Exame de Época Normal – 2018-06-21 15h00. Duração: 90 min (+10 tolerância).

Responda de forma sucinta, mas completa, às perguntas colocadas, em folha devidamente identificada.

Q1.

O planeamento de um projeto orientado por histórias de utilização (*user stories*) permite à equipa criar especificações recorrendo a um discurso centrado na área da aplicação, adequadas para alimentar testes de aceitação e suportar o seguimento/controlo do projeto.

A Figura 1 mostra a evolução de um projeto ao longo de uma iteração de 2 semanas.

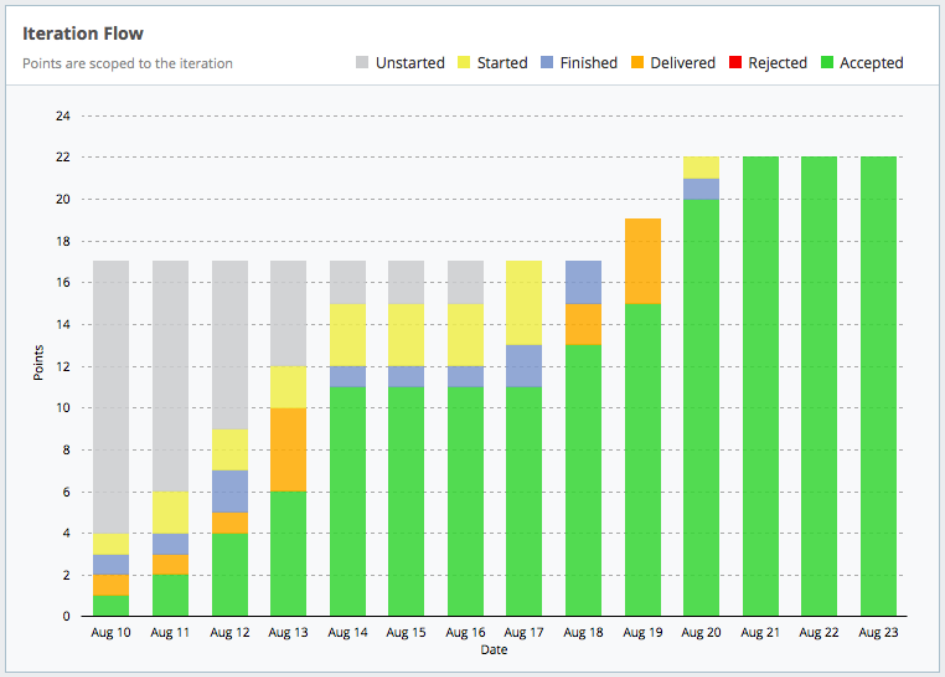

Figura 1: Pontos realizados ao longo de uma iteração.

a) Como é que as histórias podem ser usadas como "especificações executáveis" na automação de testes, ligando os requisitos aos testes?

b) A figura ilustra alguns estados possíveis para as histórias do produto. Que mapeamento pode haver entre estes estados e a organização do fluxo de trabalho de desenvolvimento com o Git (*Git workflows*)?

Q2.

Considere a afirmação: "[No nosso projeto] os testes são realizados por alguém que não tenha desenvolvido a classe que está a ser testada e seguem a política de caixa aberta, segunda a perspetiva do *developer*."

a) Que tipo de testes seguem uma estratégia de "caixa aberta"? Justifique.
b) Tendo em conta que a frase consta no Manual de Qualidade de um dos projetos de grupo da disciplina, também recomendaria a adoção das práticas descritas, para o seu projeto? Justifique.

Q3.

Considere que está a participar numa sessão de revisão de código e que está a analisar os elementos representados (Figura 2).

a) Tendo em consideração as classes do diagrama: que tipo de alterações estruturais recomendaria (*refactoring* do código), tendo em vista a facilidade de manutenção?
b) Que nível de cobertura (*code coverage*) seria aceitável para estas classes? Justique.

