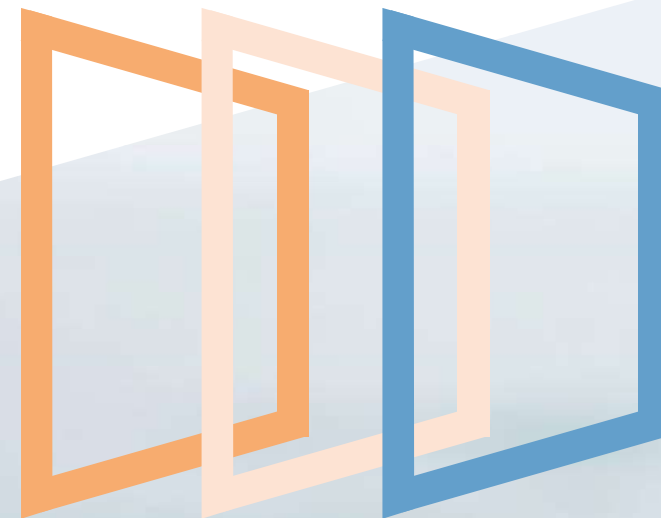


# Testes automatizados com Mockito em Java



minsaït

An Indra company

# Testes automatizados com Mockito em Java



**Instrutor:** Gleyser Bomfim Guimarães

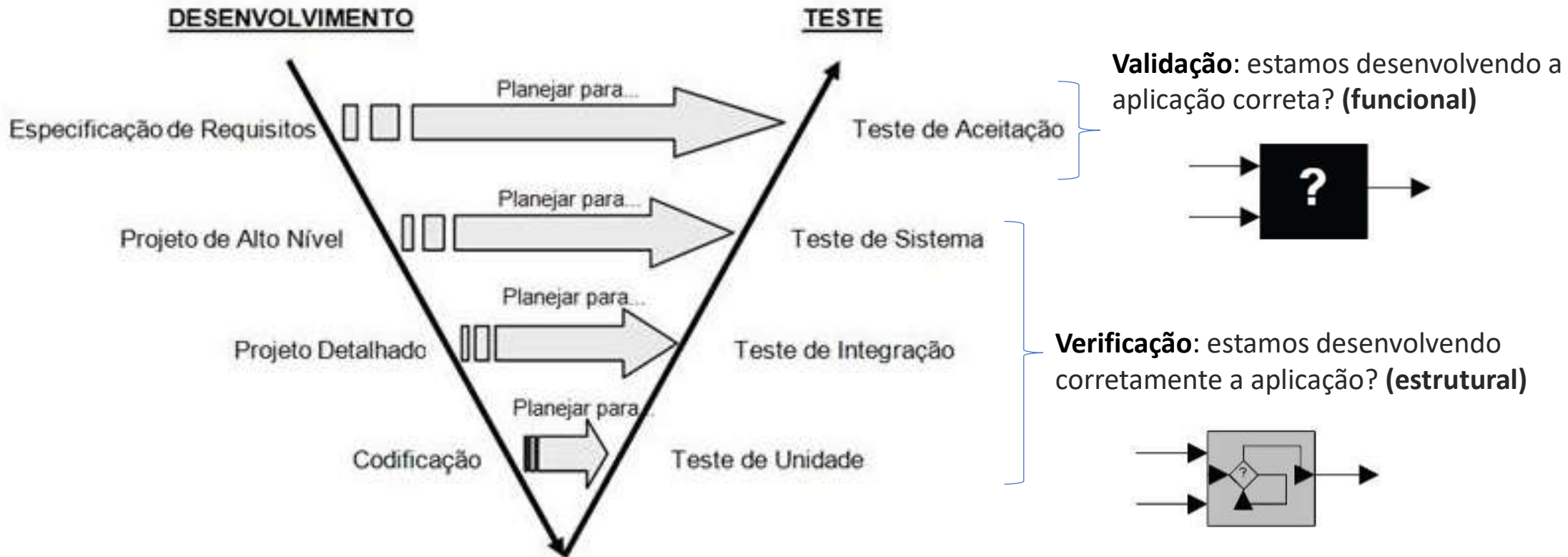
- Engenheiro de Software – minsait / Projeto Sicoob
- Bacharel e mestre em Ciência da Computação/UFCG



