Centro de Computação e Tecnologia da Informação

TDD
Test-Driven Development

Engenharia de Software III

Prof. Daniel Luis Notari

Agosto - 2012

O que é TDD?

- *Test-Driven Development*: Desenvolvimento Guiado por Testes
- Os testes
  - são feitos ANTES da implementação, diferentemente da abordagem tradicional,
  - em que os testes são feitos DEPOIS da implementação (*)
- A **autoria** é comumente creditada a Kent Beck, embora o conceito não seja de todo original
- Referências:
  - Beck, K. Test-Driven Development by Example, Addison Wesley, 2003
  - Link, J. Unit Testing in Java: How Tests Drive the Code, Morgan Kaufmann, 2003

Premissas do *TDD*

- Os testes são definidos em função dos requisitos
- A implementação é definida em função dos testes
  - Cada acréscimo funcional precisa ser justificado por um teste (*)
  - Busque sempre a implementação mais simples que atenda ao teste
  - Se você acha que faltou algo no código, escreva mais um teste que justifique a complementação

(*) Em geral, não é boa prática testar questões de layout, como posicionamento ou cores

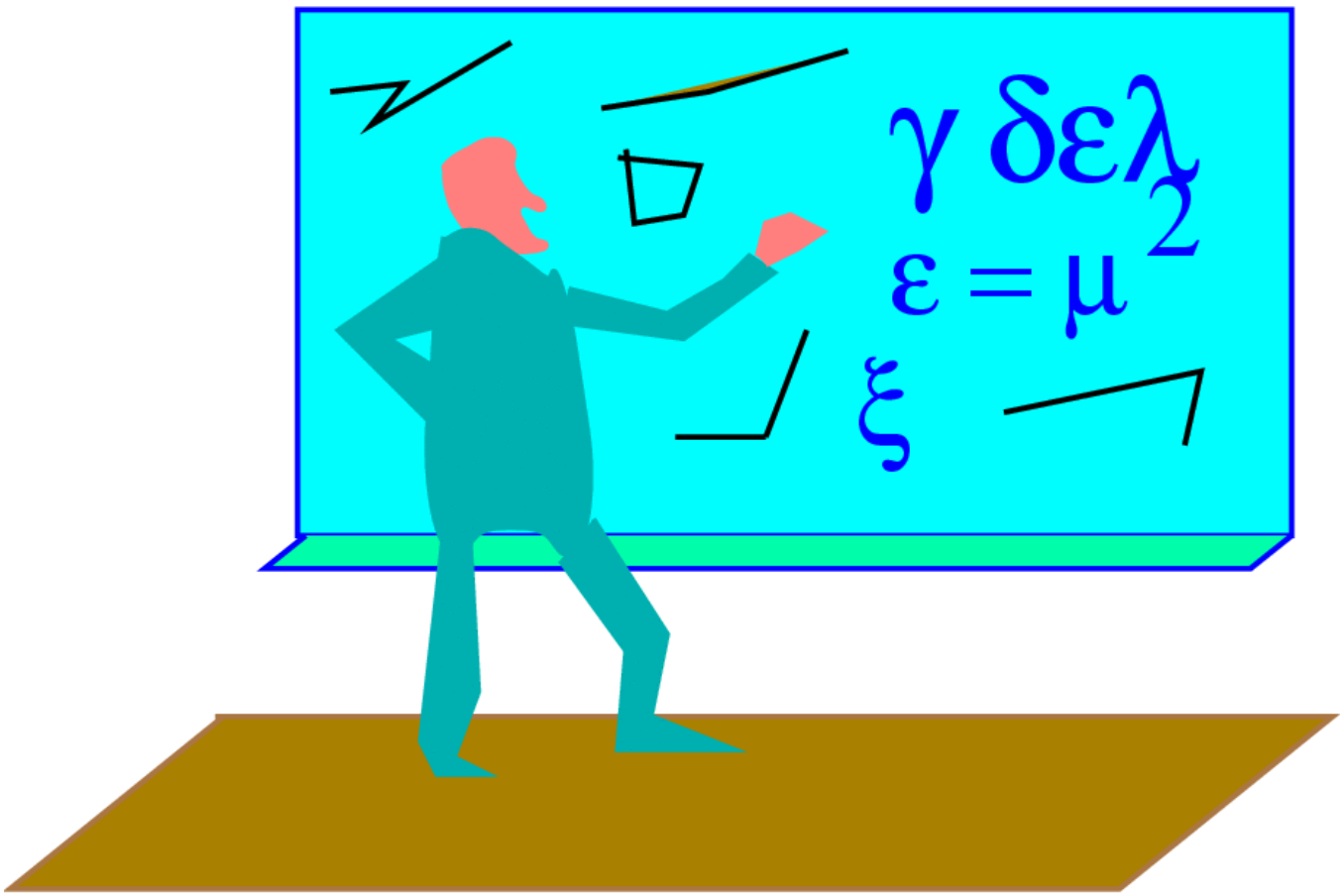
O Processo do TDD (exemplo com um método simples)

