Fatec São José dos Campos Prof. Jessen Vidal

10 anos 2006-2016

Centro Paula Souza

GOVERNO DO ESTADO DE SÃO PAULO

Curso Superior de Banco de Dados

Disciplina: Laboratório em Desenvolvimento de Banco de Dados V

Prof. Emanuel Mineda Carneiro

emanuel.mineda@fatec.sp.gov.br

São José dos Campos - SP

Roteiro

- Teste de Unidade
- Cobertura de Código
- Classes de Equivalência
- *Mock Objects*
- JUnit

Teste de Unidade

Teste de Unidade

- Testa uma unidade de software (método) individualmente
  - Automatizado
  - Caixa Branca – Quando criado para garantir o funcionamento de uma unidade de software existente, cujo código é conhecido
  - Caixa Preta – Quando criado antes do desenvolvimento, de forma a guiar o desenvolvimento da unidade de software
    - *Test Driven Development* (TDD)
- **Independente** – Um teste de unidade deve executar de forma independente, sem depender do resultado de outros testes de unidade
- Cada teste de unidade valida uma única funcionalidade
  - Isso garante uma fácil identificação da razão pela falha de um teste
  - Uma unidade de software pode ter vários testes de unidade associados, dependendo de sua complexidade (quantidade de regras de negócio associadas)