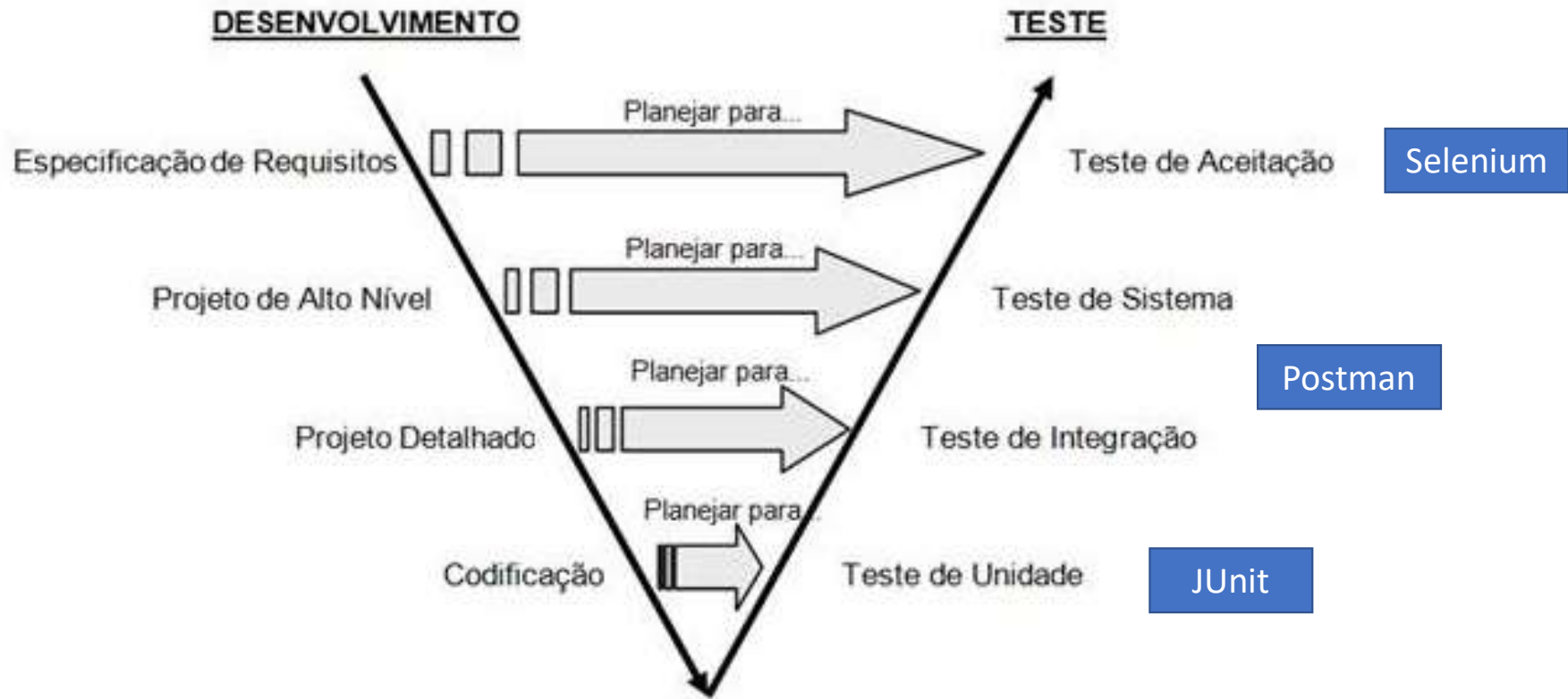
# Agenda de hoje!

- Processo de teste de software e Mocks
- Introdução ao Mockito
- Mockito: configurações iniciais
- Criação de Mocks e testes automatizados
- Definindo o comportamento dos mocks
- Exceções
- Capturando objetos com Mockito

