código contribui para encontrar defeitos invisíveis".
- Exemplifique situações que podem ser sinalizadas como vulnerabilidade de segurança na análise estática.

3.5 Redesenho (*refactoring*)

- Apresentar exemplos comuns de transformações de código enquadradas no conceito de *refactoring*.
- Explicar em que consistem as transformações suportadas nas operações de *refactoring* disponíveis em IDE populares (e.g., ItelliJ IDEA).
- 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?
- Identificar más práticas na utilização das Exceções em Java e propor *refactoring* adequados.
- Que fatores podem influenciar positiva ou negativamente a complexidade cognitiva de um trecho de código?
- Exemplifique anti-padrões (*bad-smells)* recorrentes e oportunidades de *refactoring* associadas.

3.6 Integração contínua

- Explicar cada uma das práticas enumeradas por Fowler no contexto da implementação de um sistema de integração contínua.
- Descrever o fluxo de trabalho (do desenvolvimento em equipa) quando se aplica uma abordagem de integração continua.
- O que é a cultura de integração continua? Como é que isso "está para além das ferramentas"?
- Distinguir ente Continuous integration, Continuous Delivery e Continuous Deployment.
- O *feedback* é crucial num processo de CI. De que tipo de *feedback* se trata?
- Porque é que a integração é chamada de contínua?
- Distinguir os conceitos de "compilar" (*compile*) e "construir" (*build process).*

3.7 Entrega contínua

- Apresentar o workflow típico de CD

- 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]
- Explicar qual o papel de cada ferramenta num processo de CD, designam ente: Jenkins, repositório de artefactos (e.g.: Artifactory), Docker e ferramentas associadas.
- Apresentar argumentos a favor e cenários de *Continuous Delivery* vs. *Continuous Deployment*.
- Relacionar ferramentas como Jenkins, GitLab CI, GitHub Actions, Travis CI no contexto de CI/CD.
- 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

- No contexto dos métodos ágeis, o que é uma (*user) story*?
- 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.
- Relacionar BDD com testes funcionais de um sistema.
- Como é que as *user stories* podem ser vistas como um instrumento conversacional para recolher e evoluir os requisitos do sistema?
- Como é que a *user story* é usada como unidade elementar de funcionalidade do produto (na gestão do trabalho, verificação de qualidade, entrega contínua)?

5 Ferramentas

- Interpretar, a partir de um exemplo concreto, a informação apresentada no painel de monitorização (*dashboard*) do SonarQube.
- Escrever testes unitários para cenários simples, utilizando o *framework* JUnit. (v5).
- Escrever testes unitários simples com a utilização de "*mock objects*" para obter comportamento previsível de módulos externos (com Mockito).
- Reconhecer a utilização das primitivas do Mockito em excertos de código e a explicar a respetiva ação.
- Explicar o papel da interface WebDriver (do framework Selenium) na automação de testes de aceitação, em Java.
- Interpretar e explicar por palavras próprias um teste funcional escrito com recurso a JUnit e Selenium WebDriver.
- A ferramenta *maven* define um conjunto de objetivos progressivos (*maven goals*) para a construção do projeto. Caraterize os objetivos mais comuns.
- Escrever (a especificação de) uma *feature* para ser usada no *framework* Cucumber dado um cenário de teste de uma *story*.
- 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.
- Interpretar uma página de resultados do painel de monitorização (*dashboard*) do Jenkins, para um projeto.
- Interpretar ficheiros de configuração do Jenkins (Jenkinsfile) e argumentar quanto às vantagens de usar "pipelines as code".
- Como é que as tecnologias de virtualização baseada em *containers* (e.g.: Docker) estão relacionadas com a prática de Continuous Deployment?

- Apresentar estratégias para realizar testes de integração sobre serviços REST (na ótica de quem os desenvolve).
- Distinguir e interpretar diferentes tipos de teste (unitários, integração) no contexto das boas práticas em SpringBoot.
- Explicar um processo de testes e ferramentas necessárias para implementar abordagem BDD sobre um projeto *full-stack* em Java EE.
- O JMeter permite a criação de testes modulares com a inserção de vários *Samplers*, *Timers* e *Listners*. Explique o papel dos elementos designados.
- Distinguir o papel de @Mock e @Spy (no *framework* Mockito) e os respetivos casos de utilização.
- Relacionar o @InjectMocks com o padrão de software inversão do controlo (IoC)