

## Exame de Época Normal – 2019-06-13 15h00. Duração: 90min (+10 tolerância).

NOME:

Nr.MEC:

Questões de escolha múltipla (1 a 20): **responda na grelha**, assinalando uma opção por pergunta (pretende-se a **opção verdadeira** e, havendo várias que possam ser consideradas parcialmente verdadeiras, pretende-se a mais abrangente); as não-respostas valem zero; **respostas erradas descontam**  $\frac{1}{4}$  da cotação; as respostas assinaladas de forma ambígua serão consideradas não-respostas.

Questões 21 e 22: responder no espaço vazio, no fim do enunciado.

Teste: A1

Grelha de respostas para a escolha múltipla (perguntas 1 a 20):

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
a)																				
b)																				
c)																				
d)																				
e)																				

### P1.

O modelo de qualidade do ISO-25010 identifica vários parâmetros relativos à qualidade do produto (de software), entre eles a “*Maintainability*”, que está relacionado com:

- Integridade dos dados, não repúdio das ações, reutilização dos módulos.
- Facilidade de operação por parte dos utilizadores, tolerância a falhas pontuais do hardware, facilidade de alterar um componente sem introduzir defeitos ou degradar a qualidade já existente.
- Capacidade para estabelecer e executar testes de um sistema, uso de módulos para gerir a interdependência de componentes, capacidade de analisar o impacto de uma alteração no sistema global.
- Facilidade de testar, facilidade de operação, percepção da utilidade do sistema pelos utilizadores.
- Eficiência na utilização interna dos recursos, tempos de resposta das operações de acordo com os requisitos estabelecidos, existência de mecanismos de prevenção dos erros de utilização.

### P2.

A metáfora da “pirâmide dos testes” quer transmitir a ideia de que:

- Existem testes mais importantes que outros (do topo para a base da pirâmide).
- O esforço da equipa com as atividade de teste é cumulativo e aumenta de iteração para iteração.
- O número de testes diminui com o *burndown*, i.e., à medida que menos itens de trabalho subsistem no *backlog*, há menos testes para executar.
- Os testes podem ser agrupados tendo em conta a sua granularidade e objetivos.
- Os testes das camadas superiores devem usar os testes das camadas inferiores.

### P3.

Que problemas poderiam ser apontados a um portfolio de testes baseado numa “pirâmide invertida” (menos testes unitários, mais testes de aceitação)?

- Demora muito tempo a executar, é difícil de manter, perde especificidade na localização dos problemas.
- Não é escalável porque os testes de aceitação requerem operação manual.
- Obriga a fazer a gestão explícita de *user stories*, numa ferramenta adicional, não integradas no repositório de código.
- Não é adequado para projetos orientados à disponibilização de API/serviços.
- É impeditivo da adoção de práticas de *refactoring* de código, pois não oferece a necessária “rede de segurança” para prevenir regressões.

### P4.

A análise estática de código:

- é uma forma de detectar defeitos com muito baixo custo.
- permite a validação antecipada/precoce dos requisitos dos utilizadores.
- a análise estática, com ferramentas completas, tornam o teste dinâmico desnecessário.
- a análise estática torna possível encontrar defeitos de *runtime* logo início do ciclo de desenvolvimento.
- na análise de vulnerabilidades de segurança, a análise estática tem menos valor que os testes dinâmicos, já que estes são mais eficazes a localizar os defeitos do código.

### P5.

Qual das seguintes opções é a descrição mais adequada do conceito de cobertura (de instruções) do código?

- é uma métrica usada para medir a percentagem de testes que foram executados com sucesso.

- b) é uma métrica que traduz o número de defeitos corrigidos sobre o total de defeitos encontrados.
- c) é uma métrica utilizada para medir a percentagem de linhas de código fonte que são realmente executadas.
- d) é uma métrica que determina a relação das instruções que foram executadas num conjunto de testes, em relação ao total de instruções.
- e) é uma métrica que dá uma confirmação verdadeiro/falso se todas as instruções são utilizadas pelos testes.