1. Criar uma implementação vazia do método (“casca”)
2. Criar um teste para mapear um ou mais requisitos (é recomendável começar simples)
3. Rodar o novo teste, para confirmar que este mapeia o(s) requisito(s) (*Red Bar*)
4. Implementar a solução mais simples que faça todos os testes rodarem
5. Rodar os testes para verificar a implementação (*Green Bar*)
6. Simplificar a implementação, se possível (*Refactoring*)
7. Repetir os passos 2 a 5 até atender a todos os requisitos

Refatoração

- Refatoração é o processo
  - de alterar e otimizar o código de maneira que seu comportamento externo não seja alterado.
- Em código que está funcionando não se mexe?
  - TDD permite que este medo deixe de ter fundamento,
  - pois ele atua como uma rede de segurança, capturando qualquer bug que seja inserido durante o refatoração.

Exemplo de TDD com um método simples



Especificação do método

```
 /**
   * Emite um relatório (array de Strings), indicando quantas vezes cada palavra
   * aparece no array fornecido. O relatório é apresentado na mesma ordem dos dados
   * de entrada, sendo que cada String representa uma palavra e seu número de
   * ocorrências. Por exemplo, para a chamada:
   *
   * countWords(new String[] {"java","in","out","java","java","out"})
   *
   * o relatório gerado é:
   *
   * ["java - 3" , "in - 1", "out - 2"]
   *
   * @param words array com as palavras a serem contadas.
   * @return relatório com o cada palavra e o número de repetições.
   */
  String[] countWords(String[] words) {
      //…
  }
```

Revisando o método

- Requisito(s) atendido(s)? SIM
- O método está robusto? NÃO
- Próximos passos: construir mais testes para mapear aspectos não cobertos

Revisando o processo

- Pontos negativos
  - Desenvolvimento é mais lento
  - Mudança de hábito (inicialmente)
  - Quem cria os testes é o desenvolvedor, portanto ambos os testes e o código de produção podem ter os mesmos erros conceituais
  - A barra verde pode levar a uma falsa sensação de segurança e fazer com que a equipe relaxe nos testes de integração e funcionais

Revisando o processo (cont.)

- Pontos positivos
  - O desenvolvedor pode resolver o problema aos poucos, aspecto a aspecto
  - Testes facilitam o entendimento/documentam dos requisitos
  - Bugs são percebidos mais cedo
    - É mais fácil identificar a causa
    - A correção é menos custosa
    - O aprendizado é melhor
  - Garante uma boa base de testes
  - A arquitetura tende a apresentar baixo nível de acoplamento
  - O código é, naturalmente, facilmente testável
- Consequentemente…Refatoração são menos arriscados

“Nova versão”

1. Novo requisito: case-insensitive
    1. Basta criar novo teste
2. Mudança no requisito: deve-se ordenar do maior para o menor, com os números na frente
    1. Deve-se primeiro ajustar os testes, para em seguida ajustar a implementação
    2. Se encontrar uma solução geral for complicado, comente os testes e vá progressivamente restabelecendo-os
3. *Refatoração*

Análise Abordagem Tradicional

- Implementação primeiro, testar depois o que foi feito
- Muitos bugs só são encontrados quando o desenvolvimento já foi encerrado
- Em caso de erros nas estimativas / pressão entrega, testes são prejudicados
- Se houver poucos testes, *refatoração é* arriscado
- Tendência a tornar o código "genérico" (mais complexo do que necessário) para evitar *refatoração*

Riscos/Dificuldades do uso de TDD

- Resistência da equipe (mudança de cultura)
- Negociação do prazo com o cliente
- Estimativas / Acompanhamento do desenvolvimento

Potenciais benefícios do uso de TDD

- Maior facilidade de evolução do projeto
  - Mudanças/novos requisitos são menos ameaçadores
  - Código é mais simples
- Rotatividade causa menos impacto
  - Testes cobrem/documentam as principais funcionalidades
  - Código é mais legível

JUnit é muito bom, mas…

- Não basta!
- Em uma arquitetura cliente-servidor, posso utilizar o JUnit para testar a construção dos métodos do servidor (testes unitários), mas e a interface gráfica do cliente? Como faço testes funcionais?

