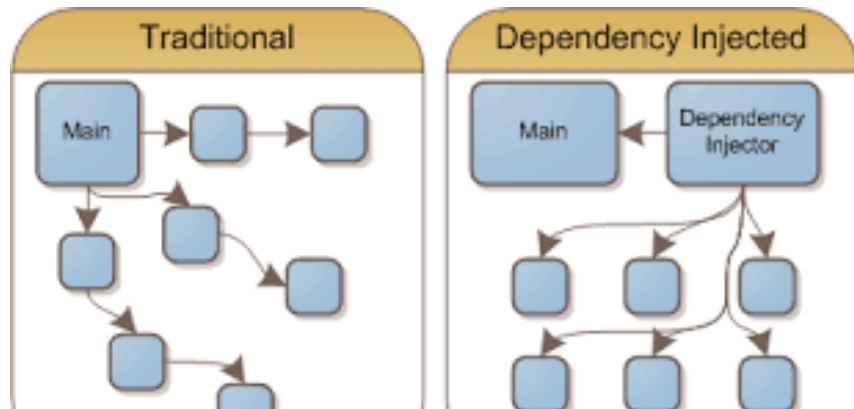


Verificação e Validação de Software

Prof. Bernardo Copstein





Testando classes com dependências

- Leituras recomendadas:
- Maurício Aniche. Effective Software Testing: A developer's guide (capítulo 6). Manning Publications Co.

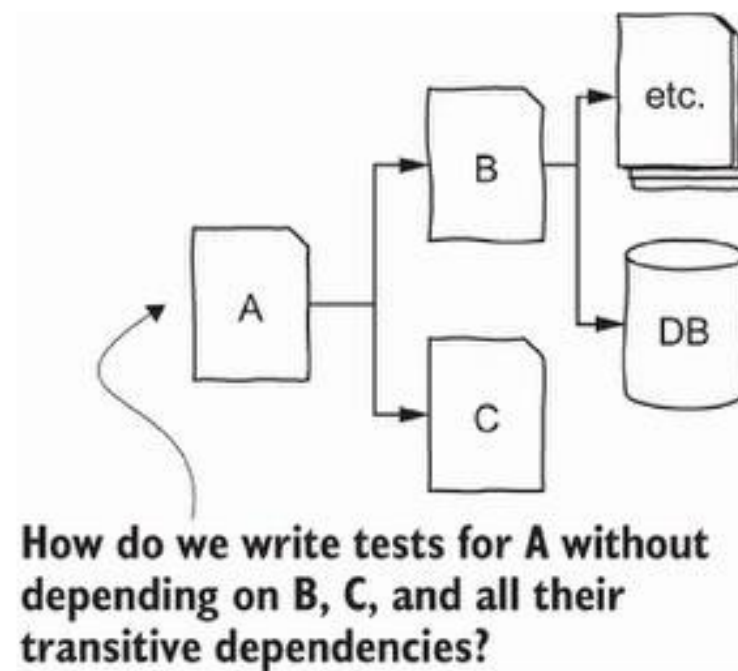


Usando doubles

- Até agora testamos classes isoladas umas das outras
 - Informamos valores de entrada para um método e verificamos suas saídas
 - Ou chamamos um método que altera o estado de uma classe e verificamos se o estado alvo foi atingido
- Algumas classes, porém, dependem de outras para fazer seu trabalho
 - Naturalmente é importante testar essas classes trabalhando em conjunto com suas dependências, porém ...
 - É importante saber se suas funcionalidades intrínsecas estão funcionando
 - Pode ser difícil, lento ou dar muito trabalho testar uma classe em conjunto com suas dependências

Exemplo: a classe *IssuedInvoices*

- Considere um sistema de faturas que contém:
 - Uma classe chamada *IssuedInvoices* que acessa o banco de dados e contém código SQL para pesquisar faturas
 - Uma classe chamada *InvoiceGenerationService* que gera novas faturas que depende de *IssuedInvoices*.
- Isso significa que para testar a classe *InvoiceGenerationService*, indiretamente estaremos usando *IssuedInvoices*.
- Como testar uma classe que acessa um banco de dados e outras coisas “complicadas”?