#### P6.

Nos projetos baseados em práticas ágeis, é mais necessário implementar a automação de testes do que em projetos “tradicionais”, porque:

- i. as alterações aos requisitos acontecem diariamente e os incrementos precisam de ser testados quanto a possíveis regressões. A alteração diária requer testes automatizados, porque o teste manual é muito lento.
- ii. os testes devem gerar feedback sobre a qualidade do produto o mais cedo possível. Portanto, todos os testes de aceitação devem ser executados em cada iteração, idealmente logo que as modificações são feitas, recorrendo a testes automatizados.
- iii. a prática de integração contínua exige que o conjunto de testes de regressão seja executado sempre que o código é entregue, para gerar feedback sobre o estado da *build*. Na prática, só pode ser realizado por testes automatizados.
- iv. As iterações/sprints são de duração fixa. A equipa deve garantir que todos os testes podem ser completamente executados no último dia de cada iteração/Sprint. Na prática, isso requer testes automatizados.
- v. Os projetos ágeis dependem de testes unitários em detrimento de testes de sistemas. Como os testes unitários não podem ser executados manualmente, todos os testes devem ser automatizados.

Selecione o par de afirmações verdadeiras:

- a) i) e ii)      b) i) e iii)
- c) ii) e iii)      d) i) e iv)      e) iii) e v)

#### P7.

Qual a definição mais adequada de “*user story*”, tal como é usada nos testes em métodos ágeis?

- a) Um artefato a preparar pelos os *testers*, para detalhar apenas os requisitos funcionais do sistema.
- b) Um artefacto do projeto, a produzir pelos programadores e que o *Product Owner* deve aprovar antes que os testes possa começar.
- c) Um artefato preparado pelos representantes do negócio/cliente para orientar os programadores e *testers* quanto às condições de aceitação do incremento.
- d) Um artefato escrito colaborativamente pelos programadores, *testers* e especialistas do negócio para fixar os requisitos do produto.

- e) Um artefacto, apresentado de forma breve, para documentar novas funcionalidades e *bugs* que precisam de ser corrigidos.

#### P8.

Segundo M. Fowler, qual das seguintes práticas NÃO é recomendada num sistema de Integração Contínua:

- a) As *builds* que falham devem ser corrigidas de imediato.
- b) A realização de uma *build* deve ser rápida e, por isso, excluir testes de aceitação.
- c) Os testes devem ser feitos em ambientes específicos, que “clonam” as condições de produção.
- d) Todos os programadores devem fazer entrega de código para o repositório partilhado, com regularidade (e.g.: pelo menos, diariamente).
- e) Todos os membros da equipa têm acesso imediato ao *feedback* do estado das *builds*.

#### P9.

O que é o *pipeline* de Entrega Contínua (*continuous delivery*)?

- a) É um ficheiro de configuração que deve acompanhar o repositório Git, de modo a que as novas entregas sejam detetadas de imediato.
- b) É uma implementação automatizada do processo de compilação, montagem, teste e instalação de uma aplicação.
- c) É um processo automático para instalar uma aplicação num ambiente de Cloud (e.g.: usando *containers*).
- d) É uma visualização possível da execução dos projetos (*jobs*) configurados no Jenkins, considerando diferentes etapas na construção (*stages*), que dependentes do sucesso das antecedentes.
- e) É a configuração das regras de qualidade (do código) e do nível de cobertura necessários para que a *build* passe com sucesso.

#### P10.

Qual a hierarquia de elementos necessária na escrita de um *pipeline* declarativo do Jenkins, com inclusão de testes unitários?

- a) Pipeline, agent, stages, stage, steps
- b) Node, agent, stages, stage, steps, step
- c) Pipeline, docker, stages, stage, post
- d) Pipeline, agent, stage, junit, post
- e) Node, stage, test, deploy

#### P11.

Considere a implementação da *story* de login, com sucesso, quando o utilizador já se encontra registado. O interface é baseado numa página web e acede aos serviços de retaguarda, invocando uma API. Que testes podem ser úteis, neste contexto?