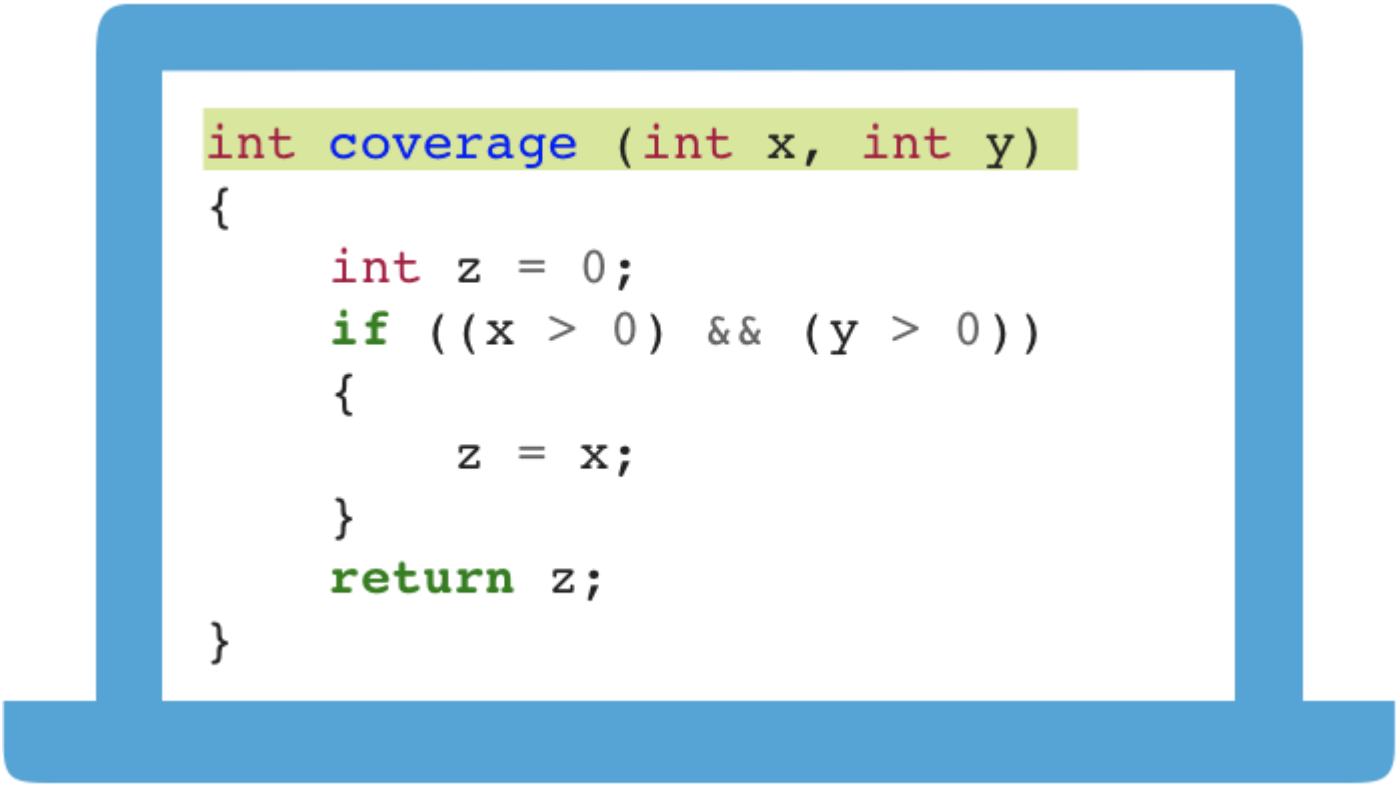
Cobertura de Código

Cobertura de Código

- A cobertura código é uma medida que indica a quantidade de código explorada pelos testes
- Tipo de cobertura de código:
  - **Função** – Verifica quantidade de funções/métodos testados
  - **Instrução** – Verifica quantidade de instruções testadas
  - **Ramificação –** Verifica se todas as ramificações geradas por estruturas de controle (if, else, switch case, for, while, etc) foram testadas
  - **Condição** – Verifica se todas as comparações (expressões booleanas) foram testadas

Cobertura de Código

- Função:

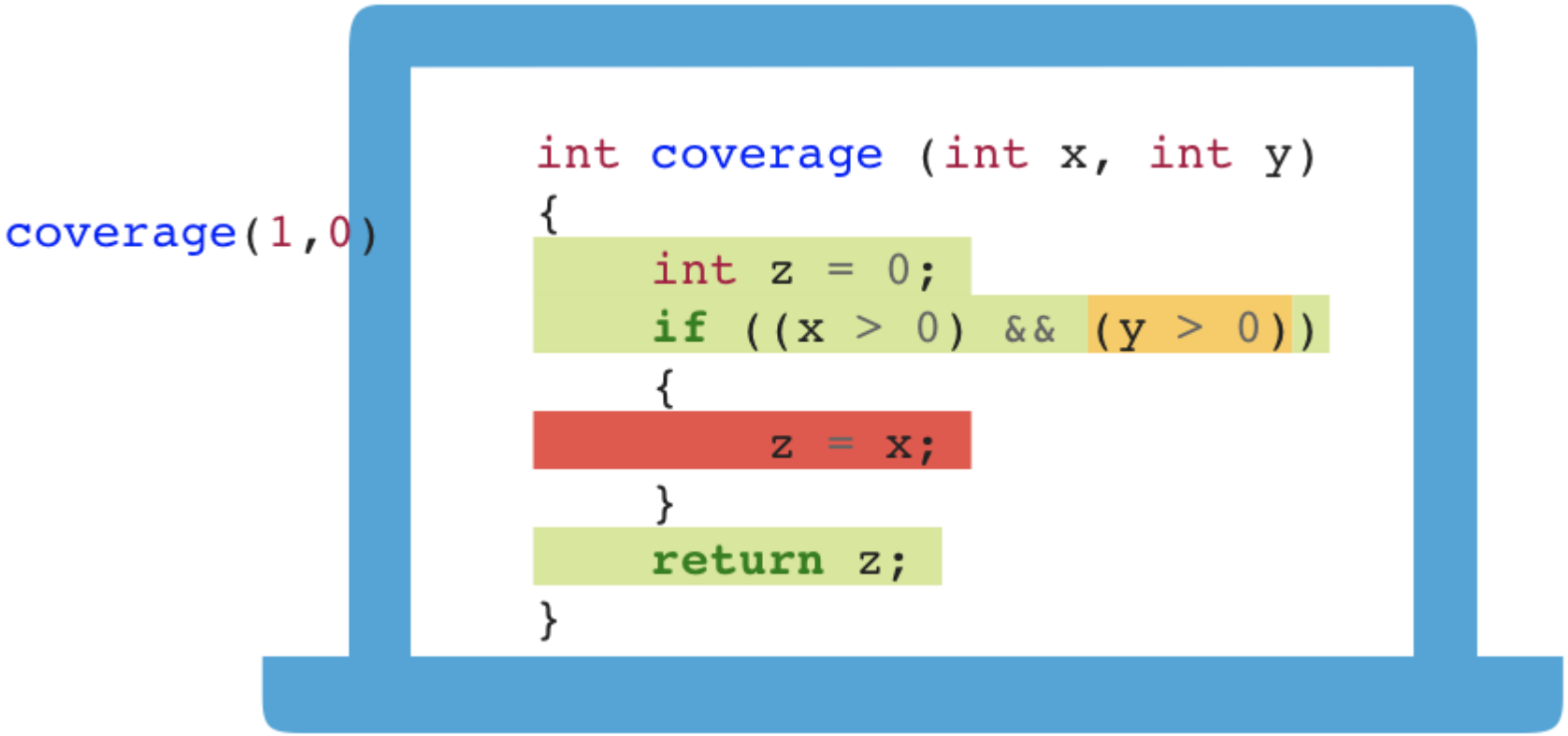


```
int coverage (int x, int y)
{
    int z = 0;
    if ((x > 0) && (y > 0))
    {
        z = x;
    }
    return z;
}
```

