# Revisão da Prova 2

Eduardo Figueiredo
http://www.dcc.ufmg.br/~figueiredo
dcc603@dcc.ufmg.br

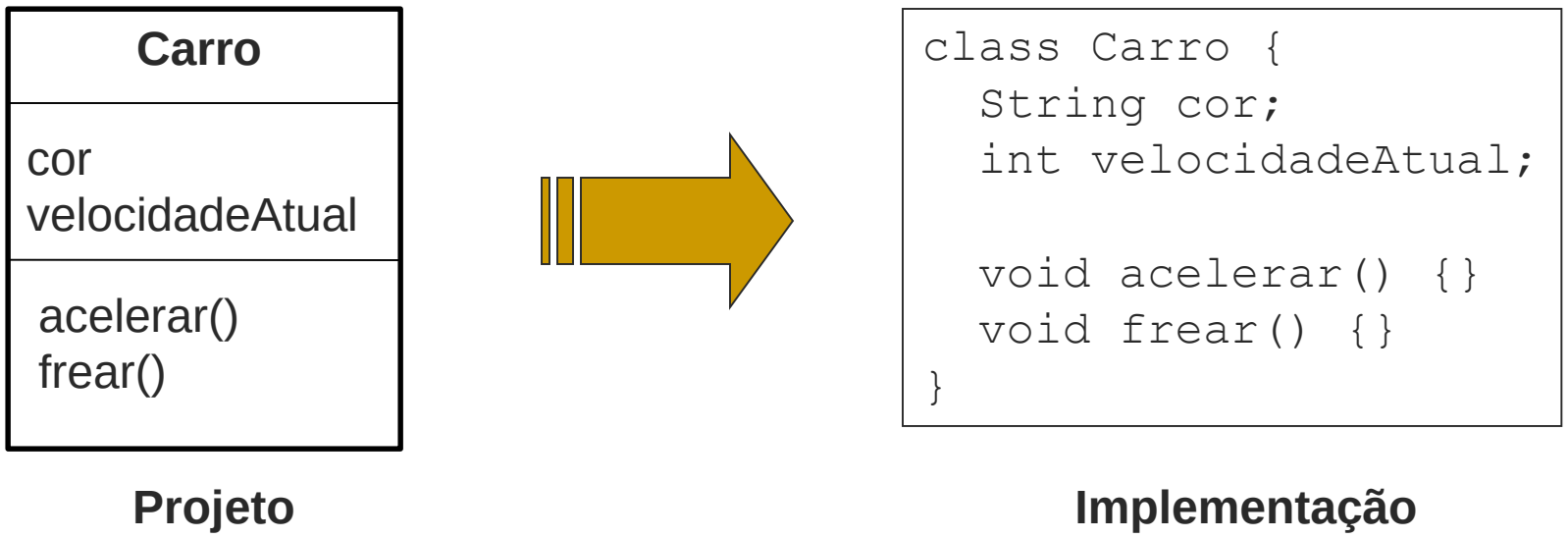
**16 Maio 2018**

# Aula 16: POO

- Um programa OO é geralmente constituído de várias classes
  - Cada classe possui vários métodos (comportamento)
  - Classes também possuem atributos (estados)
- Objetos são criados por uma classe
  - Objetos trocam mensagens pela chamada de métodos

# Do Projeto para Implementação

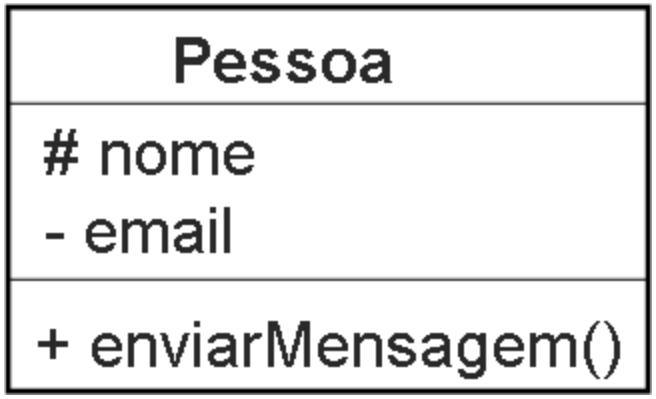
- Antes do carro ser codificado, ele deve ser projetado