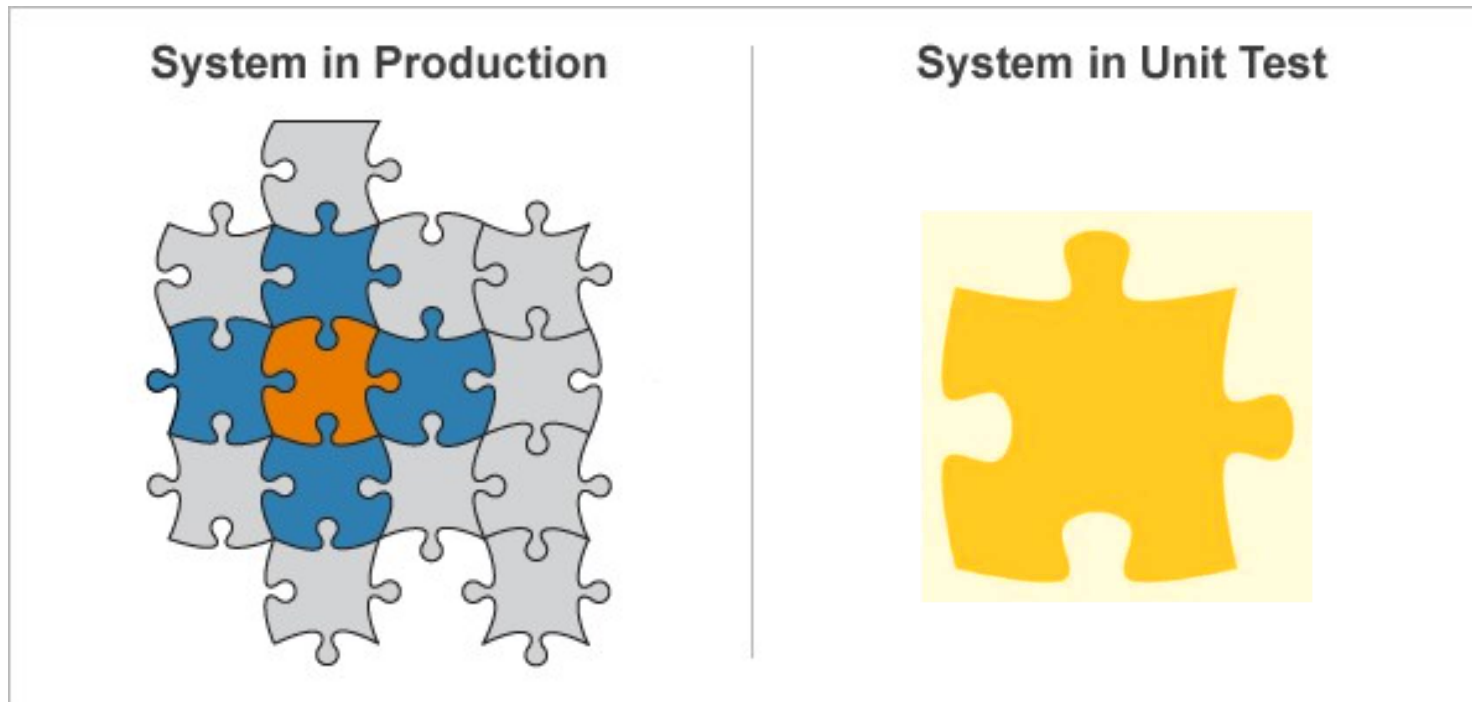
# Processo de teste de software



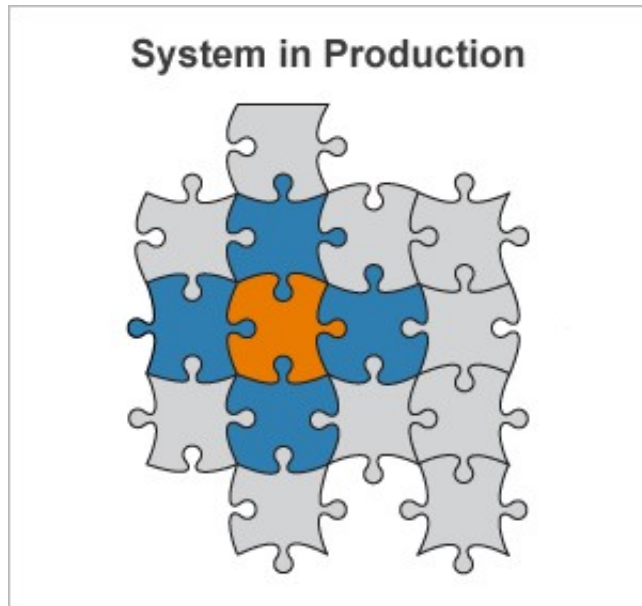
# Ferramentas



# Testes de unidade (testes unitários)



# Testes de unidade (testes unitários)



```
19  
20 @Service  
21 public class SondaService {  
22  
23     private final SondaRepository sondaRepository;  
24     private final MalhaService malhaService;  
25
```



Unidade a ser testada



Dependências que se associam  
por composição

No teste de unidade, o interesse é **verificar** a menor unidade possível (classes e métodos), **desconsiderando integrações** com os demais componentes. Nesse sentido, precisa-se isolar a unidade testada.