Usando doubles

- Para testar classes que dependem de outras sem testar a integração entre elas usamos doubles
- Um double “imita” o comportamento de uma classe real de uma maneira controlada.
- Assim podemos testar o comportamento da classe alvo mantendo o controle sobre o comportamento das classes das quais ela depende.
- Vantagens do uso de doubles:
 - Maior controle
 - Podemos definir o comportamento esperado para o double
 - Mais velocidade
 - Operações com bancos de dados ou serviços podem ser “lentas”
 - Melhoria no projeto das classes
 - Quando usados como uma técnica de projeto, doubles tendem a fazer os desenvolvedores refletirem sobre como as classes devem interagir entre si, como devem ser seus contratos e quais são seus limites conceituais.

Tipos de doubles

- **Dummys:** são objetos passados para a classe alvo mas nunca usados. Podem ser, por exemplo, objetos que necessitam ser informados por parâmetro mas que não serão usados no teste específico.
 - **Fakes:** são classes reais mas implementadas de uma forma muito mais simples que a original. Por exemplo, ao invés de acessar um banco de dados um “repository” que consulta um arranjo inicializado com um conjunto fixo de objetos.
 - **Stubs:** provém respostas prontas para as chamadas executadas durante o teste. São o tipo mais popular de duple.
 - **Mocks:** aparentemente são equivalentes a “stubs”. A diferença é que podem registrar as chamadas de métodos que foram feita para conferência posterior. Podem verificar se um determinado método foi chamado ou quantas vezes foi chamado.
 - **Spies:** neste caso o objetivo não é substituir o objeto original, mas apenas observar sua interação com os demais. Neste caso o “spie” funciona como um envelope ao redor do objeto acrescentando a capacidade de registrar as interações.
-



Introdução aos frameworks para trabalhar com doubles

- O “mockito” é um dos frameworks mais populares para se trabalhar com doubles em Java
 - Permite criar stubs e mocks
 - Consulte a documentação em: <https://site.mockito.org/>
-

Fundamentos do Mockito

O mockito é tão simples que é suficiente conhecer 3 métodos:

- **mock(<class>):** cria um mock/stub de uma determinada classe. Pode-se obter a classe usando-se “<NomeDaClasse>.class”.
- **when(<mock>.<method>).thenReturn(<value>):** neste caso o “value” será retornado. Ex: *when(issuedInvoices.all()).thenReturn(someListHere)*.
- **verify(<mock>).<method>:** verifica se o mock foi exercitado da forma esperada em relação ao método especificado. Ex: *verify(issuedInvoices).all()*.

Dependências para o Maven

```
<dependency>  
  <groupId>org.mockito</groupId>  
  <artifactId>mockito-core</artifactId>  
  <version>3.5.13</version>  
</dependency>
```



Um exemplo prático

- Suponha o seguinte requisito: “Retornar todas as faturas com valores menores que 100. O conjunto de faturas esta armazenado no banco de dados. A classe *IssuedInvoices* contém um método que recupera todas as faturas”.
- O código ao lado implementa este requisito.



```
import java.util.List;
import static java.util.stream.Collectors.toList;

public class InvoiceFilter {

    public List<Invoice> lowValueInvoices() {
        DatabaseConnection dbConnection = new DatabaseConnection();
        IssuedInvoices issuedInvoices = new IssuedInvoices(dbConnection);

        try {
            List<Invoice> all = issuedInvoices.all();
            return all.stream()
                .filter(invoice -> invoice.getValue() < 100)
                .collect(toList());
        } finally {
            dbConnection.close();
        }
    }
}
```