# Visibilidade em UML

- Pública (+)
  - O atributo ou método pode ser utilizado por qualquer classe
- Protegida (#)
  - Somente a classe ou sub-classes terão acesso
- Privada (-)
  - Somente a classe terá acesso

# Visibilidade UML x Java




```
public class Pessoa {

   protected String nome;
   private String email;

   public void enviarMensagem() {
      ...
   }
}
```

# Associação em Java

```
public class AlarmClock {

   protected Time alarm;

   public void setAlarm(Time1 time) { alarm = time; }
   ...
}
```

# Membros de Uma Classe

- Construtor
- Métodos
  - De classe
  - De objeto
- Variáveis
  - De classe
  - De objeto
- Constantes

# Idiomas e Estilos

- Um conjunto de idiomas definem um estilo de programação
  - O estilo de programação é definido pela forma como são usadas as construções da linguagem
- Exemplos de idiomas
  - A forma como os *loops* são usados
  - O formato de nomes
  - A formatação do código fonte

# Uma Classe por Arquivo

- Deve-se declarar uma única classe por arquivo Java
  - A única classe do arquivo deve ser pública para que outras classes tenham acesso
- Exemplo

1

**Arquivo Carro.java**

```
public class Carro {
   String cor;
   int velocidadeAtual;

   void acelerar() {}
   void frear() {}
}
```

# O Método Main

- Deve-se colocar o método *main()* em uma classe separada

2

  - Apenas código de iniciação do sistema deve estar na classe que contém o método *main()*

3

- Exemplo

```
public class TesteCarro {
   public static void main(String[] args) {
      ...
   }
}
```

# Expressões

- Evite expressões lógicas complexas como condição de um *if*
  - Particione-as em vários comandos *if* aninhados

21

- Todos os blocos { } vazios devem receber um comentário indicando que estão propositalmente vazios

22

# Bibliografia


- DEITEL, H. M.; DEITEL P. J. **Java: Como Programar**, 8a. Edição. Pearson, 2010.
  - Seções 1.5 a 1.10
  - Capítulos 3, 8 e 9
- A. von Staa. **Programação Modular**. Elsevier, 2000.
  - Apêndices 3, 4 e 5

# Aula 17: Exercício Java

- Implementar o sistema em Java
  - Seguir os idiomas de programação
  - Considerar os objetos principais (vide  Diagrama de Classes feito em exercício anterior)
- Não precisa estar 100% implementado
  - Vou olhar principalmente os conceitos de implementação OO

# Aula 18: V&V e Testes

- Objetivos da verificação e validação
  - Mostrar que o software atende a sua especificação
  - Mostrar que o software atende as necessidades do cliente

- Teste é a principal técnica de V&V
  - Técnicas de inspeção e revisão também são usadas