Frameworks de Apoio (xUnit)

- Fornecem classes que facilitam a escrita e execução dos testes:
  - .Net Framework (C#, VB.NET, Delphi.NET)
    - NUnit (www.nunit.org)
    - MbUnit (www.mbunit.org/)
  - JUnit (Java)
  - DUnit (Delphi Win32)
  - xUnit (www.xprogramming.com/software.htm)
- Fáceis de aprender

NUnit

- Pode ser usada em qualquer linguagem .NET
- O xUnit mais popular para .Net
- Portada inicialmente a partir do JUnit
- Aproveita vantagens exclusivas da plataforma .Net (atributos, etc)
- Compatível com Mono
- Permite automatizar processo incluindo testes no build gerando logs (inclusive XML).

- www.nunit.org

Você precisa de boas ferramentas!

- TDD é um processo teoricamente possível de ser realizado sem ferramentas - na prática, é inviável
- Existem diversas ferramentas que auxiliam em cada caso (embora não haja “balas de prata”)

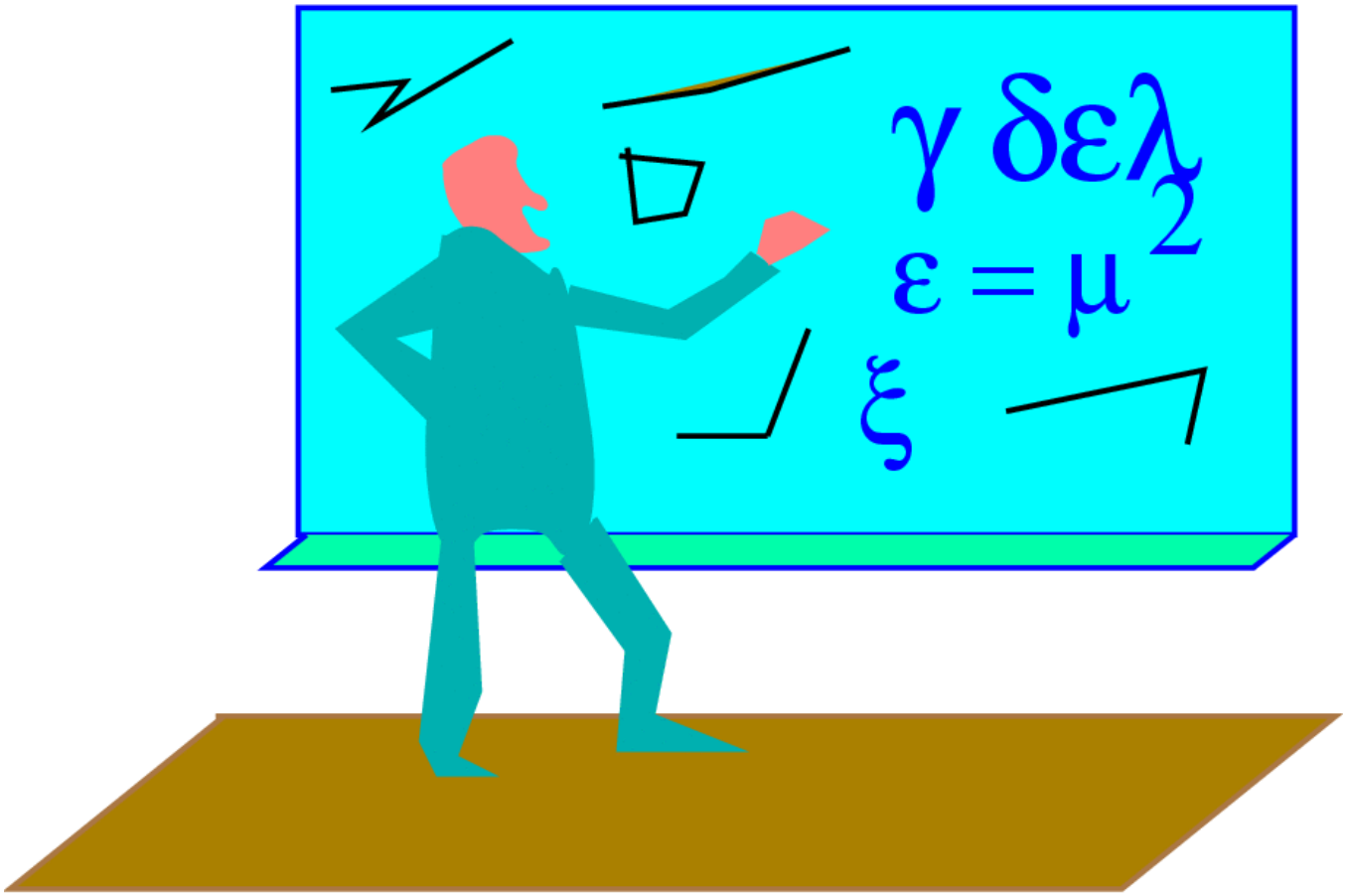
Exemplo de ferramenta: FEST-Swing

- Focada em testes realizados sobre uma interface gráfica Swing (Java)
- Um dos módulos de uma coleção de APIs criada para simplificar o processo de teste
- Referência: http://fest.easytesting.org/swing/wiki/pmwiki.php