Fonte: Candido (2019)

Cobertura de Código

- Instrução:



Fonte: Candido (2019)

Cobertura de Código

- Ramificação:

coverage(1,1)

```
int coverage (int x, int y)
{
    int z = 0;
    if ((x > 0) && (y > 0))
    {
        z = x;
    }
    return z;
}
```

Fonte: Candido (2019)

Cobertura de Código

- Condição:

coverage(1,0)

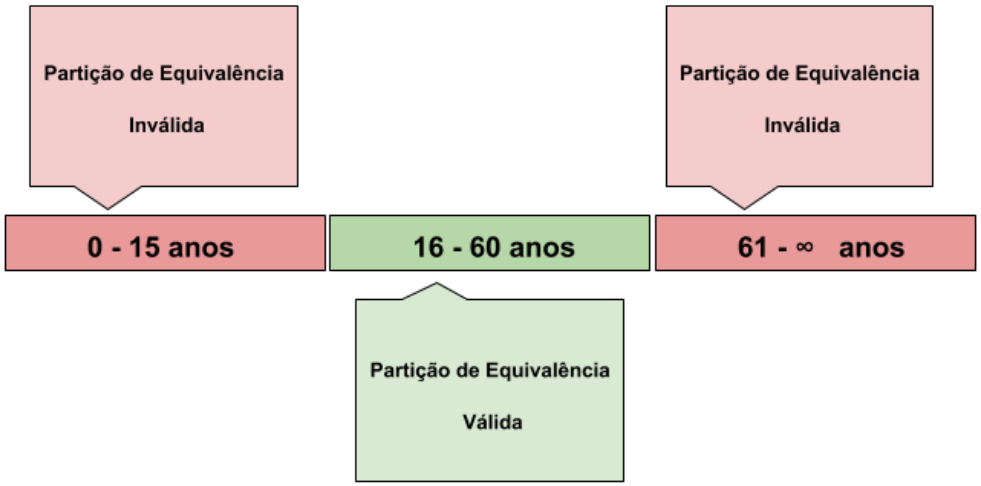
```
int coverage (int x, int y)
{
    int z = 0;
    if ((x > 0) && (y > 0))
    {
        z = x;
    }
    return z;
}
```

Fonte: Candido (2019)

Classes de Equivalência

Classes de Equivalência

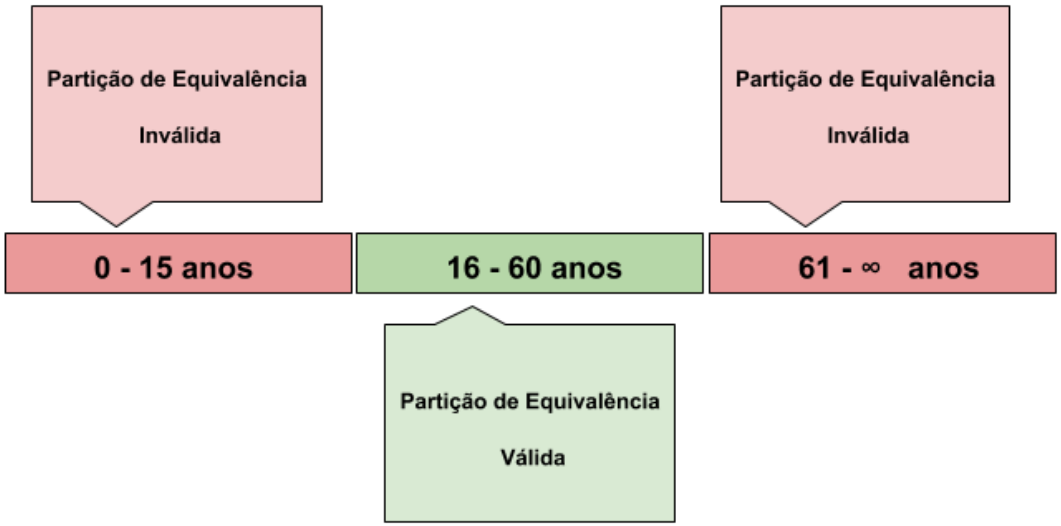
- A técnica de classes de equivalência divide os valores de um parâmetro numérico em grupos, de forma a diminuir a quantidade de testes necessários
  - **Exemplo:** Imagine uma aplicação para cadastro de candidatos a vagas de emprego. Nesta aplicação, foi definido que candidatos com idade inferior a 16 anos ou superior a 60 anos não prosseguirão no processo seletivo
    - Neste caso, em vez de testarmos todos os valores possíveis de idade, bastaria testar com valores que representem cada grupo/classe



