Introdução à Programação

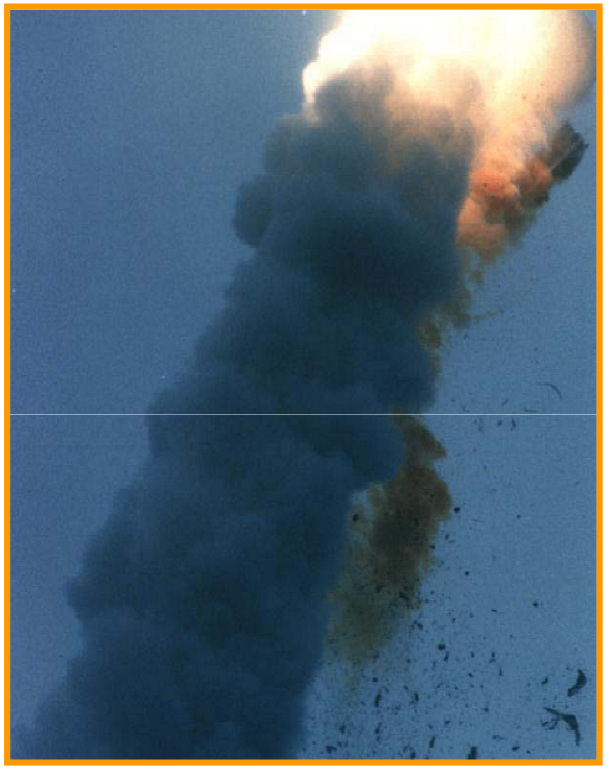
Introdução a Testes e Test First Design

Teste de Software

Lançamento do Ariane 5

Começou assim...

Mais de U$500 milhões de prejuízo



Terminou assim.

Falta de Testes

Tópicos da Aula

- Hoje, aprenderemos algumas formas de testar um programa
  - Importância de Testes
  - Objetivos
  - Cobertura
  - Teste de Caixa Preta x Teste de Caixa Branca
  - Teste de Unidade xTeste de Integração
- Veremos, ainda, uma técnica de desenvolvimento de programas baseada em testes
  - Test First Design
  - Vantagens
  - Construindo a classe Conta a partir de um teste de unidade
- Veremos também, diferença entre operador == e o método `equals`
  - Quando podemos e devemos utilizar método `equals`
  - Exemplos

Teste de Programa

- Podemos dizer que **testar** um programa consiste no processo de executar de forma controlada um programa de modo a verificar se o comportamento dele está de acordo com o esperado
  - Satisfaz a especificação
  - Satisfaz os desejos do cliente
- Entenda-se por execução de forma controlada, a definição, por parte do desenvolvedor do teste, das entradas e as operações executadas pelo programa

Importância de Testes

- Processo de teste deve ser conduzido durante várias etapas de desenvolvimento do software
  - Implementação de um componente
  - Integração de componentes
  - Validação do sistema pelo usuário final , etc
- Um processo efetivo de teste aumenta a qualidade de software
  - ⬆ Corretude
  - ⬆ Robustez
  - ⬇ Manutenção

Objetivos de Testes

- Testes podem ter diferentes propósitos:
  - Validação
    - Para demonstrar ao desenvolvedor e ao cliente que o software atende aos requisitos especificados
    - Um teste com sucesso mostra que o software opera da maneira esperada
  - Detecção de erros
    - Para descobrir falhas ou defeitos no software (ou tem comportamento anormal ou não está de acordo com a especificação)
    - Um teste com sucesso consegue fazer com que o software execute de forma anormal ou incorreta, expondo assim as falhas ou defeitos do software

Um teste não serve somente para verificar se o software executa corretamente, mas também para expor erros!