Análise da ferramenta

- Pontos fortes
  - Open Source (licença Apache 2.0)
  - Uso bastante intuitivo
  - Extensível
  - Testes são escritos na mesma linguagem da implementação (Java)
  - Comunidade bastante ativa
- Pontos fracos
  - Documentação pobre
  - Jovem (início em meados de 2007)

Exemplo de TDD com uma janela Swing



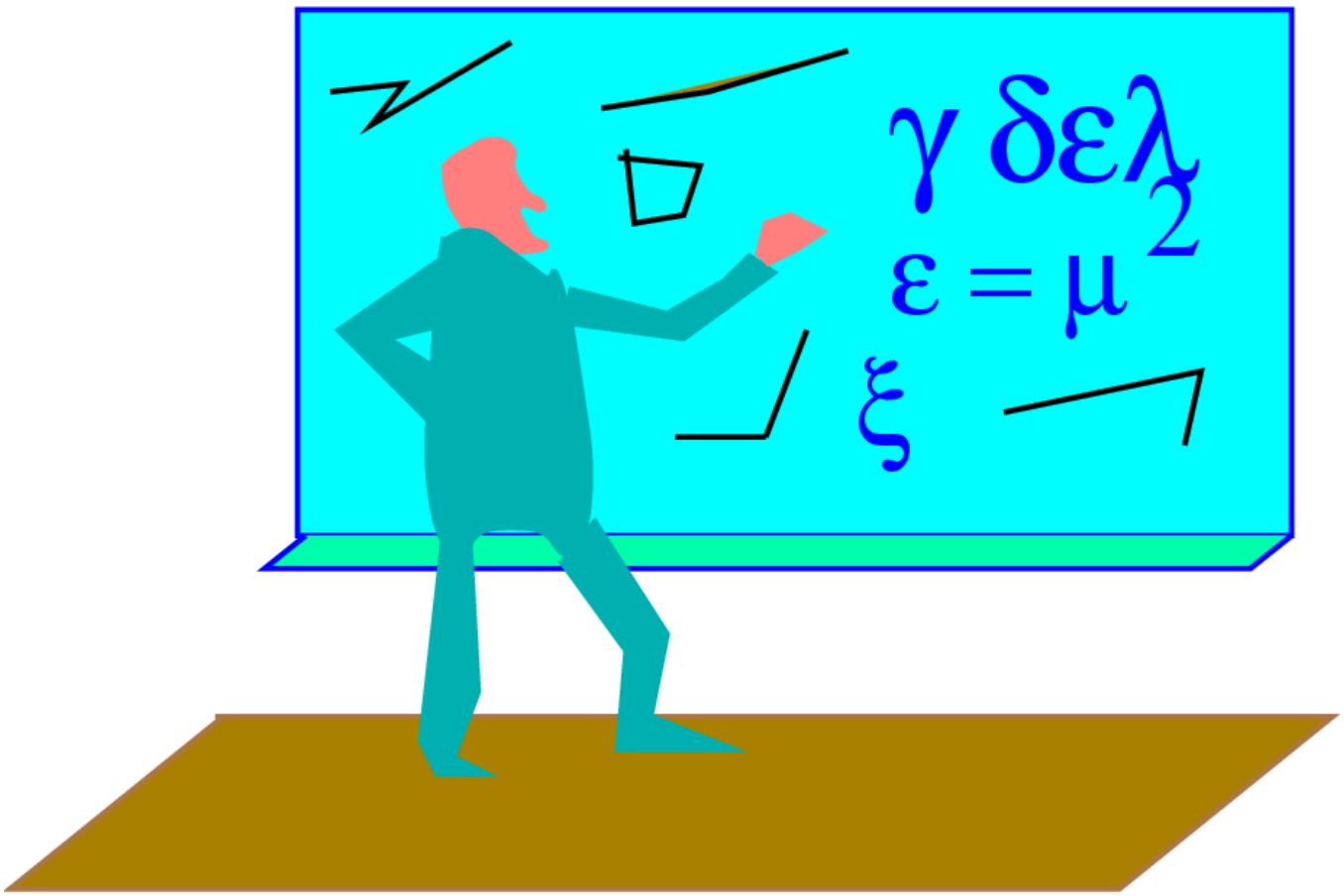
Especificação da janela (Login)

- Deve conter pares rótulo/campo para conta e senha
- Deve conter um botão “Entrar” que irá validar o login e exibir uma mensagem de “bem vindo ao sistema”
- Em caso de conta ou senha errada, deve exibir uma mensagem “login inválido” ao usuário

Até aqui tudo bem, mas...