Fonte: Venture (2020)

Classes de Equivalência

- A técnica de **análise do valor de limite** é complementar à de classes de equivalência e indica o uso para testes dos valores imediatamente inferiores e superiores aos limites de cada classe
  - **No exemplo:** {-1, 0, 1, 14, 15, 16, 17, 59, 60, 61, 62}



Fonte: Venture (2020)

Fatec São José dos Campos Prof. Jessen Vidal

Centro Paula Souza

GOVERNO DO ESTADO DE SÃO PAULO

Mock Objects

Mock Objects

- O termo *Mock Object* se refere a elementos que imitam objetos reais em testes de unidade
  - Ajudam no isolamento do teste
    - Pressupõe-se que as unidades de teste externas chamadas se encontram devidamente testadas por seus próprios testes de unidade
    - Uma unidade de software com defeito não afetas testes de outras unidades de software que a utilizam

Atividade

Atividade

- Com base no projeto https://github.com/mineda/projetolabv
- Planeje os testes necessários para:
  - Método “save” de UsuarioRepository
  - Método “novoUsuario(Usuario usuario)” de “SegurancaService”

Fatec São José dos Campos Prof. Jessen Vidal

Centro Paula Souza

GOVERNO DO ESTADO DE SÃO PAULO

JUnit

JUnit

- O JUnit é um framework para criação de testes automatizados com a linguagem Java
  - A dependência para o JUnit é incluída automaticamente em projetos Spring Boot
  - Os testes ficam em diretórios localizados em “src/test/java” que espelham a estrutura da aplicação, localizada em “src/main/java”
  - Uma classe de teste deve ser criada para cada uma das classes da aplicação a ser testada
    - Por padrão a classe de teste tem o mesmo nome da classe a ser testada, com a adição da do sufixo “Test” ao final
      - Exemplo: “UsuarioServiceTest.java” seria a classe de teste para “UsuarioService.java”
  - O Maven executa, por padrão, os testes de qualquer classe cujo nome termine em “Test” ou “Tests”

JUnit

- Exemplo de classe de teste:

```
@SpringBootTest
public class SegurancaServiceTest {

    @Autowired
    private SegurancaService service;

    @MockBean
    private UsuarioRepository usuarioRepo;

    @BeforeEach
    public void setUp() {
        Usuario usuario = new Usuario();
        usuario.setId(1L);
        usuario.setNome("Teste");
        usuario.setSenha("Senha");
        Optional<Usuario> usuarioOp = Optional.of(usuario);
        Mockito.when(usuarioRepo.findById(1L)).thenReturn(usuarioOp);
    }

    @Test
    public void buscarUsuarioPorIdTestOk() {
        assertEquals("Teste", service.buscarUsuarioPorId(1L).getNome());
    }
}
```

JUnit

- Toda classe de teste no Spring Boot deve ser anotada com @SpringBootTest

```
@SpringBootTest
public class SegurancaServiceTest {
```

- A classe a ser testada deve ser injetada por meio da anotação @Autowired

```
    @Autowired
    private SegurancaService service;
```

- Outras classes acessadas pela classe a ser testada devem ser "mockadas" por meio de @MockBean

```
    @MockBean
    private UsuarioRepository usuarioRepo;
```