Casos de Teste

- Casos de teste (***test cases***) são especificações das entradas para o teste e as saídas esperadas
  - Pode conter também ações iniciadas pelos usuários
  - Contém também a descrição do que está sendo testado
- Geralmente um conjunto de *test cases* logicamente relacionados são organizados em ***test suites***

Cobertura de Teste

- Cobertura de teste é uma medida que descreve o quanto o código do software foi testado
  - Elaborar um teste com 100% de cobertura significa que todos os fluxos de controle e entrada possíveis foram testados (teste exaustivo)
    - Impraticável na maioria das vezes
    - Esforço computacional grande demais
  - Um bom plano de teste deve maximizar a cobertura, mas ao mesmo tempo minimizar o tempo de elaboração e realização do teste

Teste de Caixa Preta x Teste de Caixa Branca

- Testes em componentes ou no sistema podem levar em conta diferentes aspectos
  - Somente a funcionalidade
    - Comportamento externo
  - Além da funcionalidade, as estruturas internas
    - Comportamento interno

Teste de Caixa Preta x Teste de Caixa Branca

- Teste de Caixa Preta
  - Somente a funcionalidade importa
    - Comportamento é analisado através das entradas e saídas
- Teste de Caixa Branca
  - Comportamento é analisado através da execução de todos os caminhos possíveis de código

Uma boa metodologia de teste deve incluir estes dois modos de teste!

Teste de Unidade x Teste de Integração

- Testes podem ser executados em diferentes etapas de construção do software
- É comum no desenvolvimento de um software se aplicar pelo menos 2 tipos de teste:
  - Teste de Unidade
    - Teste de componentes individuais do programa
  - Teste de Integração
    - Teste de grupos de componentes integrados para compor um sub-sistema ou um sistema

Teste de Unidade

- Processo em que se testa componentes (unidades) isoladamente
- Unidades em desenvolvimento OO são as classes
  - Geralmente, testa-se todos os métodos (ou pelo menos os que não são privados)
  - O estado do objeto após a execução dos métodos pode também ser inspecionado
- Muitas vezes quem elabora o teste é o próprio desenvolvedor da classe

Exemplo de Teste de Unidade da Classe Conta

```
public class TesteContas {

  public static void main(String[] args) {

    MiniJavaSystem system = new MiniJavaSystem();
    Conta conta = new Conta("123",150);
    String n = conta.getNumero();
    system.println(n.equals("123") +" == true");
    conta.creditar(100);
    system.println((conta.getSaldo() == 250) +"
      == true");

    conta.creditar(-100);
    if (conta.getSaldo() == 150)
      system.println("erro na operação crédito");
    ...
}
```

Validando requisitos

Detectando erros

Automatizando Testes de Unidade

- Algumas ferramentas auxiliam a geração de testes de unidade para Java
  - JUnit (http://www.junit.org)
  - TestNG (http://www.ibm.com/developerworks/java/library/j-testng/)

Exemplo de Teste de Unidade da Classe Conta com *JUnit*

```
import org.junit.Assert;
import org.junit.Test;

public class ContaTeste{

  ...
  @Test
  public void testCreditar() {
       try {
              Conta c = new Conta("123", 400);
              c.creditar(-300);
              fail("IllegalArgumentException esperado");
       } catch(IllegalArgumentException ie){
              console.println("Teste com sucesso");
       }

 }
}
```

Detectando erros

Teste de Integração

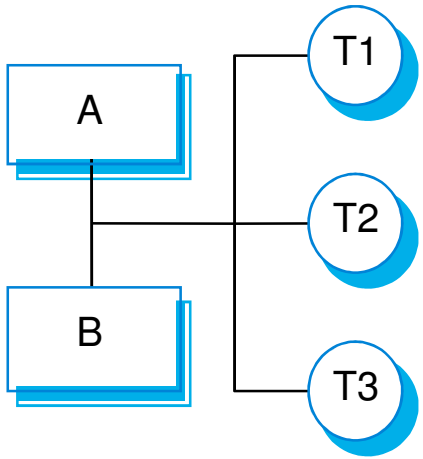
- Processo em que vários componentes são integrados para formar um (sub)sistema e se testa se esta integração resulta no comportamento esperado
- Pode ser um processo incremental, onde a cada etapa um novo componente é integrado
  - Facilita a localização do erro
- Procura-se detectar erros na integração
- É comum que o teste de integração seja realizado por uma equipe de teste independente

A validação de componentes por testes de unidade NÃO garante que a integração destes mesmos componentes será bem sucedida!