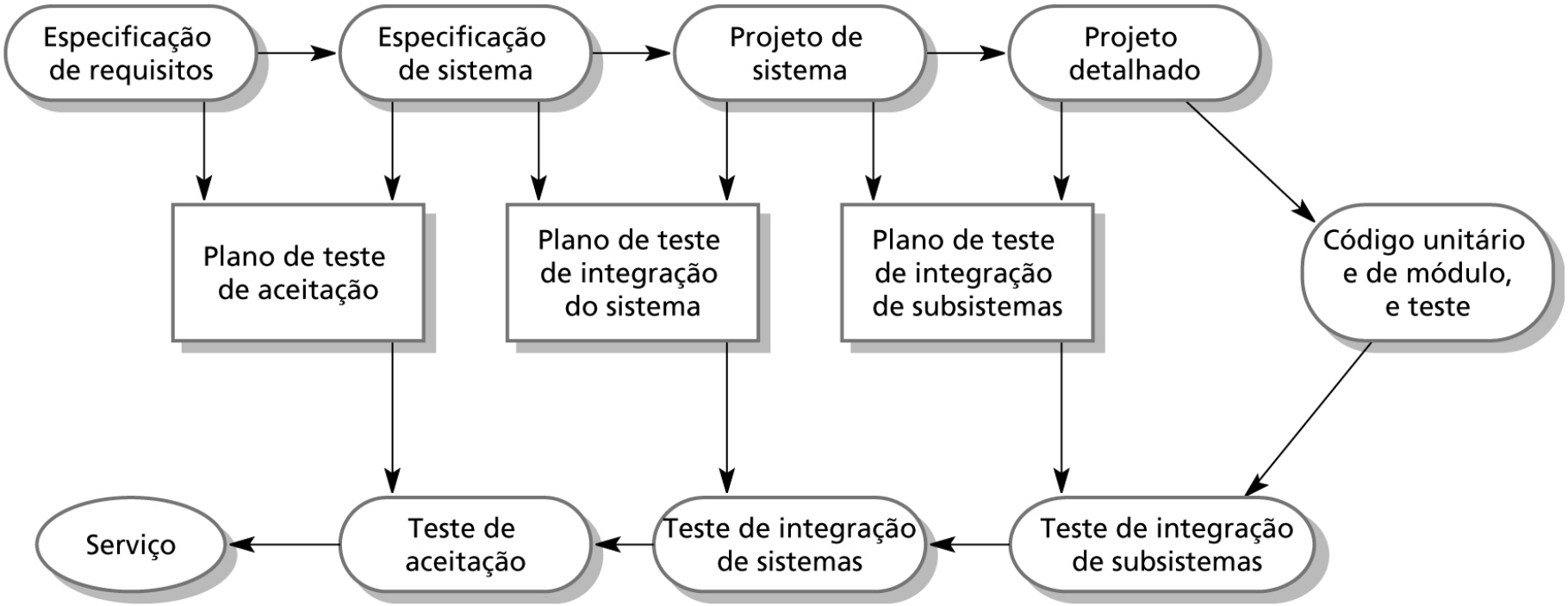
# Verificação

- O objetivo é verificar se o software atende aos requisitos funcionais e não funcionais especificados
- Verificação inclui da realização de testes para encontrar erros
- Pergunta principal
  - *Estamos construindo o produto da maneira correta?*

# Validação

- A inexistência de erros não mostra a *adequação operacional* do sistema
  - Deve ser feita a **validação** com o cliente
- A validação procura assegurar que o sistema atenda as expectativas e necessidades do cliente
- Pergunta principal
  - *Estamos construindo o produto correto?*

# O Modelo V


Especificação de requisitos
Especificação de sistema
Projeto de sistema
Projeto detalhado
Plano de teste de aceitação
Plano de teste de integração do sistema
Plano de teste de integração de subsistemas
Código unitário e de módulo, e teste
Serviço
Teste de aceitação
Teste de integração de sistemas
Teste de integração de subsistemas