JUnit

- Um método anotado com @BeforeEach terá seu conteúdo executado antes de cada teste
  - Neste exemplo usamos esse método para definir como vai ser o comportamento de um dos métodos (findById) de nosso repositório "mockado" UsuarioRepository
    - **Quando o método findById é chamado com o parâmetro 1L, será retornado um Optional contendo o usuário { "id": 1, "nome": "Teste", "senha": "Senha" }**
      - **Chamadas com qualquer outro parâmetro resultarão em um Optional vazio**

```
@BeforeEach
public void setUp() {
    Usuario usuario = new Usuario();
    usuario.setId(1L);
    usuario.setNome("Teste");
    usuario.setSenha("Senha");
    Optional<Usuario> usuarioOp = Optional.of(usuario);
    Mockito.when(usuarioRepo.findById(1L)).thenReturn(usuarioOp);
}
```

    - **Podemos utilizar o método any() para indicar que o método deve aceitar qualquer valor de parâmetro**

```
Mockito.when(usuarioRepo.findById(any())).thenReturn(usuarioOp);
```

JUnit

- Um método de teste deve ser anotado com @Test. O comparação que define o teste deve ser realizada por meio de um *assert*
  - Todos os *asserts* ficam disponíveis com o seguinte import:
    - import static org.junit.jupiter.api.Assertions.*;
    - *assertEquals* verifica se o valor esperado (“Teste”) é igual ao retorno do método

```
@Test
public void buscarUsuarioPorIdTestOk() {
    assertEquals("Teste", service.buscarUsuarioPorId(1L).getNome());
}
```

  - A criação dos “mocks” pode ser realizada diretamente no método, em vez de usar um @BeforeEach

```
@Test
public void buscarUsuarioPorIdTestOk() {
    Usuario usuario = new Usuario();
    usuario.setId(1L);
    usuario.setNome("Teste");
    usuario.setSenha("Senha");
    Optional<Usuario> usuarioOp = Optional.of(usuario);
    Mockito.when(usuarioRepo.findById(1L)).thenReturn(usuarioOp);

    assertEquals("Teste", service.buscarUsuarioPorId(1L).getNome());
}
```

Junit – Teste de Controller

- Para realizar teste de unidade em controller, a classe de teste precisa receber uma anotação diferente, @WebMvcTest, que recebe como parâmetro a classe a ser testada
  - Essa anotação faz com que somente o controller seja executado, sem necessidade de inicializar o restante do sistema

```
@WebMvcTest(UsuarioController.class)
public class UsuarioControllerTest {
```

  - Para testar um controller se faz necessário realizar requisições e, para isso, fazemos uso do MockMvc, que injetamos com o @Autowired

```
    @Autowired
    private MockMvc mvc;
```

  - As demais dependências são "mockadas" normalmente com o @MockBean

```
    @MockBean
    private SegurancaService service;
```

Junit – Teste de Controller

- Para realizar uma requisição usamos o método "perform" do MockMvc
  - O método "post" faz uma requisição POST, assim como o método "get" faz uma requisição GET, etc
    - Para a URL, usamos o endereço definido em @RequestMapping, no controller
    - O JSON passado como parâmetro deve ser informado no método "content". Use "\"" sempre que precisar de aspas duplas dentro de uma string
    - Por meio do método contentType, informe "MediaType.APPLICATION_JSON"