# Mocks

```
47 public class SolicitacaoDeHonraService {  
48  
49     private static final Logger LOGGER = LogManager.getLogger(SolicitacaoDeHonraService.class);  
50  
51     @Inject  
52     private HonraDAO honraDAO;  
53  
54     @Inject  
55     private MovimentacaoHonraDAO movimentacaoHonraDAO;  
56  
57     @Inject  
58     private ArquivoRemessaDAO arquivoRemessaDAO;  
59
```



Dependências que precisam ser *mockados*

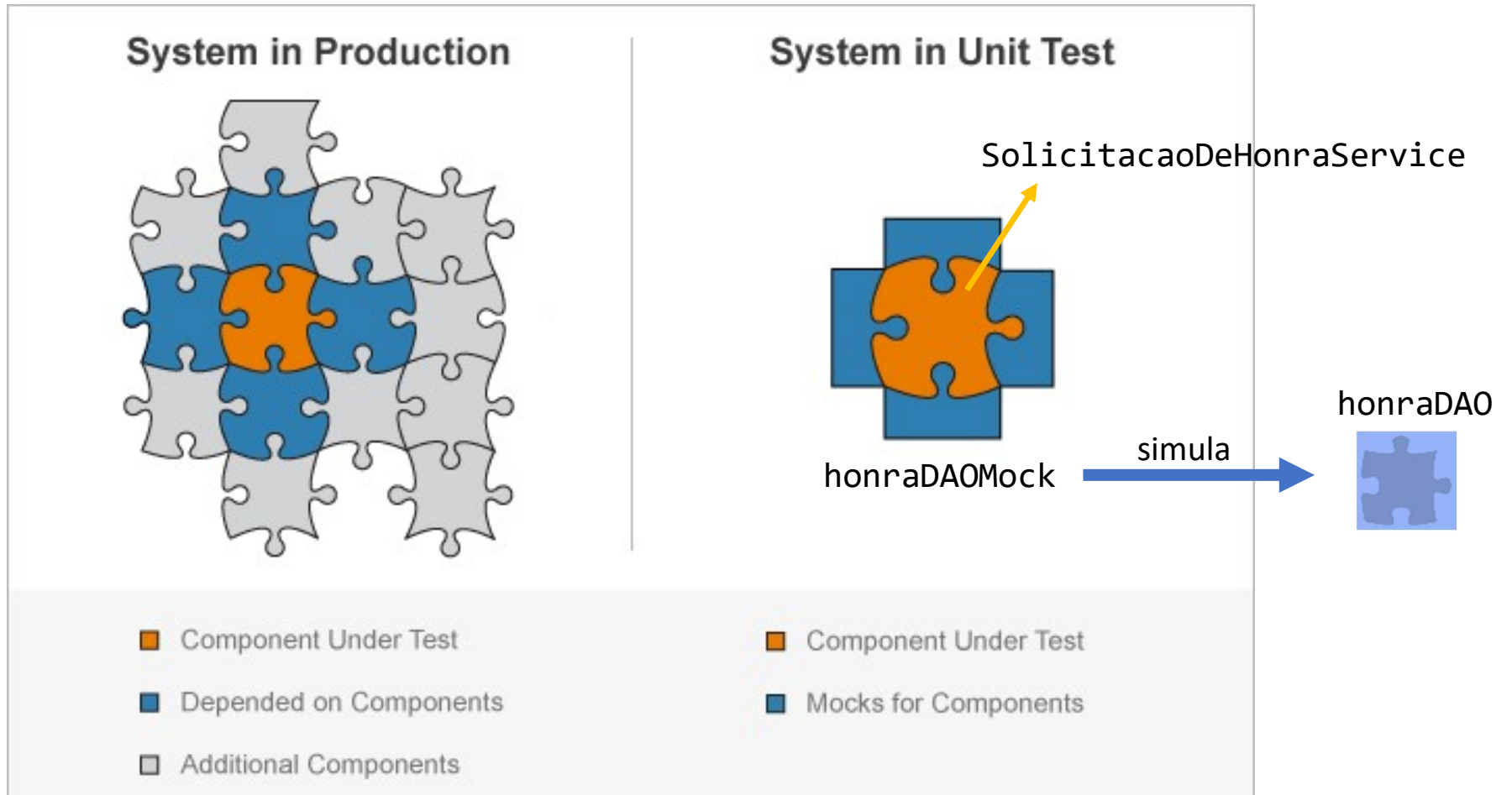
**Mocks são classes que simulam os comportamentos de outras classes funcionando como duplês para as classes de dependências;**

Nesse cenário, objetiva-se **testar as lógicas e os algoritmos de uma classe** que possui dependências de outra classe, mas **isolando essas dependências**.

Os Mocks **simulam comportamentos das dependências** de uma classe para que os testes de unidade não se tornem testes de integração;



# Mocks



# Introdução ao Mockito



- Framework para testes de unidade em Java;
- <https://site.mockito.org/> com a documentação e referências
- Como adicionar o Mockito na aplicação?
  - Baixar e adicionar o .jar
  - Como dependência pelo gerenciador de dependências (Maven, Gradle)

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

Does spring boot starter test include Mockito?