# Inspeção de Software

- *"Testes podem somente revelar a presença de defeitos, não a ausência"*

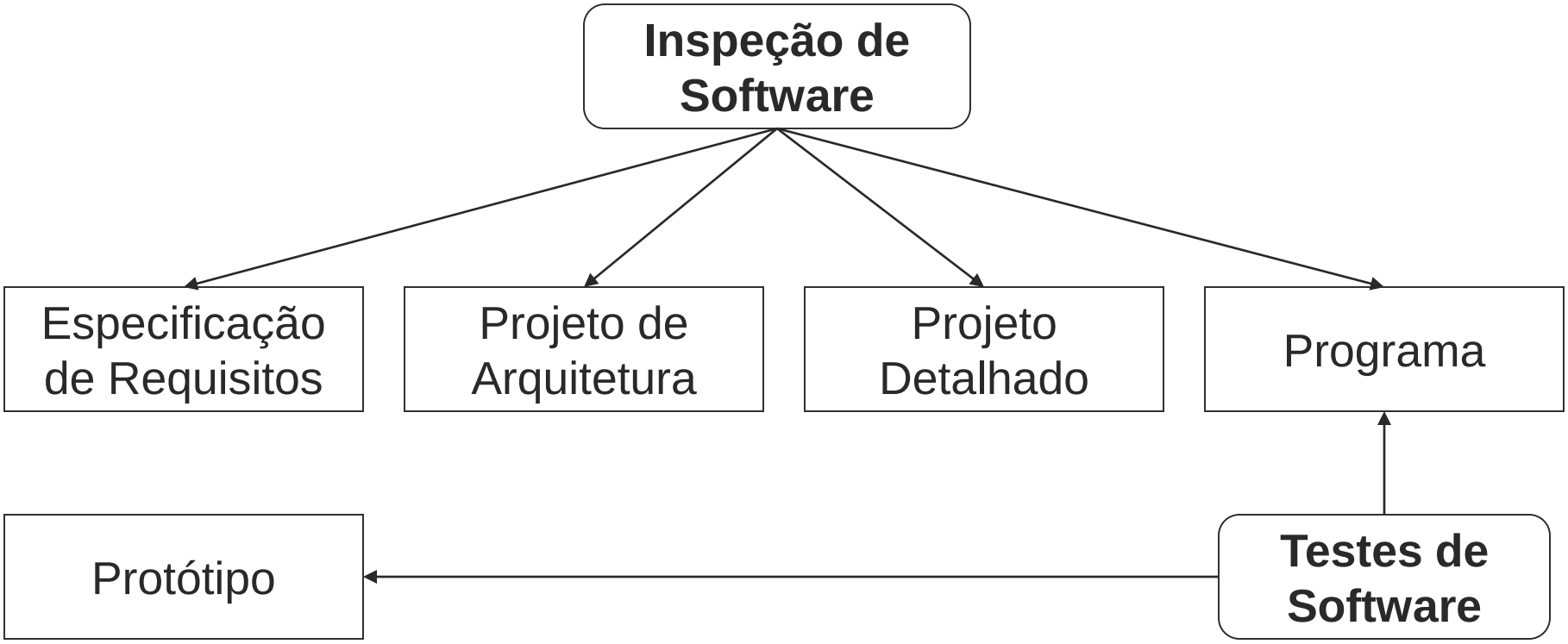
  **Dijkstra**

- Testes fazem parte do processo de verificação e validação (V&V)
  - Devem ser usados em conjunto com a verificação estática (inspeção)

# Inspeções de Software


- Técnica estática de V&V
  - Não precisa executar o programa
- Pode ser usada em qualquer atividade de desenvolvimento
  - Requisitos, projeto, código fonte, etc.
- Pode ser semi-automatizada por análise estática
  - Análise não automatizada é cara