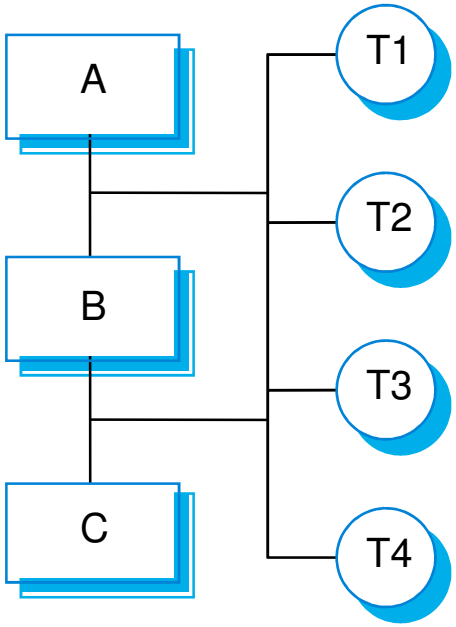
Teste de Integração

Tn - Teste n

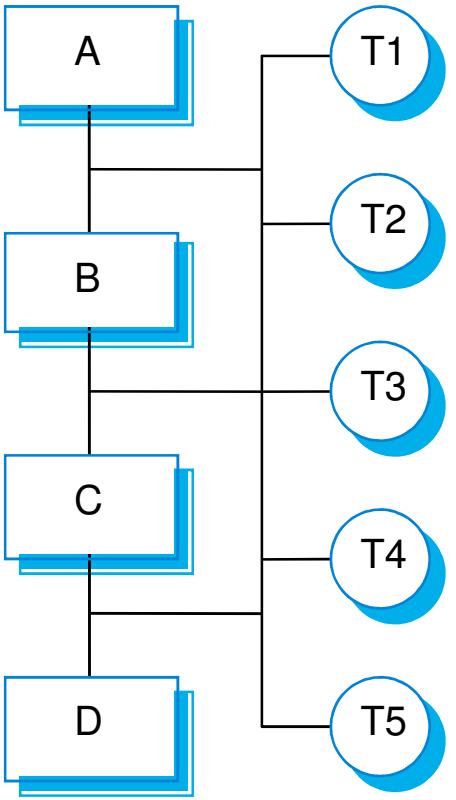
N - Componente n



1ª Integração



2ª Integração



3ª Integração

Test First Design

- ***Test First Design*** é uma técnica de desenvolvimento de software que tem como idéia principal o desenvolvimento de testes de unidade antes mesmo de se implementar a unidade propriamente dita

- Foco nos requisitos antes de realmente escrever o código

Test First Design

- É uma técnica preconizada pela metodologia de desenvolvimento de software Extreme Programming (XP) (http://www.extremeprogramming.org/)

- Com ***Test First Design***, os testes devem ser escritos para que inicialmente o componente falhe
  - Adiciona-se, então, código ao componente para que este passe a funcionar
  - O código do componente vai sendo construído incrementalmente a cada falha no teste

Vantagens de Test First Design

- Código da unidade será testável, pois foi concebido a partir de um teste
  - ⬆ Testabilidade
- Cada unidade tende a ser pequena e bem delimitada
  - ⬆ Modularidade

Vantagens de Test First Design

- Cobertura de teste tende a ser maior, pois cada vez que o teste falha, código é adicionado a unidade
  - Teste é desenvolvido para expor erros do programa
  - ⬆ Corretude e ⬆ Robustez
- Favorece ao uso de técnicas de refactoring
  - Como o teste é concebido antes, a cada refactoring é só aplicar o mesmo teste

Construindo Conta com Test First Design

```
public class Conta {

    String numero;
    double saldo;
    public Conta(String num, double saldo) {}

    public String getNumero() {
      return numero;
    }

    public void creditar(double v) {}

    public double getSaldo() {
      return saldo;
    }
}
```

Inicialmente construtor e método `creditar` estão vazios!