Anatomy of the Spring Boot Starter Test

This starter includes Spring-specific dependencies and dependencies for auto-configuration and a set of testing libraries. **This includes JUnit, Mockito, Hamcrest, AssertJ, JSONassert, and JsonPath.** 25 de mar. de 2020



Apache Maven

[maven.apache.org](https://maven.apache.org)

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>4.9.0</version>
</dependency>
```



Gradle Groovy DSL

[gradle.org](https://gradle.org)

```
implementation 'org.mockito:mockito-core:4.9.0'
```

# Criando Mocks

Após incluir o Mockito nas dependências do projeto, podemos criar os mocks. Para isso, comece analisando a unidade a ser testada:

- 1 - Identifique a unidade a ser testada (classe -> método)
- 2 - Identifique as dependências que são chamadas/utilizadas no método (menor unidade possível) a ser testada
- 3 - Identifique quais métodos dessas dependências são chamados/invocados e qual o comportamento
- 4 - Crie os mocks necessários
- 5 – Defina os comportamentos dos métodos dos Mocks chamados de acordo com o objetivo de teste

# Criando Mocks 1 – análise da classe RemessaService

1 - Identifique a unidade a ser testada (classe -> método)

```
14 @Service
15 public class RemessaService {
16
17     private UsuarioDAO usuarioDao;
18     private RemessaDAO remessaDao;
19     private NotificadorService notificadorService;
20
21     @Autowired
22     public RemessaService(UsuarioDAO usuarioDao, RemessaDAO remessaDao, NotificadorService notificadorService) {
23         this.usuarioDao = usuarioDao;
24         this.remessaDao = remessaDao;
25         this.notificadorService = notificadorService;
26     }
27
28     public String retornaNomeProprietario(String emailDoProprietario) {
29         return this.usuarioDao.buscarPorEmail(emailDoProprietario).getNome();
30     }
31 }
```

2 - Identifique as **dependências** que são chamadas/utilizadas no método (menor unidade possível) a ser testada

Unidade de teste

3 - Identifique quais métodos dessas dependências são chamados/invocados e qual o comportamento

# Criando Mocks 1

Classe de teste

```
class RemessaServiceTest {
```

```
    private RemessaService remessaService; 
```

Classe sendo testada

```
    @Test
```

```
    void testaRetornarNomeDoProprietario() {
```

4 - Crie os mocks necessários

```
        // Inicializando os Mocks que sao utilizados na classe a ser testada
        UsuarioDAO usuarioDaoMock = Mockito.mock(UsuarioDAO.class);
```

```
        RemessaDAO remessaDaoMock = Mockito.mock(RemessaDAO.class);
```

```
        NotificadorService notificadorServiceMock = Mockito.mock(NotificadorService.class);
```

```
        // Incluindo os Mocks na classe a ser testada
```

```
        this.remessaService = new RemessaService(usuarioDaoMock, remessaDaoMock, notificadorServiceMock);
```


# Existem outras formas (mais?) elegantes de inicializar os mocks

## Criando Mocks 2 – análise da classe HonraService

```
14 @Service
15 public class HonraService {
16
17     @Autowired
18     private HonraDAO honraDao;
19
20     @Autowired
21     private RemessaDAO remessaDao;
22
23     @Autowired
24     private NotificadorService notificadorService;
25
26     public int getStatusDeLiquidacaoDaHonra(int idHonra) {
27
28         Honra honra = this.honraDao.getHonra(idHonra);
29         return honra.statusDeLiquidacao();
30     }
31 }
```

Dependências

Unidade de teste



# Existem outras formas (mais?) elegantes de inicializar os mocks

```
@ExtendWith(MockitoExtension.class)
public class RemessaServiceTest1 {
```

Classe de teste

```
    @Mock
    private UsuarioDAO usuarioDaoMock;

    @Mock
    private RemessaDAO remessaDaoMock;

    @Mock
    private NotificadorService notificadorServiceMock;

    @InjectMocks
    private RemessaService remessaService;
```

Classe sendo testada

12 `@ExtendWith(MockitoExtension.class)` is equivalent of `@RunWith(MockitoJUnitRunner.class)` of the JUnit4 – [Sergey Nemchinov](#) May 18, 2020 at 11:32

4 - Crie os mocks necessários



# @Inject, @Component [...]

Observe que, desde [o Spring 4.3](#), você nem precisa de um @Autowired em seu construtor, portanto, pode escrever seu código no estilo Java em vez de vincular-se às anotações do Spring. Seu trecho ficaria assim:

```
@Component
public class SomeService {
    private final SomeOtherService someOtherService;

    public SomeService(SomeOtherService someOtherService){
        this.someOtherService = someOtherService;
    }
}
```

- 1 You can use Lombok's annotation: `@RequiredArgsConstructor` instead of the explicit constructor.  
– [Marcus Voltolim](#) Jul 11 at 18:58 ✎
- 3 @stinger not really, really related but I think a lot of projects are using lombok and Spring together. In our company we have multiple project where we use both and it makes developing a breeze. Instead of writing the constructor (and potentially updating it) you simply slap a `@RequiredArgsConstructor` and mark any injected service/component `final`. Lombok will create the constructor and Spring will inject it.  
– [Stephan Stahlmann](#) Jul 29 at 10:23



# Definindo o comportamento dos mocks

Além de substituir as dependências, os nossos *dublês (Mocks)* podem simular comportamentos.

Se o comportamento não for definido, o Mock devolve valores padrão: 0, false, null... Por isso é importante que

5 – Defina os comportamentos dos métodos dos Mocks chamados de acordo com o objetivo de teste

# Definindo comportamento dos Mocks

Em RemessaService no método retornaNomeProprietario chamamos o método buscarPorEmail da dependência UsuarioDAO

Precisamos dizer ao Mockito: Veja bem Mockito, quando o método buscarPorEmail do Mock de UsuarioDAO for chamado recebendo como parâmetro qualquer String, retorne esse Usuario user.

```
28 public String retornaNomeProprietario(String emailDoProprietario) {  
29     return this.usuarioDao.buscarPorEmail(emailDoProprietario).getNome();  
30 }  
31
```

```
@Test  
void testaRetornarNomeDoProprietario() {  
  
    Usuario user = new Usuario("teste123", "g@gmail.com", "12345");  
  
    // Definindo o comportamento  
    Mockito.when(usuarioDaoMock.buscarPorEmail(Mockito.anyString())).thenReturn(user);  
  
    Assert.assertEquals("teste123", this.remessaService.retornaNomeProprietario("g@gmail.com"));  
}
```



# Definindo comportamento dos Mocks

Em HonraService no método getStatusDeLiquidacaoDaHonra chamamos o método getHonra da dependência HonraDAO

Precisamos dizer ao Mockito: Veja bem Mockito, quando o método getHonra do Mock de HonraDAO for chamado recebendo como parâmetro qualquer int, retorne essa Honra honra

```
@Test
void testaGetStatusDeLiquidacaoDaHonra() {

    Remessa remessa = new Remessa();
    remessa.setValor(new BigDecimal("100.00"));

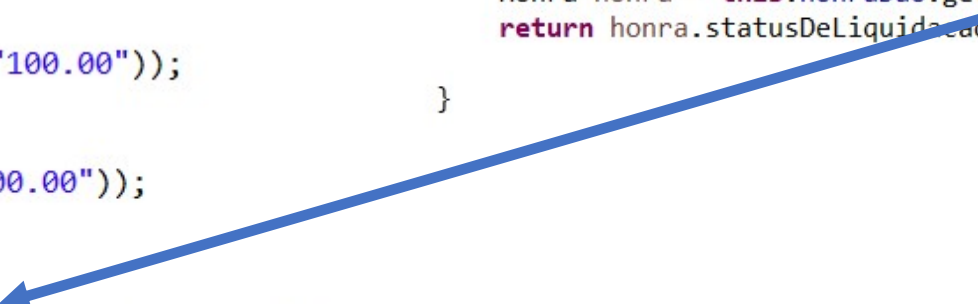
    Honra honra = new Honra();
    honra.setValor(new BigDecimal("100.00"));
    honra.setRemessa(remessa);

    // Definindo o comportamento
    Mockito.when(honraDaoMock.getHonra(Mockito.anyInt())).thenReturn(honra);

    Assert.assertEquals(0 , honraService.getStatusDeLiquidacaoDaHonra(2));
}
```

```
public int getStatusDeLiquidacaoDaHonra(int idHonra) {

    Honra honra = this.honraDao.getHonra(idHonra);
    return honra.statusDeLiquidacao();
}
```



# Vamos testar o método iniciaRemessa da classe RemessaService

```
public class RemessaServiceTest2 {  
    @Mock  
    private UsuarioDAO usuarioDaoMock;  
  
    @Mock  
    private RemessaDAO remessaDaoMock;  
  
    @Mock  
    private NotificadorService notificadorServiceMock;  
  
    @InjectMocks  
    private RemessaService remessaService;  
  
    @BeforeEach  
    public void beforeEach() {  
        // inicializando os Mocks  
        MockitoAnnotations.openMocks(this);  
    }  
  
    @Test  
    void testaIniciaRemessa() {
```

# Vamos testar o método iniciaRemessa da classe RemessaService