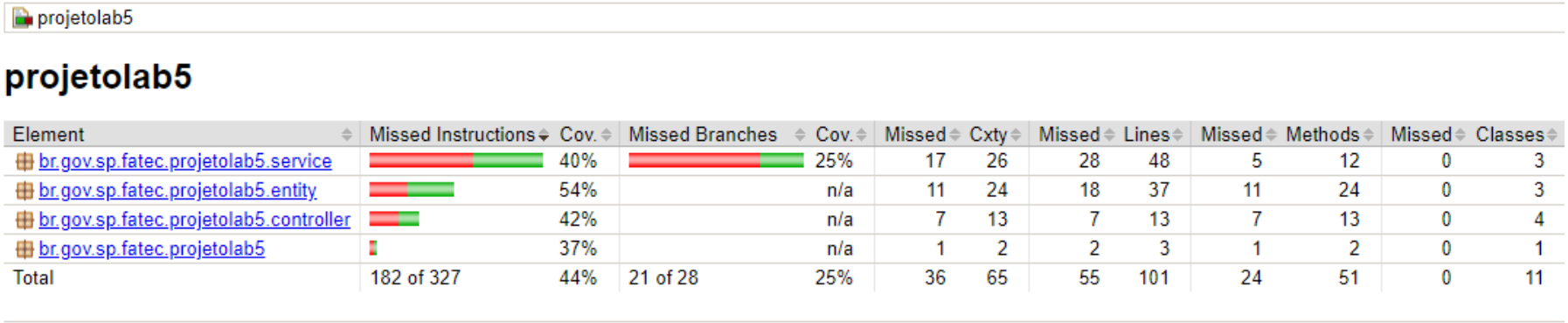
```
@Test
public void novoUsuarioTestOk() throws Exception {
    Usuario usuario = new Usuario("Teste", "senha");
    usuario.setId(1L);
    Mockito.when(service.novoUsuario(any())).thenReturn(usuario);

    mvc.perform(post("/usuario")
        .content("{\"nome\":\"TesteMvc\", \"senha\":\"senha\"}")
        .contentType(MediaType.APPLICATION_JSON))
        .andDo(print())
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.id").value(1L));
}
```

  - O método "andExpect" cuida dos testes
    - No exemplo verificamos se o status é 200 ("status().isOk()") e se no JSON retornado tem um atributo "id" com valor "1" ("jsonPath("$.id").value(1L)")

JaCoCo

JaCoCo

- O JaCoCo (Java Code Coverage) é uma ferramenta de aferição de cobertura de testes para Java
  - O comando “mvn test jacoco:report” executa os teste e gera um relatório, em forma de site, em “target/site/jacoco/index.html”



projetolab5

projetolab5

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
br.gov.sp.fatec.projetolab5.service		40%		25%	17	26	28	48	5	12	0	3
br.gov.sp.fatec.projetolab5.entity		54%		n/a	11	24	18	37	11	24	0	3
br.gov.sp.fatec.projetolab5.controller		42%		n/a	7	13	7	13	7	13	0	4
br.gov.sp.fatec.projetolab5		37%		n/a	1	2	2	3	1	2	0	1
Total	182 of 327	44%	21 of 28	25%	36	65	55	101	24	51	0	11

  - Dentro do VS Code o site pode ser aberto com a extensão “Live Server”

JaCoCo

- Para utilizá-lo é necessário configurar o plugin dentro da tag “build” do “pom.xml”:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>0.8.2</version>
            <executions>
                <execution>
                    <goals>
                        <goal>prepare-agent</goal>
                    </goals>
                </execution>
                <execution>
                    <id>report</id>
                    <phase>prepare-package</phase>
                    <goals>
                        <goal>report</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

JaCoCo

- Também configure a tag “reporting” dentro do “pom.xml”:

```
<reporting>
    <plugins>
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <reportSets>
                <reportSet>
                    <reports>
                        <!-- select non-aggregate reports -->
                        <report>report</report>
                    </reports>
                </reportSet>
            </reportSets>
        </plugin>
    </plugins>
</reporting>
```

Planejando um teste

Planejando um teste

- Para planejar um teste, precisamos, primeiramente, ver as condições a satisfazer para ter uma boa cobertura
  - Considere o seguinte código:

```
@Override
public Anotacao novaAnotacao(Anotacao anotacao) {
    if(anotacao.getTexto() == null ||
            anotacao.getTexto().isBlank() ||
            anotacao.getUsuario() == null ||
            anotacao.getUsuario().getId() == null) {
        throw new IllegalArgumentException("Anotação possui atributos em branco");
    }
    Usuario usuario;
    try {
        usuario = service.buscarUsuarioPorId(anotacao.getUsuario().getId());
    }
    catch(IllegalArgumentException exception) {
        throw new RuntimeException("Usuário não encontrado", exception);
    }
    if(anotacao.getDataHora() == null) {
        anotacao.setDataHora(new Date());
    }
    anotacao.setUsuario(usuario);
    return anotacaoRepo.save(anotacao);
}
```

Planejando um teste

- Para entrar no primeiro "if", seria necessário que:
  - O atributo "texto" de "anotacao" seja nulo
  - O atributo "texto" de "anotacao" contenha um texto em branco
  - O atributo "usuario" seja nulo
  - O "id" do "usuario" seja nulo

```
if(anotacao.getTexto() == null ||
        anotacao.getTexto().isBlank() ||
        anotacao.getUsuario() == null ||
        anotacao.getUsuario().getId() == null) {
    throw new IllegalArgumentException("Anotação possui atributos em branco");
}
```