# Verificação Estática e Dinâmica



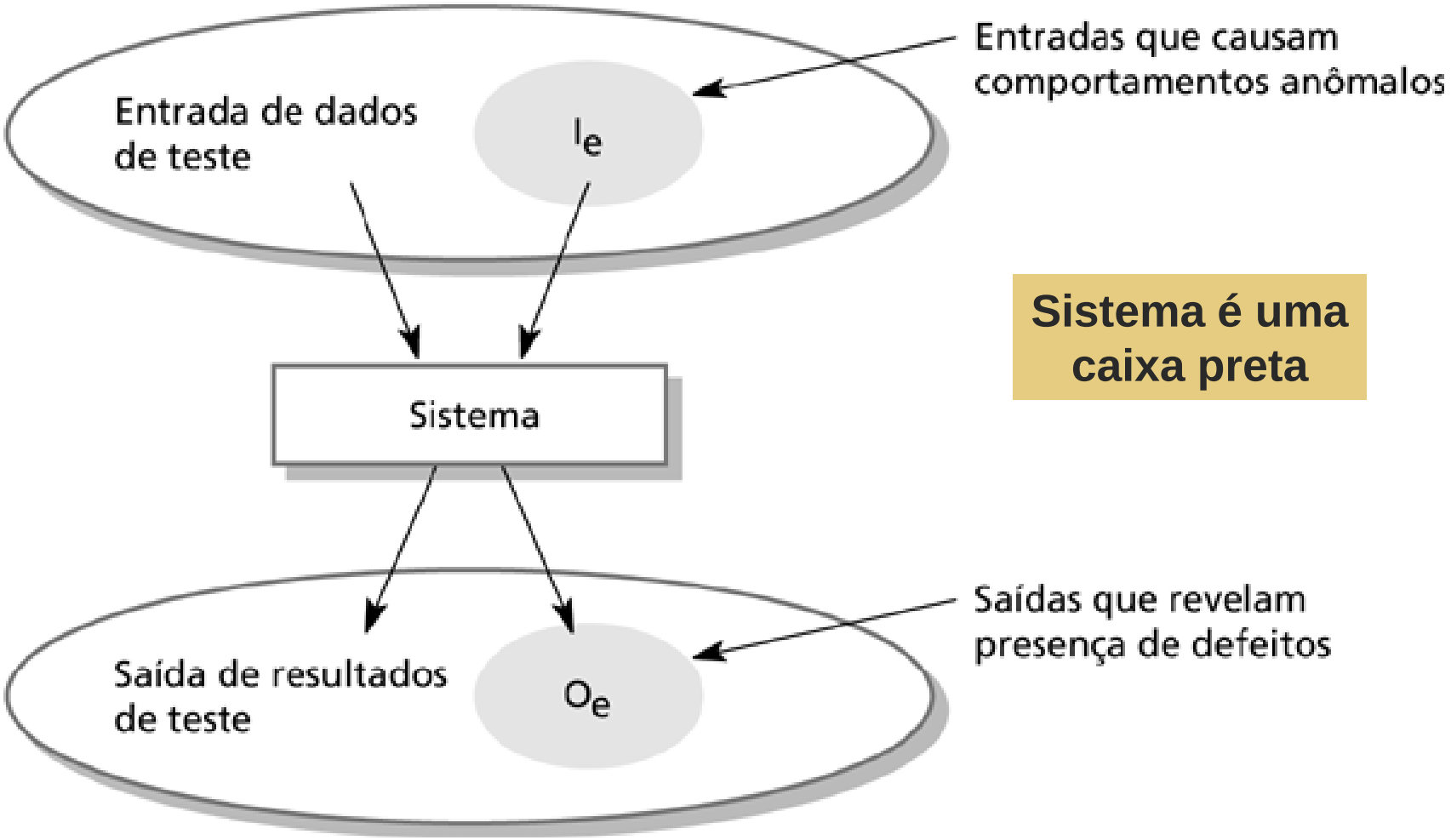
# Teste de Validação

- Pretende mostrar que o software atende aos seus requisitos
  - Faz o que o cliente deseja
- Um teste bem sucedido mostra que o requisito foi implementado
- Refletem o uso esperado do software

# Teste de Defeito

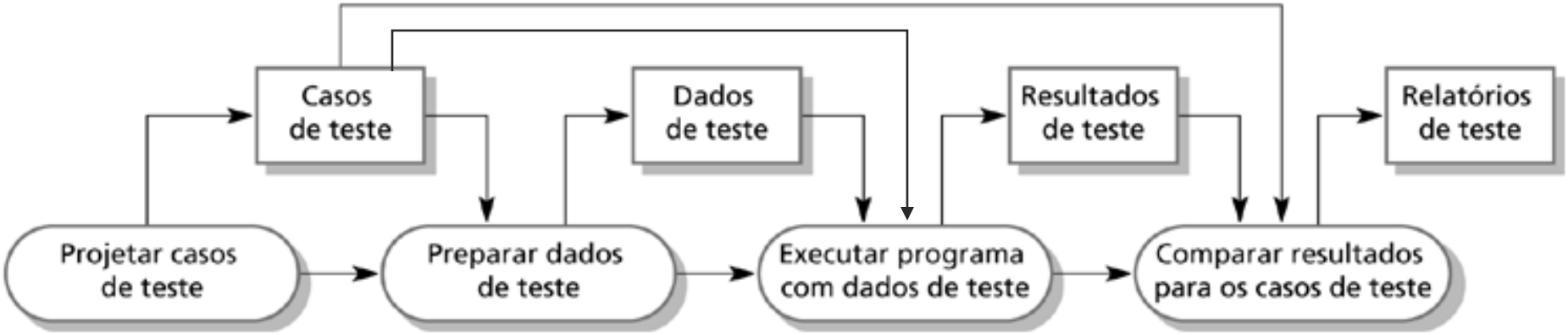
- Destinado a revelar defeitos no sistema
- Um teste de defeitos bem sucedido é aquele que revela defeitos no sistema
- Os casos de teste podem ser obscuros
  - Não precisam refletir exatamente como o sistema é normalmente usado

# Modelo de Entrada e Saída



# Processo de Teste

- Exemplo de processo de teste baseado em planos
  - As atividades são planejadas antes de serem executadas