```
38
39 @Test
40 void testaIniciaRemessa() {
41
42     // Preciso de um usuario
43     Usuario user = new Usuario("teste123", "g@gmail.com", "12345");
44     Mockito.when(usuarioDaoMock.buscarPorEmail(Mockito.anyString())).thenReturn(user);
45
46     // Preciso de um Long
47     Mockito.when(remessaDaoMock.adicionarRemessa(Mockito.any(Remessa.class))).thenReturn(new Long(1));
48
49     Assert.assertEquals(new Long(1), this.remessaService.iniciaRemessa("g@gmail.com", new BigDecimal("200.00"), "remessa123"));
50
51 }
52
```

# Verificando se os métodos do Mock foram de fato chamados

```
public Long iniciaRemessa(String emailDoProprietario, BigDecimal valor, String nome)

    Usuario proprietario = this.usuarioDao.buscarPorEmail(emailDoProprietario);

    Remessa remessa = new Remessa();
    remessa.setNome(nome);
    remessa.setValor(valor);
    remessa.setDataAbertura(LocalDate.now());
    remessa.setUsuario(proprietario);

    Long id = this.remessaDao.adicionarRemessa(remessa);

    this.notificadorService.notificaDonoDaRemessa(remessa);

    return id;
}
```

Temos um método de uma dependência que é chamado dentro do método testado. Podemos verificar se esse método (do Mock) foi de fato chamado.

# Verificando se os métodos do Mock foram de fato chamados

```
Mockito.verify(notificadorServiceMock)  
    .notificaDonoDaRemessa(Mockito.any(Remessa.class));
```

Temos um método de uma dependência que é chamado dentro do método testado. Podemos verificar se esse método (do Mock) foi de fato chamado.

# Verificando se os métodos do Mock NotificadorService não é chamado quando ocorre exceção

```
@Test
void testaIniciaRemessaComExcecao() {

    // Preciso de um usuario
    Usuario user = new Usuario("teste123", "g@gmail.com", "12345");
    Mockito.when(usuarioDaoMock.buscarPorEmail(Mockito.anyString())).thenReturn(user);

    // Preciso de um Long
    Mockito.when(remessaDaoMock.adicionarRemessa(Mockito.any(Remessa.class))).thenThrow(RuntimeException.class);

    try {
        this.remessaService.iniciaRemessa("g@gmail.com", new BigDecimal("200.00"), "remessa123");
        Mockito.verifyNoInteractions(notificadorServiceMock);
    } catch (Exception e) {

    }

}
```



# Professor, mas esse caso ai é lógico que não ia ser chamado... e nas evoluções futuras?

```
public Long iniciaRemessaFeio(String emailDoProprietario, BigDecimal valor, String nome) {  
  
    Usuario proprietario = null;  
    Long id = null;  
    Remessa remessa = null;  
  
    try {  
        proprietario = this.usuarioDao.buscarPorEmail(emailDoProprietario);  
  
        remessa = new Remessa();  
        remessa.setNome(nome);  
        remessa.setValor(valor);  
        remessa.setDataAbertura(LocalDate.now());  
        remessa.setUsuario(proprietario);  
  
        id = this.remessaDao.adicionarRemessa(remessa);  
    } catch (Exception e) {  
        // TODO: handle exception  
    }  
  
    this.notificadorService.notificaDonoDaRemessa(remessa);  
  
    return id;  
}
```

Testes unitários também  
servem como testes de  
regressão!

# Capturando objetos

```
public Long iniciaRemessa(String emailDoProprietario, BigDecimal valor, String nome)
```

```
    Usuario proprietario = this.usuarioDao.buscarPorEmail(emailDoProprietario);
```

```
    Remessa remessa = new Remessa();  
    remessa.setNome(nome);  
    remessa.setValor(valor);  
    remessa.setDataAbertura(LocalDate.now());  
    remessa.setUsuario(proprietario);
```

```
    Long id = this.remessaDao.adicionarRemessa(remessa);
```

```
    this.notificadorService.notificaDonoDaRemessa(remessa);
```

```
    return id;
```

```
}
```

Temos um objeto que é criado e manipulado dentro da classe sendo testada, podemos capturar esse objeto e verificar se está correto

# Capturando objetos

```
@Captor
private ArgumentCaptor<Remessa> captorRemessa;

- - -

@Test
void testaIniciaRemessaCapturando() {

    // Preciso de um usuario
    Usuario user = new Usuario("teste123", "g@gmail.com", "12345");
    Mockito.when(usuarioDaoMock.buscarPorEmail(Mockito.anyString())).thenReturn(user);

    // Preciso de um Long
    Mockito.when(remessaDaoMock.adicionarRemessa(Mockito.any(Remessa.class))).thenReturn(new Long(1));