- E os problemas do “Mundo Real”?
  - Interfaces Gráficas
  - Arquitetura Cliente-Servidor
  - Banco de Dados
  - etc

Exemplo de TDD mais realista



Modelo de trabalho proposto (Desenvolvimento)

1. Definição de requisitos
2. Prototipação
3. Testes de Aceitação iniciais (automatizar se possível)
4. Definição Arquitetura inicial
5. Testes Funcionais iniciais (automatizar se possível)
6. Desenvolvimento
    1. Definição interfaces
    2. Testes unitários
    3. Implementação
7. Testes de Interação

Modelo de trabalho proposto (Correção de bugs)

1. Reprodução manual do problema
2. Implementação de teste automático (ou manual, se o custo for muito alto)
3. Executar teste para garantir que o problema é capturado
4. Corrigir problema
5. Executar teste para garantir que o problema foi resolvido

Modelo de trabalho proposto (*refatoração* em código sem testes)

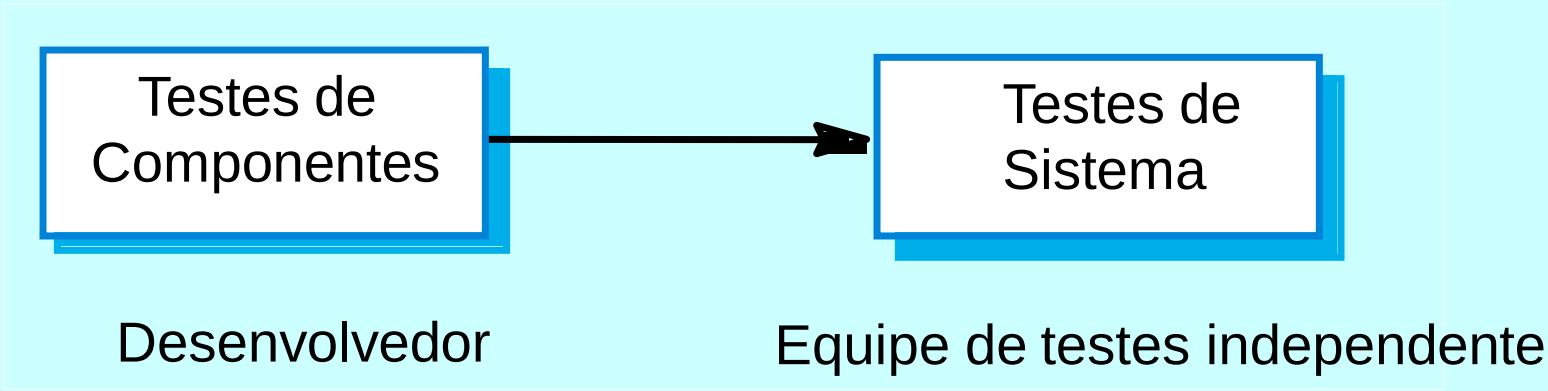
1. Levantamento dos requisitos da interface
2. Implementação de testes automáticos (ou manuais, se o custo for muito alto)
3. Ler o código atual para garantir que todos os aspectos foram tratados (se necessário, escrever mais testes)
4. Rodar testes para garantir que estão corretos
5. Reimplementar o método aos poucos, implementando a solução mais simples para cada teste

Testes tradicionais

O processo de testes

- Testes de Componentes
  - Testar os componentes individuais de um programa
  - Normalmente são de responsabilidade do desenvolvedor do componente
  - Os testes são derivados da experiência do desenvolvedor
- Testes de sistema
  - Testes de grupos de componentes integrados, que formam um sistema ou sub-sistema
  - Responsabilidade de um time de testes independente
  - Os testes são baseados em especificações do sistema

