Erros de Software e TDD

Prof. Hugo de Paula

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Departamento de Ciência da Computação

Sumário

Índice de Massa Corporal

Considere o exemplo a seguir que calcula o Índice de Massa Corporal (IMC) de uma pessoa, dados seu peso e altura.

```
public class CalculadoraIMC {
  private double peso, altura, imc;

  public CalculadoraIMC(double peso, double altura) {
    this.peso = peso;
    this.altura = altura;
  }
  public void calcular() {
    this.imc = peso / (altura * altura);
  }
  public boolean comSobrepeso() {
    return (imc > 25);
  }
  public double getImc() {
    return this.imc;
  }
}
```

Índice de Massa Corporal

- O estado do objeto corresponde ao peso, à altura e ao imc.
- O comportamento do objeto é ditado pelos métodos calcular() e comSobrepeso().
- O método getImc() é apenas um método de acesso.

```
public static void main(String[] args) {
  CalculadoraIMC calcImc = new CalculadoraIMC(70, 1.60);
  calcImc.calcular();

  System.out.println("O IMC e: " + calcImc.getImc());

  if (calcImc.comSobrepeso()) {
    System.out.println("Tem sobrepeso.");
  }
}
```

Debugging

Debugging

Processo utilizado para remover erros de programação (*bugs*).

Passos para o *debugging*:

1. Detectar o erro.
2. Localizar o erro.
3. Solucionar o erro.

Tipos de erros de programação:

- Erros de sintaxe / de compilação.
- Erros em tempo de execução.
- Erros semânticos ou de lógica.

Erros de sintaxe

- Violam normas gramaticais da linguagem.
- São capturados previamente pelos compiladores ou interpretadores.
- São identificadas pela IDE.

```
public void calcular() {
  this.imc = peso / (altura * altura),
}
```

Erros em tempo de execução

- Ocorrem durante a execução do programa.
- Não podem ser capturados pelo compilador ou pela IDE.
- Em algumas IDEs, o erro pode ser alertado. Exemplo: Variável não inicializada.
- Devem ser antecipados e tratados.

```
public void calcular() {
  this.imc = peso / (altura * altura);
}
```

Se a altura for igual a 0 (zero), ocorrerá o erro "*/ by zero*" (divisão por zero).

Erros semânticos ou de lógica

- Os mais difíceis de detectar.
- Programa irá “funcionar”, mas a saída estará incorreta.

```
public void calcular() {
  this.imc = (peso * peso) / altura;
}
```

Princípios do teste de software

- Objetivo do teste é verificação e validação.

Verificação

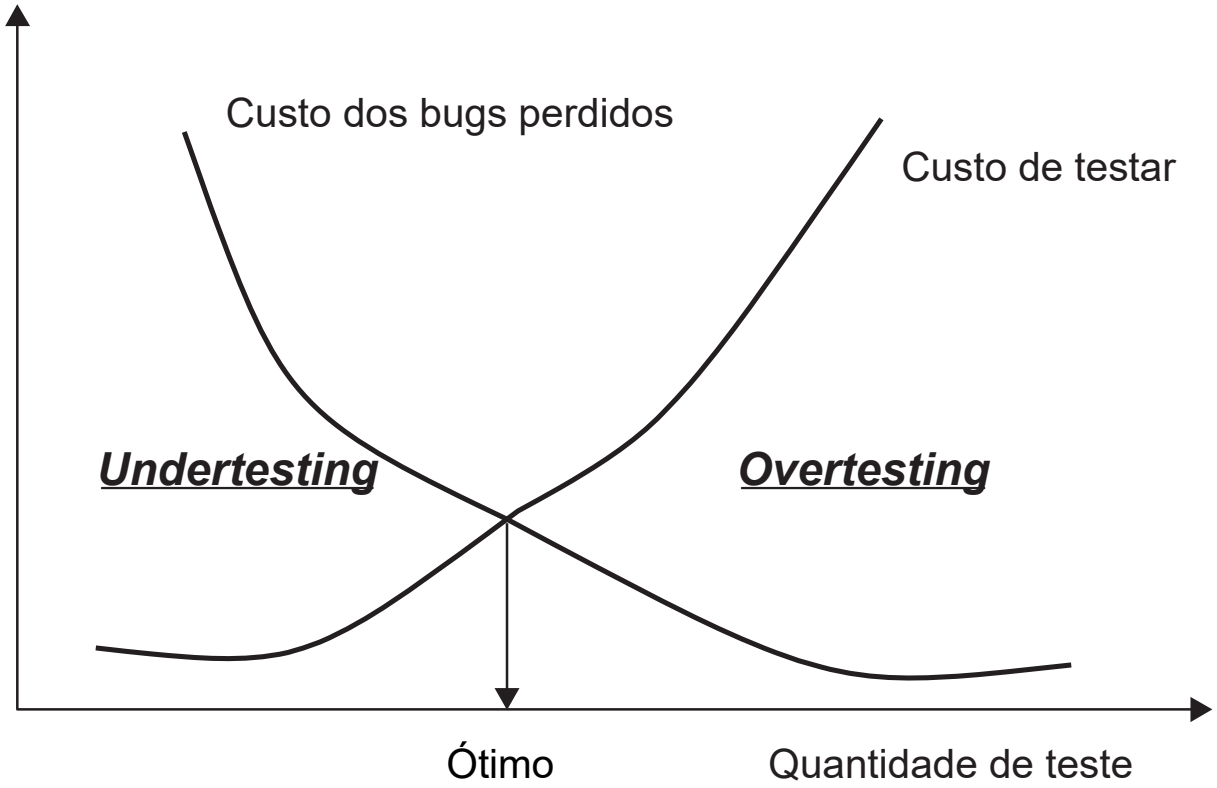
objetiva responder se o sistema foi construído corretamente.