- a) Testes unitários, para validar se a interação com a página web envolvida.

- b) Testes de regressão, para considerar os dados de autenticação das tentativas de acesso anteriores e encontrar vulnerabilidades.
- c) Testes de sistema, para confirmar o comportamento do serviço de autenticação, em situações de utilização intensiva.
- d) Testes de aceitação, para confirmar a conformidade das passwords com as regras de complexidade mínima definidas.
- e) Teste unitário, para verificar o contrato dos métodos de cifra e validação de passwords.

#### P12.

Considere o quadro de apoio à decisão da Figura 2. Tendo em conta os compromissos que o gestor da qualidade deve praticar, considera que o projeto avaliado pode seguir para produção?

- a) Sim. O “*quality gate*” definido (por omissão) está a passar e isso indica que o código não oferece problemas.
- b) Sim. Os indicadores de *maintainability* são claramente positivos.
- c) Sim. Apesar de existirem 15 ocorrências construções que levantam dúvidas, podem ser resolvidas em 15 minutos.
- d) Não. Existem vulnerabilidades de segurança graves e ainda não resolvidas.
- e) Não. Existe código duplicado em 4 blocos e tem de ser resolvido para melhorar a *maintainability* do produto.

#### P13.

Os testes funcionais implementados com recurso a Selenium/WebDriver podem recorrer ao padrão “page object model” (POM). Qual das seguintes afirmação não é um benefício aplicável à utilização desse padrão?

- a) O padrão POM permite escrever testes funcionais muito mais legíveis (do que na utilização “normal” dos *scripts* de teste com Selenium).
- b) O padrão POM permite reduzir a duplicação de código e separar a navegação (entre páginas) da verificação.
- c) Os testes ficam mais focados e curtos, já que podemos reutilizar classes (que representam páginas) em diferentes testes.
- d) As alterações no UI são facilmente implementadas, pois a manutenção necessária dos testes está bem compartimentada no modelo (cada página tem a sua classe associada).
- e) o padrão POM é diretamente suportado pelos métodos do WebDriver, que automatiza a instanciação dos objetos (“*page objects*”), à medida que são necessários no teste.

#### P14.

Quais as regras que deve observar uma equipa que adotou o “GitHub flow”, na utilização do repositório partilhado?

- a) Trabalhar sobre o *branch* “master”, com *commits* frequentes; manter um *branch* adicional para registar as releases do produto.
- b) Criar um novo *branch* para cada programador; fazer alterações no seu *branch* “privado”; integrar os incrementos no *master* partilhado.
- c) Criar um *branch* para cada nova *feature*; fazer alterações neste *branch*, com *commits* regulares; integrar o incremento no *master*, mediante um pedido de integração (“*pull request*”).
- d) Criar um novo *branch* por *feature*; desenvolver a *feature* no respetivo *branch*; integrar no *master*; manter um *branch* adicional para as *releases* em separado.
- e) Criar um novo *branch* local para cada *feature*; adicionar as alterações; integrar no *master* partilhado para revisão pelos pares.

#### P15.

Considere o trecho de código apresentado na Figura 2. Qual das seguintes práticas, aceitáveis para o tratamento de exceções, pode ser observada:

- a) Não ignorar/suprimir as exceções.
- b) Evitar a utilização da exceções genéricas, usando blocos “try...catch” adequados.
- c) Adicionar contexto à exceção e lançar as exceções no nível de abstração adequado.
- d) Registrar a condição de exceção num “log” da aplicação.
- e) Tratar a exceção no nível adequado da hierarquia de invocação dos objetos.

#### P16.

Qual o ciclo característico de uma abordagem TDD?