Note que a classe cria a instancia de *issuedInvoices*

Testando *lowValueInvoices*

- Como a classe *InvoiceFilter* cria a instancia de *IssuedInvoices* internamente, é necessário preparar o banco de dados antes. Assim, antes de cada teste é necessário:
 - Criar a conexão com o DB
 - Limpar o banco de dados
 - Criar o cenário
- Depois de cada teste é necessário encerrar a conexão com o banco

```
public class InvoiceFilterTest {
    private IssuedInvoices invoices;
    private DatabaseConnection dbConnection;

    @BeforeEach
    public void open() {
        dbConnection = new DatabaseConnection();
        invoices = new IssuedInvoices(dbConnection);
        dbConnection.resetDatabase();
        Invoice mauricio = new Invoice("Mauricio", 20);
        Invoice steve = new Invoice("Steve", 99);
        Invoice frank = new Invoice("Frank", 100);
        invoices.save(mauricio); invoices.save(steve); invoices.save(frank);
    }

    @AfterEach
    public void close() { if (dbConnection != null) dbConnection.close(); }

    @Test
    void filterInvoices() {
        InvoiceFilter filter = new InvoiceFilter();
        assertThat(filter.lowValueInvoices())
            .containsExactlyInAnyOrder(mauricio, steve);
    }
}
```

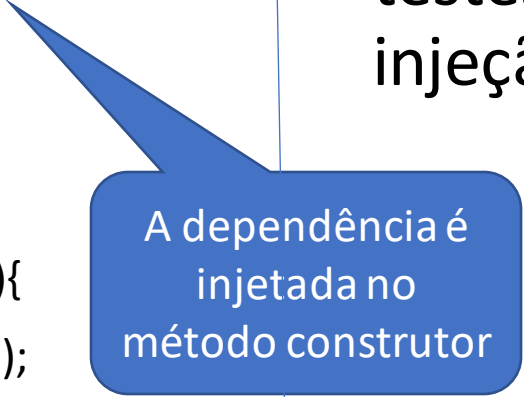


AssertJ

Refatorando *InvoiceFilter*

```
public class InvoiceFilter {  
    private final IssuedInvoices issuedInvoices;  
    public InvoiceFilter(IssuedInvoices issuedInvoices) {  
        this.issuedInvoices = issuedInvoices;  
    }  
  
    public List<Invoice> lowValueInvoices() {  
        List<Invoice> all = issuedInvoices.all();  
        return all.stream()  
            .filter(invoice -> invoice.getValue() < 100)  
            .collect(toList());  
    }  
}
```

- Esta nova versão de *InvoiceFilter* tem um design orientado a testes: usa-se o padrão de injeção de dependências.



A dependência é
injetada no
método construtor

Criando um stub

- Como a nova versão da classe InvoiceFilter agora recebe IssuedInvoices por injeção de dependência, podemos injetar um **stub** no lugar do IssuedInvoices.
- Note que o teste é feito a partir do comportamento do stub. Desta forma isolamos o comportamento de *IssuedInvoices*.

```
public class InvoiceFilterTest {  
    @Test  
    void filterInvoices() {  
        IssuedInvoices issuedInvoices = mock(IssuedInvoices.class);  
        Invoice mauricio = new Invoice("Mauricio",  
        Invoice steve = new Invoice("Steve", 99);  
        Invoice frank = new Invoice("Frank", 100);  
        List<Invoice> listOfInvoices = Arrays.asList(mauricio, steve, frank);  
        when(issuedInvoices.all()).thenReturn(listOfInvoices);  
        InvoiceFilter filter = new InvoiceFilter(issuedInvoices);  
        assertThat(filter.lowValueInvoices()  
            .containsExactlyInAnyOrder(mauricio, steve);  
    }  
}
```

Criação do stub

Criação da lista de dados

Definição do comportamento do stub

Injeção do stub no lugar do DB

Teste baseado no comportamento do stub

Usando Mocks para verificar expectativas

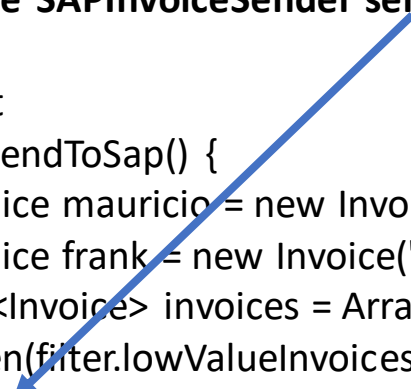
- Vamos supor que o sistema tem um novo requisito: todas as faturas de valor baixo devem ser enviadas para o SAP da empresa. A classe *SAPInvoiceSender* recebe pelo construtor a classe básica de comunicação com o SAP (a interface *SAP* abstrai o mecanismo de comunicação).

```
public class SAPInvoiceSender {  
    private final InvoiceFilter filter;  
    private final SAP sap;  
  
    public SAPInvoiceSender(InvoiceFilter filter, SAP sap) {  
        this.filter = filter;  
        this.sap = sap;  
    }  
    public void sendLowValuedInvoices(){  
        List<Invoice> lowValuedInvoices = filter.lowValueInvoices();  
        for(Invoice invoice : lowValuedInvoices){  
            sap.send(invoice);  
        }  
    }  
}
```