Validação

tenta determinar se foi construído o sistema certo.

Custo do teste de software

- O custo do erro aumenta, quanto mais se demora a detectá-lo.



Adaptado de: *Beginning Java Programming (2015) - Bart Baesens and Aimée Backiel*; Capítulo 1, pg. 7

Test-driven development

Test-driven development (TDD)

Ou desenvolvimento dirigido por teste é uma metodologia de desenvolvimento de software que enfatiza o teste.

"Escreva o código de teste antes e então escreva o código operacional e depure o programa até que ele passe no teste."

- TDD é uma metodologia de desenvolvimento, não de teste.
- Software é desenvolvido em pequenas iterações.
- Testes de unidade são desenvolvidos antes do código.

TDD: como funciona

1. Adicione um teste.
   - Estórias do usuário ajudam a entender os requisitos.
2. Execute todos os testes e veja o novo falhar.
   - Garante que o mecanismo de teste está funcionando e que o teste não irá passar por engano.
3. Escreva o código.
   - Apenas o código que foi projetado para passar no teste.
   - Nenhuma funcionalidade adicional deve ser incluída, pois não seria testada.

TDD: como funciona

4. Execute os testes e veja eles serem bem sucedidos.
   - Se o teste passou, tem-se confiança de que o código atende aos requisitos testados.
5. Refatore o código.
   - Limpe o código e reexecute os testes para garantir que nada foi quebrado.

Repita o processo.

Teoria do teste de unidade

Teste unitário (ou teste de unidade)

Para cada trecho de código operacional – uma classe ou um método, o código operacional deve estar pareado com um código de "teste unitário".

"If you can't write a test for what you are about to code, then you shouldn't even be thinking about coding."[1]

[1] George, Boby; Williams, Laurie. *A Structured experiment of test-driven development*. 2003.

Teste de unidade

- O código do teste unitário chama o código operacional a partir da sua interface pública, de diversas formas e verifica os resultados.
- O teste não precisa ser exaustivo – código de teste já agrega um grande valor, mesmo nos casos centrais.
- Testes unitários são uma forma padronizada de manter os testes em paralelo com o desenvolvimento do programa.
- Testes de unidade são fáceis de executar.
  - uma vez configurados, eles produzem *feedback* rápido para você testar suas ideias.

Tipos de teste de unidade

- **Básicos**: entradas simples e óbvias que devem funcionar. São adicionados primeiro.
- **Avançados**: casos de teste mais complexos. Normalmente feitos posteriormente quando um maior conhecimento do problema leva a identificar potenciais casos estranhos.
- **Borda/fronteira**: são casos simples mas que representam condições extremas – string vazia, lista vazia, etc.

JUnit com Eclipse

- JUnit é um sistema de teste de unidade em Java muito popular e bem integrado ao Eclipse.
- Para criar um caso de teste:
  - Clique com o botão direito na classe que deseja testar.
  - Selecione `New... JUnit Test Case`.
  - Se a classe se chama Shulambs, será criada uma classe chamada ShulambsTest.