- a) Adicionar o novo incremento; escrever os testes unitários que o verificam; executar todos os testes; melhorar o código, se necessário.
- b) Adicionar os testes relativos à *user story* em mãos; executar os testes e confirmar que falham; implementar o código necessário para fazer passar os testes; rever e aceitar o incremento.
- c) Adicionar um teste; executar todos os testes e ver o novo a falhar; fazer as alterações necessárias para o teste passar; correr todos os testes e confirmar que passam; rever o código (*refactoring*).
- d) Limpar o código anterior; adicionar um novo teste; implementar o código necessário e submeter no repositório partilhado; observar o *feedback* do sistema de CI.
- e) Escrever os testes unitários no início da interação; implementar o código necessário para fazer passar os testes; escrever os testes de integração; fazer *refactoring* do código na medida necessária.

**P17.**

A existência de código duplicado indíca, geralmente, uma má prática. Que refactoring seria INADEQUADO para resolver este *code smell*?

- os blocos de código duplicados ocorrem dentro da mesma classe: introduzir uma constante e atribuir-lhe o bloco de código comum.
- o mesmo código é encontrado em duas subclasses do mesmo nível: extrair para um método, e colocá-lo num nível de hierarquia acima.
- o mesmo código é encontrado em duas subclasses do mesmo nível: se o código duplicado estiver dentro de um construtor, colocar a parte comum num construtor num nível acima.
- se o código duplicado for encontrado em duas classes diferentes, extrair o bloco para uma superclasse nova e as classes mantêm as funcionalidades anteriores.
- se o código duplicado for encontrado em duas classes diferentes, mas não for natural criar uma única superclasse, criar uma classe adicional e mover para lá o bloco comum.

**P18.**

A utilização de ambientes de *mocking* ajuda na utilização de objetos sintetizados, em substituição de objetos/serviços reais. Qual das afirmações NÃO é uma vantagem atribuível a estes ambientes?

- Capacidade de retornar valores extremos, para exercitar condições limite.
- Manter os testes unitários rápidos, isolados de potenciais latências (de serviços necessários).
- Introduzir previsibilidade no resposta de um serviço remoto;
- Fornecer uma implementação simplificada de um módulo atribuído a outro programador;
- Facilitar a preparação das condições necessárias para o teste, através da definições das espetativas.

**P19.**

Os teste unitários são escritos pelo programador; NÃO são úteis para o ajudar a:

- entender o contrato do módulo (requisitos do que vai construir);
- escrever menos código;
- documentar a utilização pretendida de um componente;
- prevenir erros de regressão;
- aumentar a confiança no código.

**P20.**

Qual a interpretação mais adequada da conceito “dívida técnica” (*technical debt*)?

- É a diferença do nível de cobertura atual para os 100% de cobertura.
- É o número de erros encontrados num projeto.
- É o número de *user stories* não aceites na iteração corrente, por não passarem os testes definidos.
- É uma estimativa do tempo necessário para correr os problemas encontrados durante a análise estática.
- É uma estimativa do tempo de trabalho necessário para fazer passar os testes que estão a falhar.

**P21. [Opcional]**

(Esta questão é facultativa: não é necessária para obter a cotação completa do exame, mas pode somar até 1 valor adicional no grupo de escolha múltipla.)

Considere o techo de código apresentado na Figura 2. Que “vulnerabilidades” podem ser observadas (tal como definidas no analisador SonarQube)? Que *refactoring* deveria ser feito para as resolver (caso existam)?

**P22. [desenvolvimento]**

Considere uma aplicação para a gestão de informação de funcionários (Employee), implementada de acordo com os padrões comuns do *framework* Spring Boot, estruturada de acordo com o diagrama da Figura 3.

- Explique que testes devem ser feitos para verificar o comportamento dos vários componentes (i.e., o que é relevante testar?). Explique também, para cada situação de teste referida, qual a estratégia para montar/estruturar os respetivos testes, tendo em conta o suporte do *framework*; seja específico.

Pode, se assim entender, organizar a resposta numa grelha:

O que deve ser objeto de testes?	Estratégia para testar (“como” ?)
...	...

