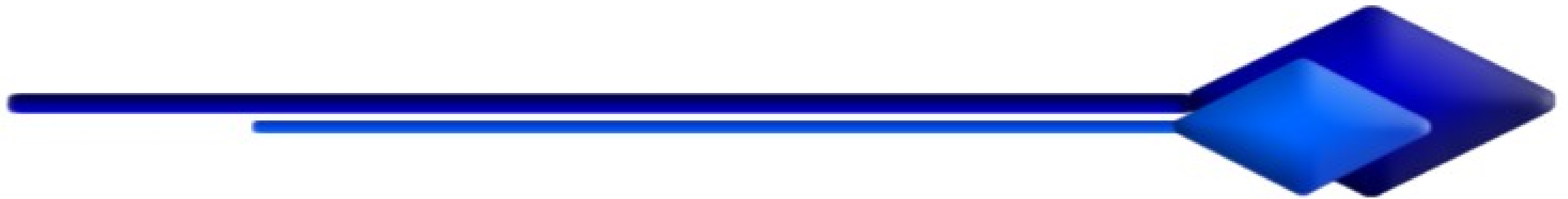
Testes Automatizados



Cursos de Verão 2007 – IME/USP

www.agilcoop.org.br

Dairton Bassi & Paulo Cheque

AgilCoop

Roteiro

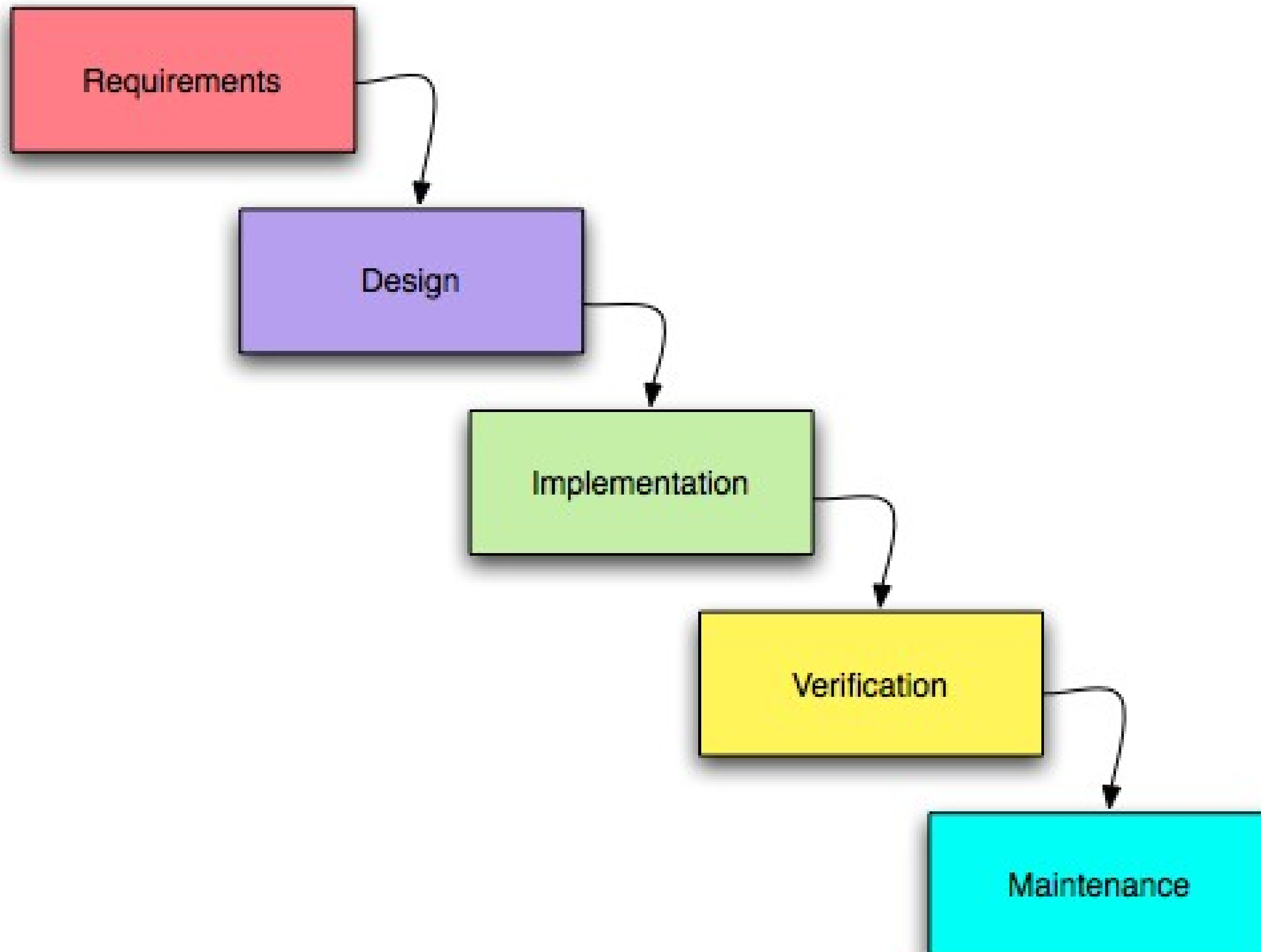
1) Motivação
2) Introdução a Testes
3) Testes de Unidade
4) Testes de Aceitação
5) Testes de Integração
6) Análise de Risco
7) Considerações Finais