JUnit.jar – adicionando ao projeto

- JUnit depende de classes disponíveis no arquivo `junit.jar`, já instalado como plugin do Eclipse.
- Ao criar uma classe de teste, o Eclipse irá perguntar se deseja adicionar o arquivo ao projeto automaticamente.
- Caso tenha que adicionar o arquivo manualmente, `Project`, selecione `properties... Java Build Path: Libraries`. Então utilize o botão `Add Jar` para adicionar o arquivo, que está na pasta `plugins`.

Exemplo JUnit: classe `Conjunto`

A clase Conjunto é uma coleção

- possui até *N* objetos.
- possui as operações básicas:

adicionar	adiciona um item
contem	verifica se item no conjunto
tamanho	número de itens

Exemplo JUnit: lista de testes

- Decidindo como testar:
  - *tamanho* $= 0$ se Conjunto está vazio.
  - *tamanho* $= N$ após adicionar N elementos.
  - adicionar um elemento que já existe no Conjunto não aumenta seu tamanho.
  - lança uma exceção se Conjunto já está lotado.
- cada teste verifica uma característica específica de Conjunto

Exemplo JUnit: primeiro teste

```java
public class Conjunto {

}
```

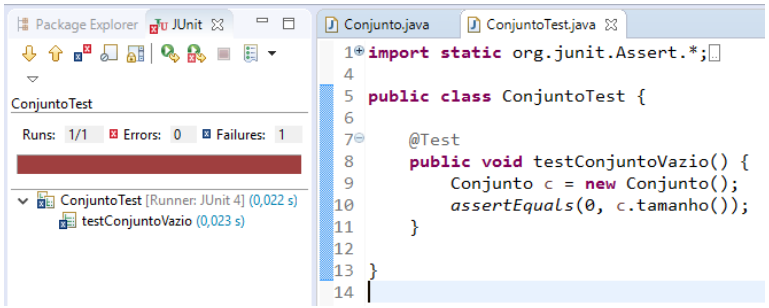
```java
public class ConjuntoTest {
    @Test
    public void testConjuntoVazio() {
        Conjunto c = new Conjunto();
        assertEquals(0, c.tamanho());
    }
}
```

- Não compila porque tamanho() não foi definido.

Exemplo JUnit: Testando o método `tamanho()`

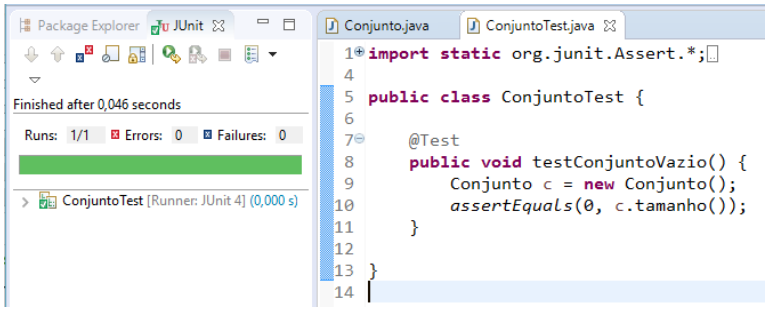
BARRA VERMELHA: código falhou no teste.

```
public class Conjunto {
    public int tamanho() {
        return 42;
    }
}
```



BARRA VERDE: código passou no teste.

```
public class Conjunto {
    public int tamanho() {
        return 0;
    }
}
```



Exemplo JUnit: testando adicionar itens

```
public class Conjunto {
    public int tamanho() {
        return 0;
    }
    public void adicionar(Object o) {

    }
}
```

```
public class ConjuntoTest {
    @Test
    public void testConjuntoVazio() {
        Conjunto c = new Conjunto();
        assertEquals(0, c.tamanho());
    }

    @Test
    public void testAdicionarUm() {
        Conjunto c = new Conjunto();
        c.adicionar(new Object());
        assertEquals(1,c.tamanho());
    }
}
```

Falhou como esperado, pois o tamanho não está sendo contado.

Exemplo JUnit: testando adicionar itens

```java
public class Conjunto {
    private int tamanho = 0;

    public int tamanho() {
        return tamanho;
    }
    public void adicionar(Object o) {
        tamanho++;
    }
}
```

```java
public class ConjuntoTest {
    @Test
    public void testConjuntoVazio() {
        Conjunto c = new Conjunto();
        assertEquals(0, c.tamanho());
    }

    @Test
    public void testAdicionarUm() {
        Conjunto c = new Conjunto();
        c.adicionar(new Object());
        assertEquals(1,c.tamanho());
    }
}
```

Passou no teste para adicionar um elemento.