Fases de Testes



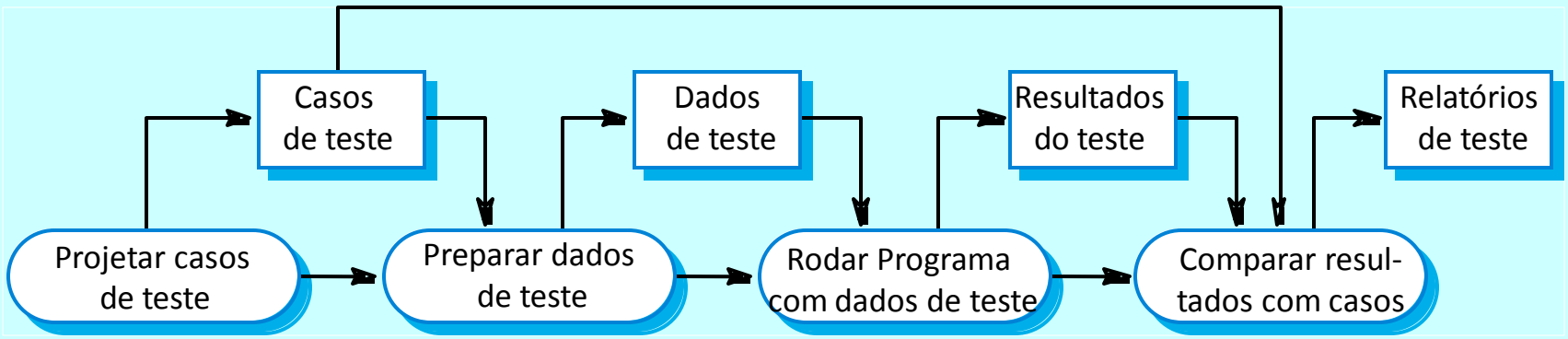
Testes para a detecção de defeitos

- O objetivo é descobrir defeitos em programas
- Um teste bem sucedido é aquele que faz com que o programa se comporte de maneira anômala
- Testes mostram a presença de defeitos
  - Mas não a sua ausência

Objetivos do processo de testes

- Testes de validação
  - Demonstrar para o desenvolvedor e clientes que o software atende seus requisitos
  - Um teste bem sucedido mostra que o sistema opera como pretendido
- Testes de defeito
  - Descobrir falhas no software onde o comportamento é incorreto ou não está de acordo com a especificação
  - Um teste bem sucedido é aquele que mostra estes problemas

O processo de testes de software



Políticas de testes

- Apenas testes exaustivos podem mostrar que um programa está livre de defeitos. Entretanto, testes exaustivos são impossíveis
- Políticas de testes definem a abordagem a ser usada na seleção de testes de sistema.
- Exemplos:
  - Todas as funções acessadas através de menus devem ser testadas
  - Combinações de funções acessadas através do mesmo menu devem ser testadas
  - Quando entradas do usuário são necessárias, todas as funções devem ser testadas com entradas corretas e incorretas