  - Para qualquer um desses casos, o resultado esperado é uma exceção do tipo "IllegalArgumentException"

Planejando um teste

- O trecho seguinte também possui diferentes caminhos, dependendo do resultado de um try/catch. Ele entrará no trecho "catch" somente se a chamada ao método "service.buscarUsuarioPorId" gerar uma exceção do tipo "IllegalArgumentException"
  - Importante notar que, para esse código, o atributo "service" é do tipo "SegurancaService"
    - Por se tratar de um método externo ao que estamos testando, precisaremos fazer um "mock" que gera exceção "IllegalArgumentException" para o teste entrar no "catch"

```
try {
    usuario = service.buscarUsuarioPorId(anotacao.getUsuario().getId());
}
catch(IllegalArgumentException exception) {
    throw new RuntimeException("Usuário não encontrado", exception);
}
```

    - O resultado esperado seria "RuntimeException"

Planejando um teste

- Para entrar no segundo "if" basta que o atributo "dataHora" de "anotacao" seja nulo

```
if(anotacao.getDataHora() == null) {
    anotacao.setDataHora(new Date());
}
```

  - O resultado esperado independe da condição. A partir daqui o método sempre terá sucesso

Planejando um teste

- Resultados:
  - Analisando o método, teremos uma classe a testar “AnotacaoService” e duas classes com “mock”: “SecurancaService” e “AnotacaoRepository”

```
@Autowired
private AnotacaoService service;

@MockBean
private AnotacaoRepository anotacaoRepo;

@MockBean
private SegurancaService segurancaService;
```

Planejando um teste

- Resultados:
  - Teste com sucesso e data preenchida

```
@Test
   public void novaAnotacaoTestOk() {
       Usuario usuario = new Usuario();
       usuario.setId(1L);
       usuario.setNome("Teste");
       usuario.setSenha("abc123");
       Anotacao anotacao = new Anotacao();
       anotacao.setId(1L);
       anotacao.setTexto("Anotação teste");
       anotacao.setUsuario(usuario);
       anotacao.setDataHora(new Date());
       Mockito.when(segurancaService.buscarUsuarioPorId(1L)).thenReturn(usuario);
       Mockito.when(anotacaoRepo.save(any())).thenReturn(anotacao);
       assertEquals("Anotação teste", service.novaAnotacao(anotacao).getTexto());
   }
```

  - Fazemos “mock” de “segurancaService.buscarUsuarioPorId”, retornando um usuário válido, para não cair no “catch”
  - Também fazemos um “mock” do método “anotacaoRepo.save”, chamado ao final
  - O assert também poderia ser “assertNotThrows”, visto que estamos apenas comparando algo retornado pelo “mock”

Planejando um teste

- Resultados:
  - Teste com sucesso sem data preenchida

```
@Test
public void novaAnotacaoDataHoraNullTestOk() {
    Usuario usuario = new Usuario();
    usuario.setId(1L);
    usuario.setNome("Teste");
    usuario.setSenha("abc123");
    Anotacao anotacao = new Anotacao();
    anotacao.setId(1L);
    anotacao.setTexto("Anotação teste");
    anotacao.setUsuario(usuario);
    Mockito.when(segurancaService.buscarUsuarioPorId(1L)).thenReturn(usuario);
    Mockito.when(anotacaoRepo.save(any())).thenReturn(anotacao);
    assertEquals("Anotação teste", service.novaAnotacao(anotacao).getTexto());
}
```

    - Idêntico ao anterior, mas a data é gerada automaticamente

Planejando um teste

- Resultados:
  - Teste com erro por “texto” nulo

```
@Test
public void novaAnotacaoTextoNullTestNOk() {
    Usuario usuario = new Usuario();
    usuario.setId(1L);
    Anotacao anotacao = new Anotacao();
    anotacao.setId(1L);
    anotacao.setUsuario(usuario);
    anotacao.setDataHora(new Date());
    assertThrows(IllegalArgumentException.class, () -> {
        service.novaAnotacao(anotacao);
    });
}
```