# Automatização de Testes

- Sempre que possível, os testes de unidade devem ser automatizados

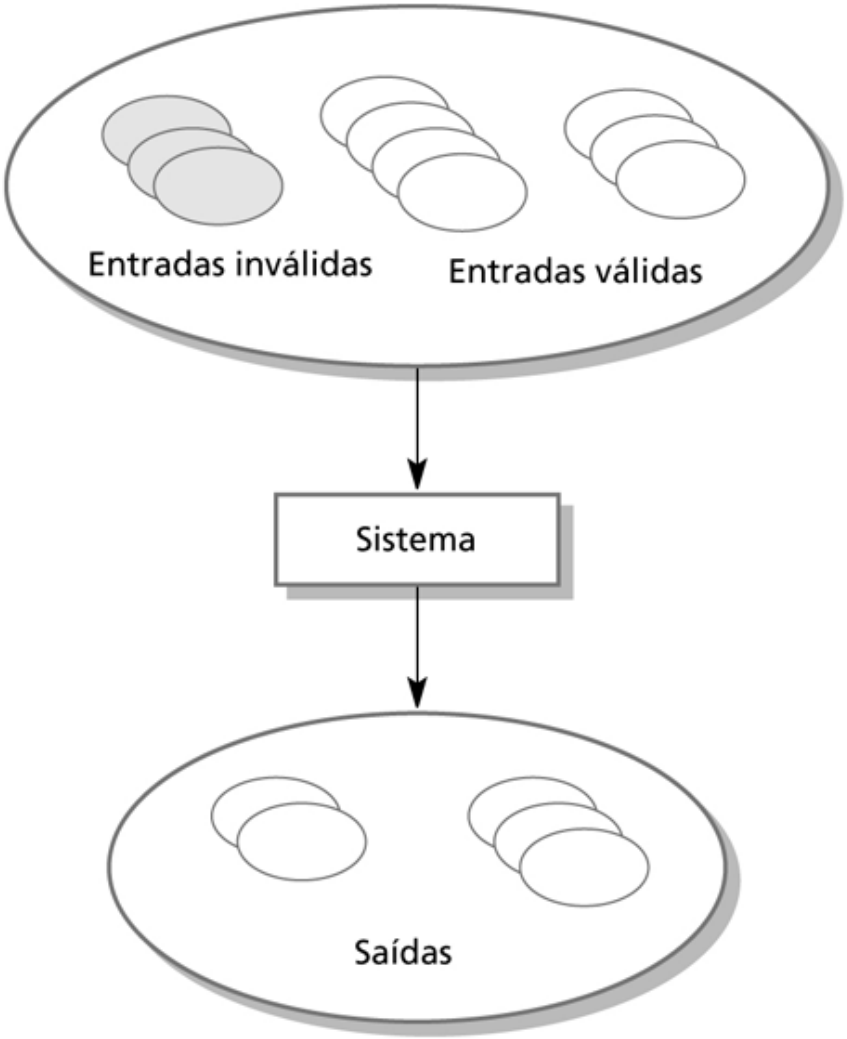
- Um teste automatizado têm três partes
  - **Configuração**: inicia o sistema com o caso de teste e dados de entrada
  - **Chamada**: chama o objeto a ser testado
  - **Afirmação**: compara o resultado da chamada ao resultado esperado