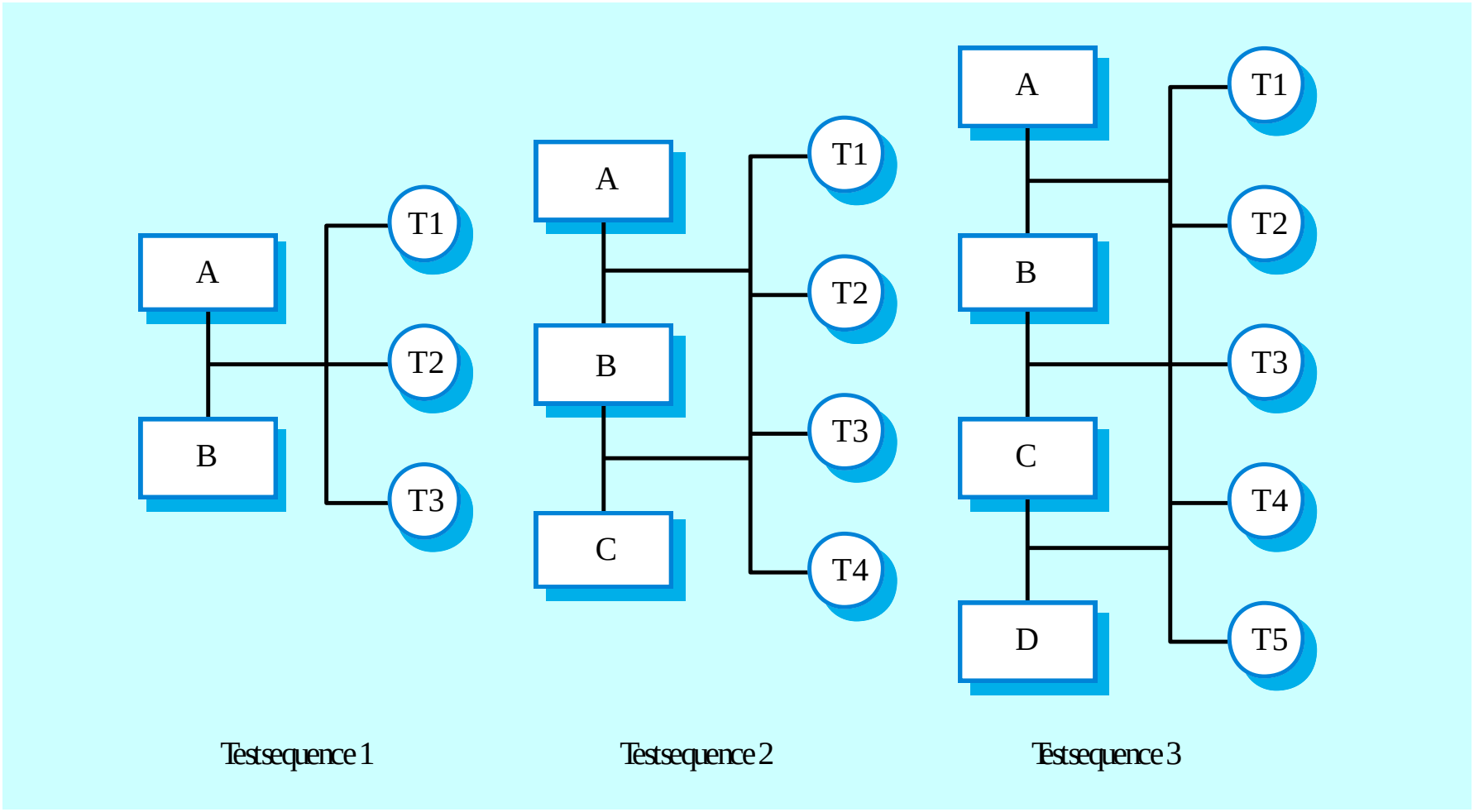
Testes de Sistema

- Envolve integrar componentes de forma a criar um sistema ou sub-sistema
- Pode envolver testar um incremento a ser entregue a um cliente
- Duas fases:
  - Testes de integração: A equipe de testes deve ter acesso ao código fonte. O sistema é testado na medida em que os componentes são integrados
  - Testes de release: A equipe de testes testa o sistema completo a ser entregue como uma ‘caixa-preta’

Testes de integração

- Envolve construir um sistema através de seus componentes e testá-lo em busca de problemas que podem ocorrer na interação entre os componentes
  - Integração top-down
    - Desenvolve-se o esqueleto de um sistema que é populado por componentes
  - Integração bottom-up
    - Componentes de infra-estrutura são integrados e depois são adicionados os componentes funcionais
  - Para simplificar a localização de erros, os sistemas devem ser integrados de maneira incremental

Testes Incrementais



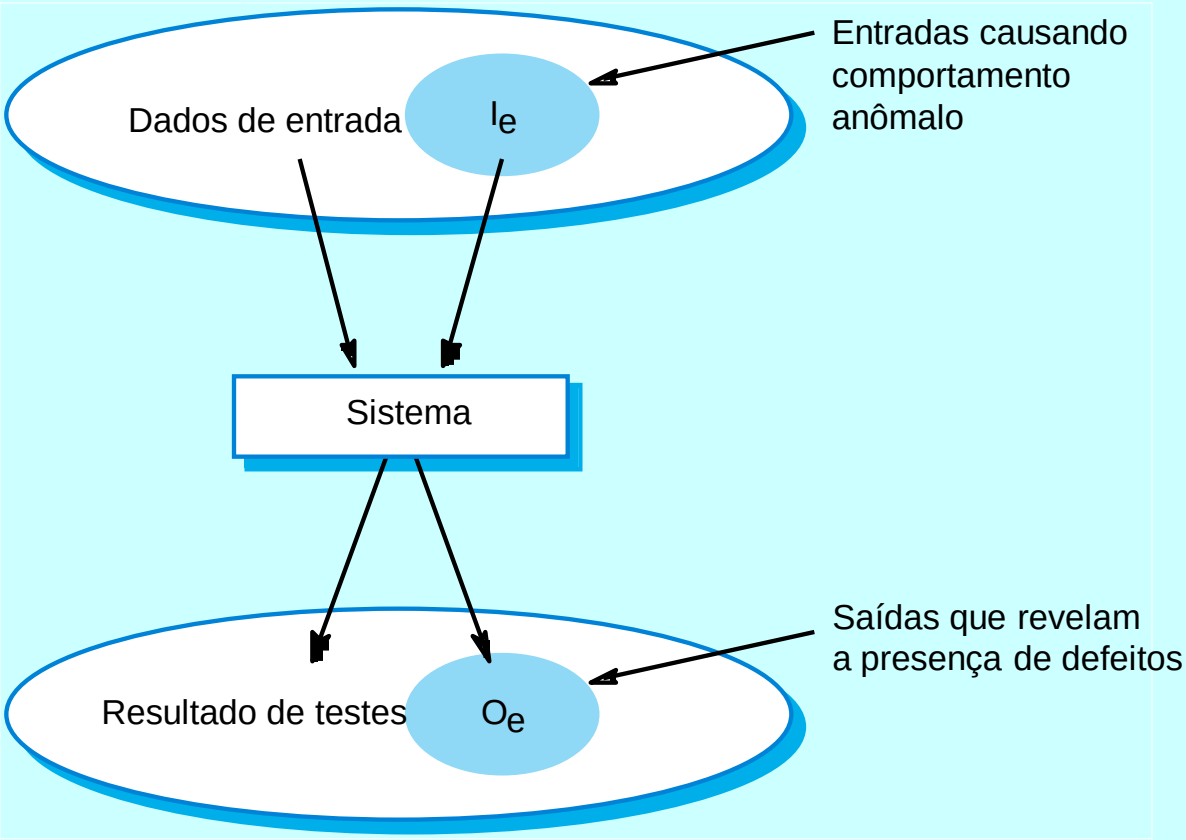
Abordagens de testes

- Validação de arquitetura
  - Testes de integração top-down são a melhor maneira de descobrir erros em uma arquitetura
- Demonstração do sistema
  - Testes de integração top-down permitem uma demonstração limitada do sistema em estágios iniciais do processo de desenvolvimento
- Implementação de testes
  - Normalmente é mais fácil com testes de integração bottom-up