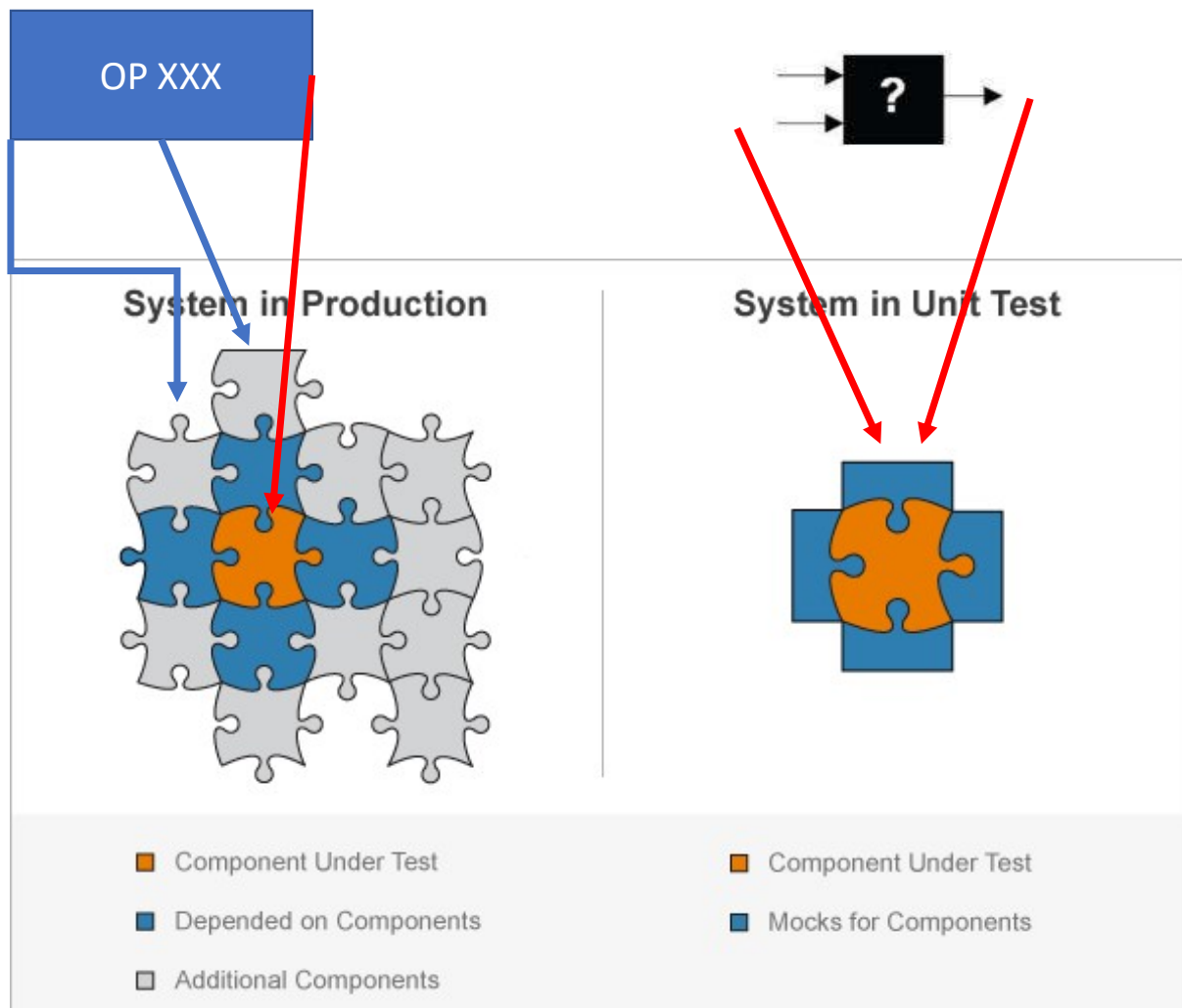
    Assert.assertEquals(new Long(1), this.remessaService.iniciaRemessa("g@gmail.com", new BigDecimal("200.00"), "remessa123"));

    Mockito.verify(remessaDaoMock).adicionarRemessa(captorRemessa.capture());

    Remessa remessaCapturada = captorRemessa.getValue();

    Assert.assertEquals("remessa123", remessaCapturada.getNome());

}
```



Agora vamos para as operações externas.



---

# Obrigado!

---



- <https://www.linkedin.com/in/gleyserguimaraes/>
- [gbomfim@minsait.com](mailto:gbomfim@minsait.com)

minsait

An Indra company

minsait

Mark Making the way forward

An Indra company