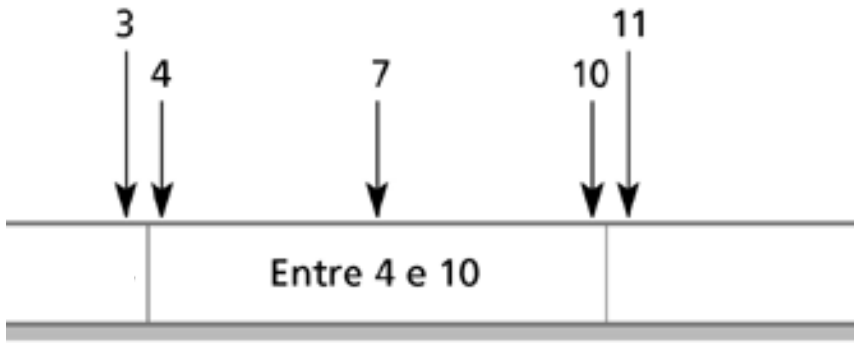
# Escolha do Caso de Teste

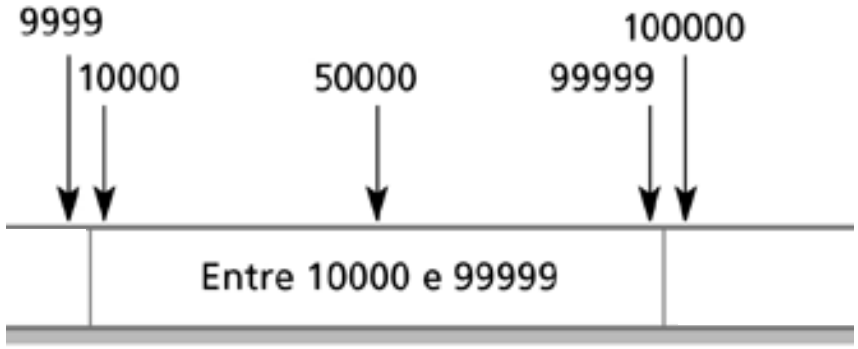
- Teste de software é caro
  - Portanto, é importante a escolha de casos de testes efetivos

- Estratégias para escolha dos testes
  - Teste de partições (caixa preta)
  - Teste estrutural (caixa branca)