Exemplo JUnit: testando adicionar item já existente

```
public class ConjuntoTest {
    @Test
    public void testConjuntoVazio() {
        Conjunto c = new Conjunto();
        assertEquals(0, c.tamanho());
    }

    @Test
    public void testAdicionarUm() {
        Conjunto c = new Conjunto();
        c.adicionar(new Object());
        assertEquals(1, c.tamanho());
    }

    @Test
    public void testAdicionarJaExistente() {
        Conjunto c = new Conjunto();
        Object o = new Object();
        c.adicionar(o);
        c.adicionar(o);
        assertEquals(1, c.tamanho());
    }

}
```

Falhou como esperado.

Exemplo JUnit: testando adicionar item já existente

```
public class Conjunto {
    private int tamanho = 0;
    public static final int MAX = 10;
    private Object[] items = new Object[MAX];

    public int tamanho() {
        return tamanho;
    }
```

```
    public void adicionar(Object o) {
        for (int i = 0; i < MAX; i++) {
            if (items[i] == o) {
                return ;
            }
        }
        items[tamanho++] = o;
    }
}
```

Passou como esperado. Agora que todos os testes passaram, podemos refatorar o loop.

Exemplo JUnit: refatorando

`Conjunto` antes	`Conjunto` depois

`public void adicionar(Object o) {`
`  for (int i = 0; i < MAX; i++) {`
`  if (items[i] == o) {`
`  return ;`
`  }`
`  }`
`  items[tamanho++] = o;`
`}`
	`public boolean contem(Object o) {`
	`  for (int i = 0; i < MAX; i++) {`
	`  if (items[i] == o) {`
	`  return true;`
	`  }`
	`  }`
	`  return false;`
	`}`
	
	`public void adicionar(Object o) {`
	`  if (!contem(o)) {`
	`  items[tamanho++] = o;`
	`  }`
	`}`

Passou no teste. A refatoração não quebrou o código.

Exemplo JUnit: finalizando

- Ainda não foi testado o limite da lista, mas isso será feito quando for feito tratamento de exceções.
- O ciclo: testar primeiro $\longrightarrow$ desenvolver código $\longrightarrow$ verificar que testes passaram $\longrightarrow$ refatorar produziu um código com garantia de qualidade.
- Testes podem ser facilmente executados toda vez que a classe for modificada.
- O código estará correto se os testes tiverem sido bem definidos.

Métodos de asserção da JUnit

assertTrue(teste)	falha se teste booleano é **false**.
assertFalse(teste)	falha se teste booleano é **true**.
assertEquals(esperado, teste real)	falha se valores não são iguais.
assertSame(esperado, teste real)	falha se valores não são os mesmos (através de ==).
assertNotSame(esperado, teste real)	falha se valores são os mesmos (através de ==).
assertNull(teste)	falha se valor não for **null**.
assertNotNull(teste)	falha se valor for **null**.
fail ()	faz com que o teste interrompa sua execução e falhe.

Melhorando a legibilidade dos testes

- Anotação @DisplayName define um nome para a classe ou o método de teste.
  - Por exemplo: @DisplayName("Testando o metodo contem.")
- String de informação é um parâmetro opcional que imprime informações no log de eventos durante a execução de um teste.
  - Será exibido caso o teste falhe.
- Exemplo:
  - assertTrue(shulambs(a. b), "chamada shulambs("+ a + "," + b + ")");

Exemplo JUnit

```
public class TestLista {
 @Test
 public void testAddGet() {
   Lista lst = new Lista();
   lst.add(1);
   assertEquals(1, lst.get(0), "recupera 1o elemento");
   lst.add(3);
   assertEquals(3, lst.get(1), "recupera 2o elemento");
 }
 @Test
 public void testIsEmpty() {
   Lista lst = new Lista();
   assertTrue(lst.isEmpty(), "lista vazia ao criar");
   lst.add(1);
   assertFalse(lst.isEmpty(), "lista nao vazia apos inserir");
   lst.remove(0);
   assertTrue(lst.isEmpty(), "lista vazia apos remover");
 }
}
```

Inicialização e finalização

- Métodos executados antes e depois de cada caso de teste.

```
@BeforeEach
void setUp() throws Exception {
}
@AfterEach
void tearDown() throws Exception {
}
```

- Métodos executados uma única vez, antes ou depois da execução de toda a classe de testes.

```
@BeforeAll
static void setUpBeforeClass() throws Exception {
}
@AfterAll
static void tearDownAfterClass() throws Exception {
}
```