  - Todos os atributos comparados são preenchidos, com exceção de “texto”
  - Não é necessário nenhum “mock”, pois o método finalizará antes de chegar nas chamadas
  - Usamos o “assertThrows”, pois o resultado esperado é uma exceção

Planejando um teste

- Resultados:
  - Teste com erro por “texto” em branco

```
@Test
public void novaAnotacaoTextoNullTestNOk() {
    Usuario usuario = new Usuario();
    usuario.setId(1L);
    Anotacao anotacao = new Anotacao();
    anotacao.setId(1L);
    anotacao.setTexto(" ");
    anotacao.setUsuario(usuario);
    anotacao.setDataHora(new Date());
    assertThrows(IllegalArgumentException.class, () -> {
        service.novaAnotacao(anotacao);
    });
}
```

  - Todos os atributos comparados são preenchidos, mas “texto” recebe uma string em branco
  - Não é necessário nenhum “mock”, pois o método finalizará antes de chegar nas chamadas
  - Usamos o “assertThrows”, pois o resultado esperado é uma exceção

Planejando um teste

- Resultados:
  - Teste com erro por “usuario” nulo

```
@Test
public void novaAnotacaoUsuarioNullTestNOk() {
    Anotacao anotacao = new Anotacao();
    anotacao.setId(1L);
    anotacao.setTexto("Anotação teste");
    anotacao.setDataHora(new Date());
    assertThrows(IllegalArgumentException.class, () -> {
        service.novaAnotacao(anotacao);
    });
}
```

    - Todos os atributos comparados são preenchidos, com exceção de "usuario"
      - Não criamos uma instância de "Usuario", visto que não vamos utilizá-la
    - Não é necessário nenhum “mock”, pois o método finalizará antes de chegar nas chamadas
    - Usamos o “assertThrows”, pois o resultado esperado é uma exceção

Planejando um teste

- Resultados:
  - Teste com erro por "id" de "usuario" nulo

```
@Test
public void novaAnotacaoUsuarioIdNullTestNOk() {
    Usuario usuario = new Usuario();
    Anotacao anotacao = new Anotacao();
    anotacao.setId(1L);
    anotacao.setTexto("Anotação teste");
    anotacao.setUsuario(usuario);
    anotacao.setDataHora(new Date());
    assertThrows(IllegalArgumentException.class, () -> {
        service.novaAnotacao(anotacao);
    });
}
```

  - Todos os atributos comparados são preenchidos, com exceção do "id" do "usuario"
  - Não é necessário nenhum “mock”, pois o método finalizará antes de chegar nas chamadas
  - Usamos o “assertThrows”, pois o resultado esperado é uma exceção

Planejando um teste

- Resultados:
  - Teste com erro por usuário não cadastrado

```
@Test
public void novaAnotacaoUsuarioNaoExisteTestNOk() {
    Usuario usuario = new Usuario();
    usuario.setId(1L);
    usuario.setNome("Teste");
    usuario.setSenha("abc123");
    Anotacao anotacao = new Anotacao();
    anotacao.setId(1L);
    anotacao.setTexto("Anotação teste");
    anotacao.setUsuario(usuario);
    anotacao.setDataHora(new Date());
    Mockito.when(segurancaService.buscarUsuarioPorId(1L))
        .thenThrow(IllegalArgumentException.class);
    assertThrows(RuntimeException.class, () -> {
        service.novaAnotacao(anotacao);
    });
}
```

    - Aqui realizamos um "mock" do método "buscarUsuarioPorId" de "segurancaService"
      - Forçamos o lançamento de uma exceção para cair no "catch"
    - Usamos o "assertThrows", pois o resultado esperado é uma exceção
      - A exceção lançada dentro do "catch" ("RuntimeException") é diferente da exceção lançada em nosso "mock" ("IllegalArgumentException")

Bibliografia

- **Candido, A.** Um pouco sobre cobertura de código e cobertura de testes. 2019. URL: https://medium.com/liferay-engineering-brazil/um-pouco-sobre-cobertura-de-c%C3%B3digo-e-cobertura-de-testes-4fd062e91007. Acessado em: 22/09/2022
- **Venture, S.** Técnica de Teste — Particionamento de Equivalência. 2020. URL: https://medium.com/revista-tspi/t%C3%A9cnica-de-teste-particionamento-de-equival%C3%AAncia-d32a7d689d82. Acessado em: 22/09/2022

Dúvidas?