# Exemplos de Partições

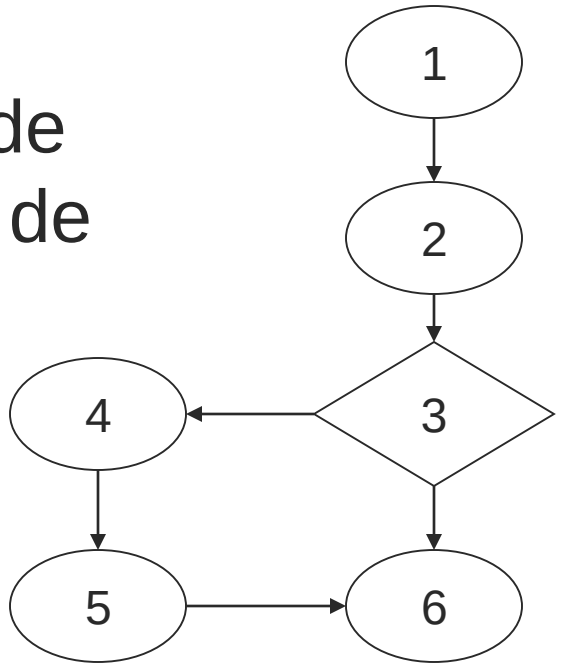


# Teste Estrutural de Caminho

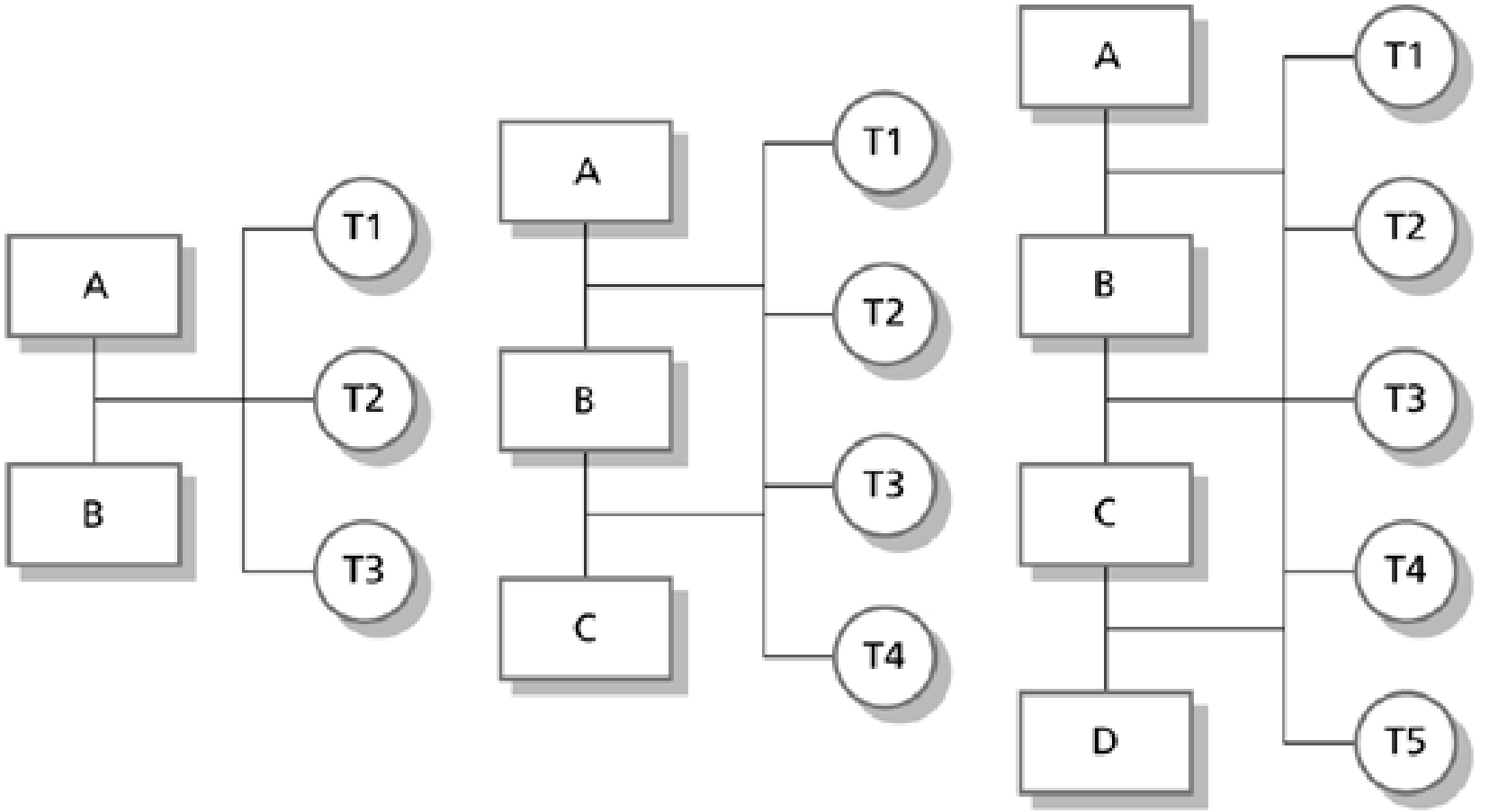
- O objetivo é assegurar que cada caminho do programa é executado pelo menos uma vez
  - Ponto de partida do teste de caminho é um fluxograma de programa




1 – 2 – 3 – 4 – 5 – 6

1 – 2 – 3 – 6

# Integração Incremental




Seqüência de testes 1 Seqüência de testes 2 Seqüência de testes 3

# Teste de Aceitação

- Uma versão particular do software é testada para uso **fora do ambiente de desenvolvimento**
  - Teste após implantação no ambiente do cliente
- O teste é caixa-preta
  - Nenhum conhecimento interno da estrutura do software é necessária

# Testes de Desempenho e Estresse

- Testes de desempenho
  1. Planejamento de uma série de testes
  2. A carga é constantemente aumentada
  3. Verifica o limite em que desempenho do sistema se torne inaceitável

- Testes de estresse forçam o sistema além de sua carga máxima de projeto

# Bibliografia


- Ian Sommerville. **Engenharia de Software**, 9ª Edição. Pearson Education, 2011.
  - Seção 2.2.3 Validação de Software
  - Cap. 8 Testes de Software

# Aula 19: Exercício JUnit

1. Criar testes automatizados (JUnit) para as funcionalidades implementadas na aula prática anterior

- Não precisa testar 100% do que foi implementado
  - Vou olhar principalmente os conceitos de OO e testes automatizados

# Estrutura do Teste

```
package tests;

import static org.junit.Assert.*;
import org.junit.Test;

public class PrimeiraoTests {

  @Test
  public void testTemplate() {

  }

  @Test
  public void test1() {
    ...
  }
  ...
}
```

**Opções de assertivas:**

assertEquals(“mensagem”, v1, v2);
assertTrue(“mensagem”, expressao);
assertFalse(“mensagem”, expressao);
assertNull(“mensagem”, referencia);
assertNotNull(“mensagem”, valor);

(...)

# Teste dos Requisitos

- Exemplos de Requisitos:
  1. O departamento tem pelo menos 1 professor
  2. O aluno é aprovado com nota >= 60 pontos e frequência >= 75%
- Exemplos de Testes:
  1. Garantir que o departamento tenha pelo menos um professor
  2. Garantir que o aluno seja considerado aprovado quanto atender as condições