Testes de Release

- É o processo de testar um release de um sistema a ser implantado/distribuído aos clientes
- O principal objetivo é aumentar a confiança de que o sistema atende as necessidades do usuário
- Testes de release são normalmente testes funcionais, caixa-preta
  - Baseados apenas na especificação do sistema
  - Testadores não precisam ter conhecimento dos detalhes de implementação do sistema

Testes Caixa-Preta



Recomendações para Testes

- Recomendações auxiliam a escolher testes que revelam a presença de defeitos no software:
  - Escolha entradas que forçem o sistema a gerar todas as condições de erros
  - Escolha entradas que causem estouro de memória
  - Force que saídas inválidas sejam geradas
  - Faça com que os resultados da computação assumam seus valores máximos e mínimos

Testes de desempenho

- Parte dos testes de release envolvem testar propriedades de um sistema, como desempenho e tolerância a falhas
- Testes de desempenho envolvem planejar uma série de testes onde a carga é incrementalmente aumentada até que o desempenho da aplicação torne-se inaceitável

Testes de stress

- Exercita o sistema além das capacidades máximas projetadas. Normalmente estressar o sistema faz com que problemas venham a tona
- Estressar o sistema visa testar o comportamento de falhas deste sistema.
  - O sistema não deve falhar de maneira catastrófica, devendo informar sobre condições anômalas
- Testes de estresse são particularmente relevantes para sistemas distribuídos, os quais podem exibir degradação quando a rede fica sobrecarregada

Testes de Componentes

- Testes de componentes ou testes unitários visam testar componentes isolados do sistema
- É um processo de descoberta de defeitos
- Componentes podem ser:
  - Métodos de uma classe
  - Classes com diversos atributos e métodos
  - Conjuntos de classes que proporcionam uma fachada de acesso às suas funcionalidades

Testes de Classes

- Uma cobertura completa de testes em uma classe envolve:
  - Testar todas as operações associadas a esta classe
  - Modificar e obter todos os atributos desta classe
  - Exercitar os objetos em todos os seus estados definidos
- Mecanismos de herança podem dificultar o projeto de testes para classes, já que a informação a ser testada pode estar espalhada pela hierarquia de classes

Testes de Interface

- Interfaces podem ser vistas como um contrato de utilização de uma classe por um conjunto de clientes
- O objetivo dos testes de interface é detectar falhas por problemas de interface
- Particularmente importante para sistemas orientados a objetos

Considerações finais

- **Testes** podem mostrar a presença de falhas em um sistema, mas **não podem provar que não existem falhas**
- **Desenvolvedores** de componentes são **responsáveis** pelos **testes de componentes**, mas os **testes de sistema** são normalmente realizados por uma **equipe separada** de testes
- Utilize a sua experiência e recomendações existentes para projetar casos de testes!

Exercício

Projetar e implementar um teste baseado em TDD para a especificação abaixo. Depois deve ser gerado o código para a implementação da história do usuário.

CARTÃO DE VISÃO

Locadora de vídeos

O sistema deverá proporcionar a locação de vídeos, fornecendo um controle dos filmes locados por cada cliente e dos pagamentos a serem realizados por estes clientes, de acordo com as notas fiscais emitidas. A identificação do cliente deverá ser feita através de seu nome e a senha por ele cadastrada.

HISTÓRIA DE USUÁRIO 1

1

2

Cadastro dos filmes

Permite que cada filme adquirido pela locadora possa ser cadastrado. Este cadastro será baseado no código de barras de cada filme, que representará o código do filme. Além do código, deverá conter o nome, atores principais, tipo de filme, diretor, sinopse, tempo de duração, censura (idade mínima), data de cadastramento, preço de compra do filme e o valor da diária. Também deverá apresentar a situação do filme (locado ou disponível) e o número de locações que o mesmo já teve.

TESTE DE ACEITAÇÃO 1

Cenário: a locadora recebe novos filmes.

Operação: os filmes não estão cadastrados no sistema.

Verificação: o código de barras (código) de cada filme será escaneado e seus dados serão cadastrados: nome, atores principais, tipo de filme, diretor, sinopse, tempo de duração, idade de censura, data do cadastramento, número de locações, situação (locado ou disponível), preço de compra e o valor da diária.