- Exemplifique, em código, um teste de integração para verificação da API do serviço `EmployeeRestController`. Documente as suposições que fizer; o código não precisa de estar compilável, mas deve captar as construções-chave para verificar o serviço.

## 45426: TESTE E QUALIDADE DE SOFTWARE | 2018-19, 2º Sem.

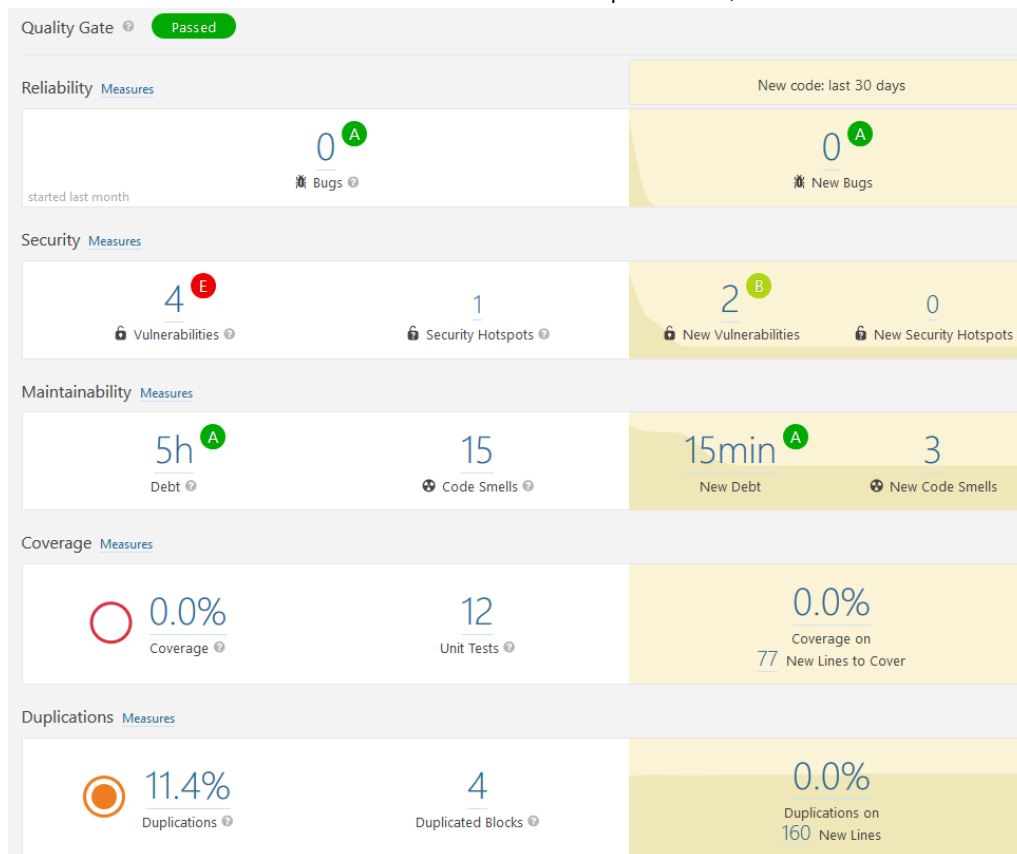


Figura 1: Dashboard do SonarQube.

```

6      public void queryCustomer(javax.servlet.ServletRequest request) {
7          try {
8              Connection connection = DriverManager
9                  .getConnection( url: "jdbc:mysql://localhost:3306/JDBCDemo",
10                               user: "root", password: "secret");
11
12              String query = "SELECT account_balance FROM user_data WHERE user_name = "
13                  + request.getParameter( s: "customerName");
14
15              Statement statement = connection.createStatement();
16              ResultSet results = statement.executeQuery(query);
17
18              doSomethingWithTheseResults( results);
19
20          } catch (Exception e) {
21              e.printStackTrace();
22          }
23      }

```

Figura 2: Código relativo ao método queryCustomer.