Teste de Unidade da Classe Conta

```
public class TesteContas {

  public static void main(String[] args) {

    MiniJavaSystem system = new MiniJavaSystem();
    Conta conta = new Conta("123",150);
    String n = conta.getNumero();
    system.println(n.equals("123") +" == true");
    system.println((conta.getSaldo() == 150) +"
      == true");
    ...
}
```

Testando construtor

Inicialmente construtor falhará!

Modificando Construtor de Conta para Passar no Teste

```
public class Conta {
    String numero;
    double saldo;
    public Conta(String num, double saldo) {
       this.numero = num;
       this.saldo = saldo;
    }

    ...

    public void creditar(double v) {}


}
```

Testando creditar

```
public class TesteContas {

  public static void main(String[] args) {

    MiniJavaSystem system = new MiniJavaSystem();
    Conta conta = new Conta("123",150);
    ...
    conta.creditar(100);
    system.println((conta.getSaldo() == 250) +"
      == true");
    ...
}
```

Testando creditar

Modificando `creditar`

```
public class Conta {
    String numero;
    double saldo;
    public Conta(String num, double saldo) {
       this.numero = num;
       this.saldo = saldo;
    }

    ...

    public void creditar(double v) {
       saldo = saldo + v;
    }


}
```

Mais um Teste em `creditar`

```
public class TesteContas {

  public static void main(String[] args) {
    MiniJavaSystem system = new MiniJavaSystem();
    Conta conta = new Conta("123",150);
    ...
    conta.creditar(100);
    system.println((conta.getSaldo() == 250) +"
      == true");
    try {
      conta.creditar(-100);
      system.println("Erro:
        IllegalArgumentException esperado");
    }catch(IllegalArgumentException iae){
       system.println("Ok:"+(conta.getSaldo() ==250)
            +"== true");
    } ...
}
```

Testando `creditar` para valores negativos

Modificando de Novo `creditar`

```
public class Conta {
    String numero;
    double saldo;
    public Conta(String num, double saldo) {
        this.numero = num;
        this.saldo = saldo;
    }
    ...
    public void creditar(double v) {
        if (v > 0){
            saldo = saldo + v;
        } else {
            throw new IllegalArgumentException();
        }
    }
}
```

Deve lançar exceção para valores negativos

Não precisa ser declarado no cabeçalho do método, pois é filho de `RuntimeException`

O Método `equals` de Java...

```
x.equals(y)
```

- Para variáveis **`x`** e **`y`** do mesmo tipo (classe ou interface), é pré-definido em toda classe Java
- É equivalente a **`x == y`**, mas pode ser redefinido pelo programador para comparar não as referências mas os valores dos atributos dos objetos

O equals de Conta

```
class Conta {...
  boolean equals(Conta c) {
    boolean r;
    r =     (numero == c.getNumero())
        && (saldo == c.getSaldo());
    return r;
  }
}
```

Compara os valores dos atributos do objeto no qual o método está sendo executado com os valores correspondentes do objeto cuja referência é passada como parâmetro

Resumindo ...

- Teste de Software
  - Importância
  - Objetivos
  - Cobertura
  - Teste de Unidade
  - Teste de Integração
- Test First Design
  - Idéia Principal
  - Vantagens
  - Construindo a classe Conta a partir de um teste de unidade
- Diferença entre operador == e o método equals
  - Quando podemos e devemos utilizar método equals