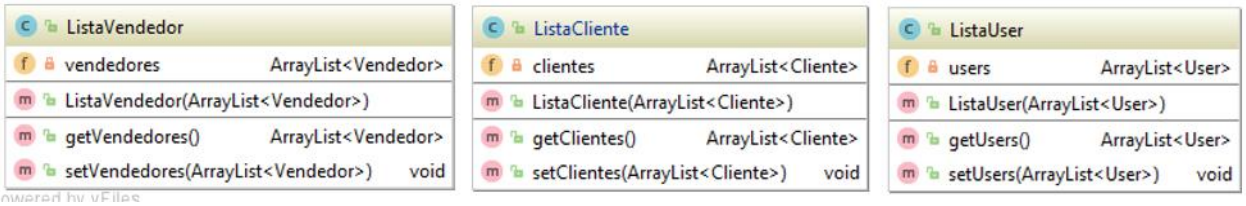

Figura 2: Diagrama de classes (parcial).

Q4.

O quadro junto (Figura 3) descreve a configuração de um patamar de qualidade (*quality gate*) no SonarQube.

a) Explique em que consistem pelo menos cinco das métricas da lista (definição contextualizada).
b) Avalie criticamente a presente configuração (é adequada? com que pressupostos? que pontos deviam ser mudados?...).

Metric	Over Leak Period	Operator	Warning	Error
Code Smells	Yes	is greater than	20	60
Condition Coverage on New Code	Always	is less than	20.0%	20.0%
Coverage	Yes	is less than	20.0%	20.0%
Duplicated Lines (%)	No	is greater than	3.0%	10.0%
Maintainability Rating on New Code	Always	is worse than	A	B
Reliability Rating on New Code	Always	is worse than	A	B
Security Rating on New Code	Always	is worse than	C	C
Vulnerabilities	No	is greater than	15	20

Figura 3: Quadro de configuração do patamar de qualidade ("quality gate").

Q5.

"O *Page Object Model* facilita imenso na compreensão e manutenção dos testes funcionais." (citação de R. Lopes, palestra convidada em 2018/6/5).

Explique o sentido da afirmação, descrevendo os conceitos e ferramentas que lhe parecer relevantes neste contexto.

Q6.

Um sistema de integração continua realiza a construção automática do projeto de código (*builds*), incorporando verificações de qualidade, como seja a ativação de testes ou de análise estática do código.

O trecho junto apresenta o conteúdo do ficheiro Jenkinsfile para configuração de um processo de integração contínua num determinado projeto.

a) Descreva o processo de integração contínua configurado (o que faz).
b) Neste *pipeline*, há execução de testes? Justifique.

```
pipeline {
    agent any
    stages{
        stage('Build'){
            steps {
                sh 'mvn clean package'
            }
            post {
                success {
                    echo 'Now Archiving...'
                    archiveArtifacts artifacts: '**/target/*.war'
                }
            }
        }
        stage ('Deploy to Staging'){
            steps {
                build job: 'Deploy-to-staging'
            }
        }

        stage ('Deploy to Production'){
            steps{
                timeout(time:5, unit:'DAYS'){
                    input message:'Approve PRODUCTION Deployment?'
                }

                build job: 'Deploy-to-Prod'
            }
            post {
                success {
                    echo 'Code deployed to Production.'
                }

                failure {
                    echo ' Deployment failed.'
                }
            }
        }


    }
}
```

Q7.

O excerto de código seguinte apresenta um teste de integração para Java EE. Os testes apresentados verificam a disponibilidade de um recurso REST ("api/users").