Verificando expectativas

- O objetivo deste teste não é verificar o comportamento de InvoiceFilter e sim verificar se toda a fatura de custo baixo é enviada para o SAP.
- Note que fazemos isso mesmo sem ter a implementação de SAP.

```
public class SAPInvoiceSenderTest {  
    private InvoiceFilter filter = mock(InvoiceFilter.class);  
    private SAP sap = mock(SAP.class);  
    private SAPInvoiceSender sender=new SAPInvoiceSender(filter, sap);  
  
    @Test  
    void sendToSap() {  
        Invoice mauricio = new Invoice("Mauricio", 20);  
        Invoice frank = new Invoice("Frank", 99);  
        List<Invoice> invoices = Arrays.asList(mauricio, frank);  
        when(filter.lowValueInvoices()).thenReturn(invoices);  
        sender.sendLowValuedInvoices();  
        verify(sap).send(mauricio);  
        verify(sap).send(frank);  
    }  
}
```



Outras formas de verificação

- Comandos mais precisos:

```
verify(sap, times(2)).send(any(Invoice.class));  
verify(sap, times(1)).send(mauricio);  
verify(sap, times(1)).send(frunk);
```

- Verificando o oposto:

```
verify(sap, never()).send(any(Invoice.class));
```

- Como forçar a geração de exceções em determinadas ocasiões (visando testar o tratamento de exceções)

```
doThrow(new SAPException()).when(sap).send(frunkInvoice);
```


Quando usar e quando não usar mocks

Deve-se usar mocks quando:

- As classes das quais a classe alvo depende forem muito lentas
- As classes das quais a classe alvo depende conectam com infra estrutura externa (BDs, serviços)
- Quando queremos testar situações difíceis de simular (exceções difíceis de gerar)
- Quando a classe alvo depende de classes que dependem por sua vez de muitas outras
- Quando queremos isolar data e hora

Não deve-se usar mocks quando:

- A classe alvo depende de entidades que não são complexas de instanciar
- Bibliotecas externas ou métodos utilitários
- Classes muito simples

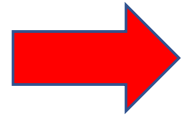
Lidando com bibliotecas externas

- Não é aconselhável “mockar” bibliotecas externas ou classes sobre as quais não temos controle sobre as alterações
 - Elas podem mudar e todos os testes que utilizam “mocks” dessas classes podem se tornar inconsistentes (mas não ter erro de compilação)
 - Exemplo: o método somava +1 e agora soma +2, porém, o comportamento “mockado” continua somando +1
- A solução nestes casos é criar um “envelope” ao redor da classe externa. Dessa maneira, em caso de mudança na classe externa, podemos ajustar o comportamento na classe envelope.

Exemplo de classe externa: LocalDate

```
public class ChristmasDiscount {  
    public double applyDiscount(double amount) {  
        LocalDate today = LocalDate.now();  
        double discountPercentage = 0;  
        boolean isChristmas = (today.getMonth() ==  
            Month.DECEMBER && today.getDayOfMonth()==25);  
        if (isChristmas)  
            discountPercentage = 0.15;  
        return amount - (amount * discountPercentage);  
    }  
}
```

- **Dois problemas:**
 - Dependência escondida
 - Classe externa
- **Soluções:**
 - Classe envelope (Clock)
 - Injeção de dependência