Exemplo JUnit: pré-configuração do teste

```
public class ConjuntoTest {
  public static Conjunto c;

  @BeforeEach
  public void setUp() throws Exception {
    c = new Conjunto();
  }

  @Test
  public void testConjuntoVazio() {
    assertEquals(0, c.tamanho());
  }

  @Test
  public void testAdicionarUm() {
    int t = c.tamanho();
    c.adicionar(new Object());
    assertEquals(t+1, c.tamanho());
  }

  @Test
  public void testAdicionarJaExistente() {
    int t = c.tamanho();
    Object o = new Object();
    c.adicionar(o);
    c.adicionar(o);
    assertEquals(t+1, c.tamanho(),
                 "adic. Object 2 vezes");
  }
}
```

Exemplo de JUnit: Data

Baseado nos materiais de M. Ernst, S. Reges, D. Notkin, R. Mercer, http://www.cs.washington.edu/331/

- Considere uma classe Data com os seguintes métodos:

```
public Data(int ano, int mes, int dia)
public Data()                          // hoje
public int getDia(), getMes(), getAno()
public void adicionaDias(int dias) // avanca a data em dias
public int diasNoMes()
public String diaDaSemana()            // ex. "Segunda-feira"
public boolean equals(Object o)
public boolean eAnoBisexto()
public void proximoDia()               // avanca um dia
public String toString()               // ex. 5 de marco de 2035
```

- Proponha testes unitários para as situações a seguir:
  - Data não pode entrar em estado inválido.
  - Função adicionaDias funciona adequadamente.

Exemplo de JUnit: Data

- O que está errado com o exemplo a seguir?

```
public class DataTest {
  @Test
  public void test1() {
    Data d = new Data(2050, 2, 15);
    d.adicionaDias(4);
    assertEquals(d.getAno(), 2050);
    assertEquals(d.getMes(), 2);
    assertEquals(d.getDia(), 19);
  }
  @Test
  public void test2() {
    Data d = new Data(2050, 2, 15);
    d.adicionaDias(14);
    assertEquals(d.getAno(), 2050);
    assertEquals(d.getMes(), 3);
    assertEquals(d.getDia(), 1);
  }
}
```

Exemplo de JUnit: Data

```
public class DataTest {
  @Test
  public void test1() { // Usar nomes significativos.
    Data d = new Data(2050, 2, 15);
    d.adicionaDias(4);
    assertEquals(2050, d.getAno());   // Valor esperado
    assertEquals(2, d.getMes());      // a esquerda.
    assertEquals(19, d.getDia());
  } // Valores esperado e real ja sao exibidos
    // e nao precisam estar na mensagem.
  @Test
  public void test2() {
    Data d = new Data(2050, 2, 15);
    d.adicionaDias(14);
    assertEquals(2050, d.getAno(), "ano apos +14 dias");
    assertEquals(3, d.getMes(), "mes apos +14 dias");
    assertEquals(1, d.getDia(), "dia apos +14 dias");
  } // Mensagens devem explicar o que esta sendo testado
}
```

Eliminando redundâncias e tornando os testes curtos

- Use objetos com valores esperados para reduzir testes.

```
public class DataTest {
  @Test
  public void testAdicionaDias_numMesmoMes() {
    Data real = new Data(2050, 2, 15);
    real.adicionaDias(4);
    Data esperado = new Data(2050, 2, 19);
    assertEquals(esperado, real);
  } // Data deve possuir metodo toString e equals.

  @Test
  public void testAdicionaDias_proximoMes() {
    Data real = new Data(2050, 2, 15);
    real.adicionaDias(14);
    Data esperado = new Data(2050, 3, 1);
    assertEquals(esperado, real, "data apos +14 dias");
  }
}
```

Eliminando redundâncias e tornando os testes curtos

```
public class DataTest {
  @Test
  public void testAdicionaDias_numMesmoMes() {
    adicionaHelper(2050, 2, 15, 4, 2050, 2, 19);
  }

  @Test
  public void testAdicionaDias_proximoMes() {
    adicionaHelper(2050, 2, 15, 14, 2050, 3, 1);
  }

  private void adicionaHelper(int ano1, int mes1, int dia1,
                              int adic, int ano2, int mes2, int dia2) {
    Data real = new Data(ano1, mes1, dia1);
    real.adicionaDias(adic);
    Data esper = new Data(ano2, mes2, dia2);
    assertEquals(esper, real, "data apos +" + adic + " dias");
  }
}
```