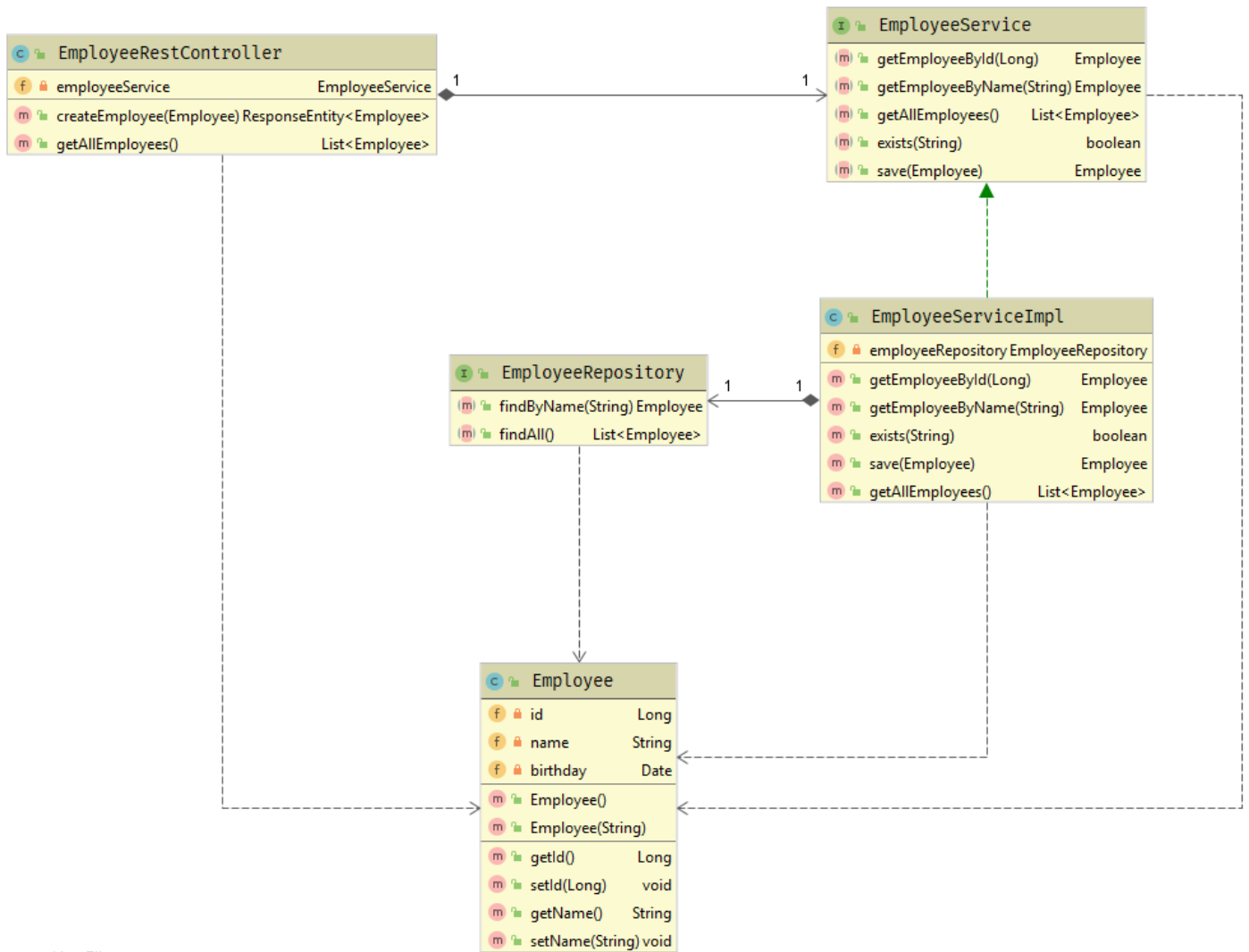


Figura 3: Diagrama de classes, parcial, de uma aplicação Spring Boot para gestão de informação de funcionários.