a) Explique a relevância da anotação @RunAsClient nestes testes [linha 41].
b) Como é que o programador, nestes testes, poderia aceder diretamente a objetos em execução do lado do servidor (e.g.: obter uma instância do EntitityManager)?
c) É possível instruir o Arquillian para usar bases de dados específicas (e.g.: em memória) para diferentes pacotes de testes? Justifique e, se aplicável, com referência ao código apresentado.

```
40  @RunWith(Arquillian.class)
41  @RunAsClient
42  public class UserEndpointTest {
43      private static final User TEST_USER = new User("id", "last name", "first name", "login", "password");
44      private static String userId;
45      private Client client;
46      private WebTarget userTarget;
47
48      @ArquillianResource
49      private URI baseURL;
50
51      @Deployment(testable = false)
52      public static WebArchive createDeployment() {
53          File[] files = Maven.resolver().loadPomFromFile("pom.xml").importRuntimeDependencies()
54                  .resolve().withTransitivity().asFile();
55          return ShrinkWrap.create(WebArchive.class)
56                  .addClasses(User.class, UserEndpoint.class)
57                  .addClasses(PasswordUtils.class, KeyGenerator.class, SimpleKeyGenerator.class, LoggerProducer.cl
58                  .addAsResource("META-INF/persistence-test.xml", "META-INF/persistence.xml")
59                  .addAsWebInfResource(EmptyAsset.INSTANCE, "beans.xml")
60                  .addAsLibraries(files);
61      }
62      @Before
63      public void initWebTarget() {
64          client = ClientBuilder.newClient();
65          userTarget = client.target(baseURL).path("api/users");
66      }
67
68      @Test
69      @InSequence(1)
70      public void shouldGetAllUsers() throws Exception {
71          Response response = userTarget.request(APPLICATION_JSON_TYPE).get();
72          assertEquals(200, response.getStatus());
73      }
74
75      @Test
76      @InSequence(2)
77      public void shouldCreateUser() throws Exception {
78          Response response = userTarget.request(APPLICATION_JSON_TYPE).post(Entity.entity(TEST_USER, APPLICATION_
79          assertEquals(201, response.getStatus());
80          userId = getUserId(response);
81      }
82
```

Q8.

Considere que se pretende implementar testes unitários para uma classe Java (CurrencyConverter) que faz a conversão de valores de moeda. Para o fazer, a classe acede a um serviço na Internet (compatível com a interface RatesProvider) para obter as taxas de câmbio naquele momento.

a) Exemplifique, em código, a escrita de testes unitários para verificar o contrato dos métodos CurrencyConverter#convert() e CurrencyConverter#convertForeingToDomestic().
b) Como é que o padrão de desenho "Inversion of Control - IoC", que se depreende dos construtores, pode ajudar na transposição dos testes unitários (pedidos) para a escrita de testes de integração?

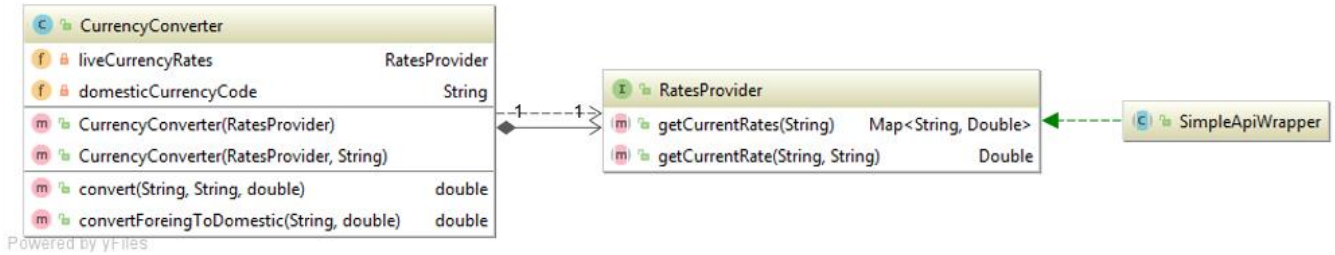

Figura 4: Classe CurrencyControl e dependências.

convert

```
public double convert(String from,
                      String to,
                      double value)
               throws IllegalArgumentException
```

Parameters:

from - base currency code

to - target currency code

value - the amount, in the base currency

Returns:

the amount in the target currency

Throws:

IllegalArgumentException - if the currencies are unknown

convertForeingToDomestic

```
public double convertForeingToDomestic(String from,
                                       double value)
                                throws IllegalArgumentException,
                                       IllegalStateException
```

Parameters:

from - base currency code

value - the amount, in the foreing currency

Returns:

the amount in the domestic currency

Throws:

IllegalArgumentException - unknown currency

IllegalStateException - the home currency is not defined yet

Constructor Detail

CurrencyConverter

```
public CurrencyConverter(RatesProvider liveCurrencyRates)
```

Parameters:

liveCurrencyRates - the exchange rates lookup service

CurrencyConverter

```
public CurrencyConverter(RatesProvider liveCurrencyRates,
                         String domestic)
                  throws IllegalArgumentException
```

Parameters:

liveCurrencyRates - the exchange rates lookup service

domestic - the currency code for the domestic currency

Throws:

IllegalArgumentException

Figura 5: Vista parcial do "javadoc" da classe CurrencyControl.