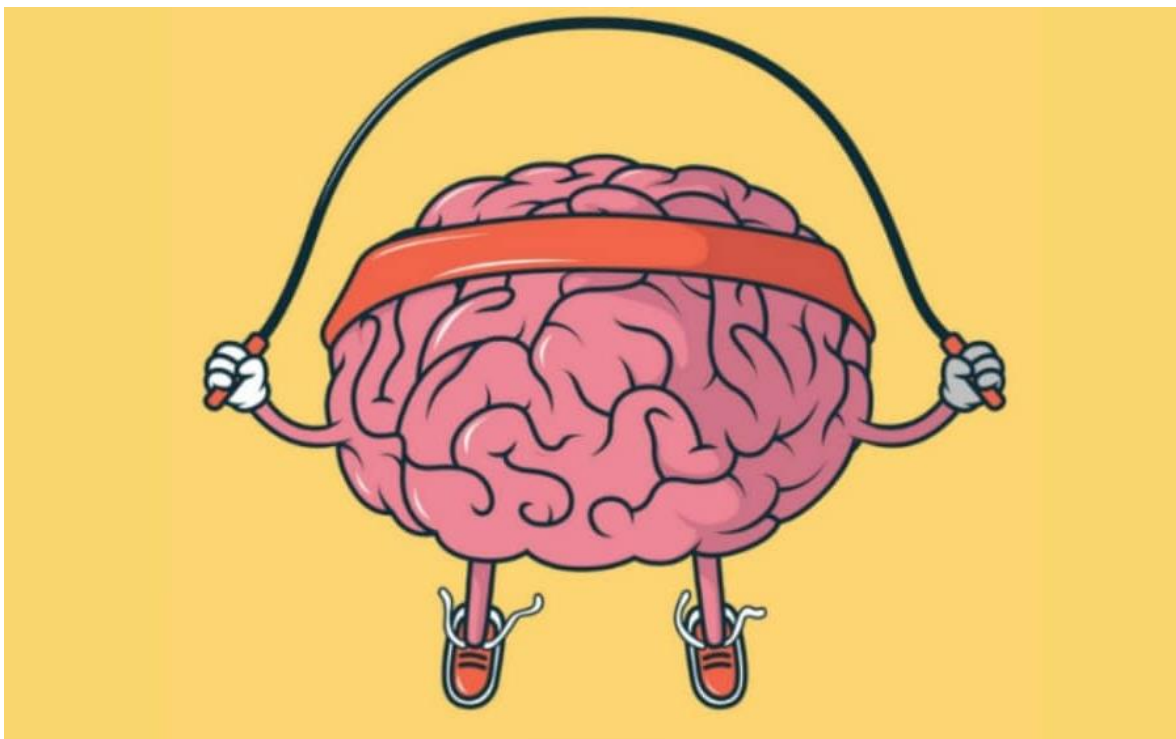


```
public class Clock {  
    public static LocalDate now() {  
        return LocalDate.now();  
    }  
    // outras operações de data e hora necessárias  
}
```

```
public class ChristmasDiscount {  
    private Clock today;  
    public ChristmasDiscount(Clock clock){  
        this.today = clock;  
    }  
    public double applyDiscount(double amount) {  
        LocalDate today = Clock.now();  
        double discountPercentage = 0;  
        boolean isChristmas = (today.getMonth() ==  
            Month.DECEMBER &&  
today.getDayOfMonth()==25);  
        if (isChristmas)  
            discountPercentage = 0.15;  
        return amount - (amount * discountPercentage);  
    }  
}
```

Testando ChristmasDiscount

```
public class ChristmasDiscountTest {  
    private final Clock clock = mock(Clock.class);  
    private final ChristmasDiscount cd = new ChristmasDiscount(clock);  
  
    @Test  
    public void christmas() {  
        LocalDate christmas = LocalDate.of(2015, Month.DECEMBER, 25);  
        when(clock.now()).thenReturn(christmas);  
  
        double finalValue = cd.applyDiscount(100.0);  
        assertThat(finalValue).isCloseTo(85.0, offset(0.001));  
    }  
  
    @Test  
    public void notChristmas() {  
        LocalDate notChristmas = LocalDate.of(2015, Month.DECEMBER, 26);  
        when(clock.now()).thenReturn(notChristmas);  
        double finalValue = cd.applyDiscount(100.0);  
        assertThat(finalValue).isCloseTo(100.0, offset(0.001));  
    }  
}
```



Veja a lista de
exercícios

.....
.....
.....
.....