Este cenário é familiar?

- Projeto Atrasado
- Prazos curtos
- Implementação em andamento
- Pressão do cliente
- Muito Stress!!!

Naturalmente algo é sacrificado... o que será?

Modelo Cascata



Lembre-se!

Programar é Humano

Testes Manuais

Testar geralmente é:

1) Escolher alguns valores

2) Executar o programa

3) Visualizar o resultado

Problemas

- Demorado
- Implementação muda
- Poucos casos
- Casos simples
- Cansativo
- Difícil repetir os testes
- Executado poucas vezes
- Ninguém quer testar

Conseqüências

- Não é possível confiar nos testes
- Não há garantias de que o software funciona
- Provavelmente há muitos erros
- O software é testado em fase de produção

Solução

- Testes Automatizados
- Metodologias Ágeis

Manifesto Ágil

- Software funcionando é mais importante que documentação completa e detalhada
  - Software funcionando exige testes automatizados
- Colaboração com o cliente é mais importante que negociação de contratos
  - Testes de aceitação
- Adaptação a mudanças é mais importante que seguir um plano
  - Mudanças são seguras com testes automatizados
- Indivíduos e interações são mais importantes que processos e ferramentas

Manifesto Ágil

- Software funcionando é mais importante que documentação completa e detalhada
  - Software funcionando exige testes automatizados
- Colaboração com o cliente é mais importante que negociação de contratos
  - Testes de aceitação
- Adaptação a mudanças é mais importante que seguir um plano
  - Mudanças são seguras com testes automatizados
- Indivíduos e interações são mais importantes que processos e ferramentas
  - Ferramentas de teste trazem conforto para os indivíduos

O que é um Teste Automatizado?

- Código que testa outro código e verifica o resultado
- Um teste que pode ser executado facilmente várias vezes

Não Confunda!

- Testar não é depurar
  - Testar é buscar erros
  - Depurar é seguir o fluxo para identificar um erro conhecido

- “Testes podem mostrar a presença de erros, não a sua ausência” (Dijkstra)

Por que Testar Automaticamente?

- Ajuda a encontrar erros mais cedo
- Novas funcionalidades podem quebrar as funcionalidades antigas
- Dá segurança para refatorações
- Dá suporte para mudanças de configuração no ambiente
- Melhora a qualidade do código, diminuindo o número de erros

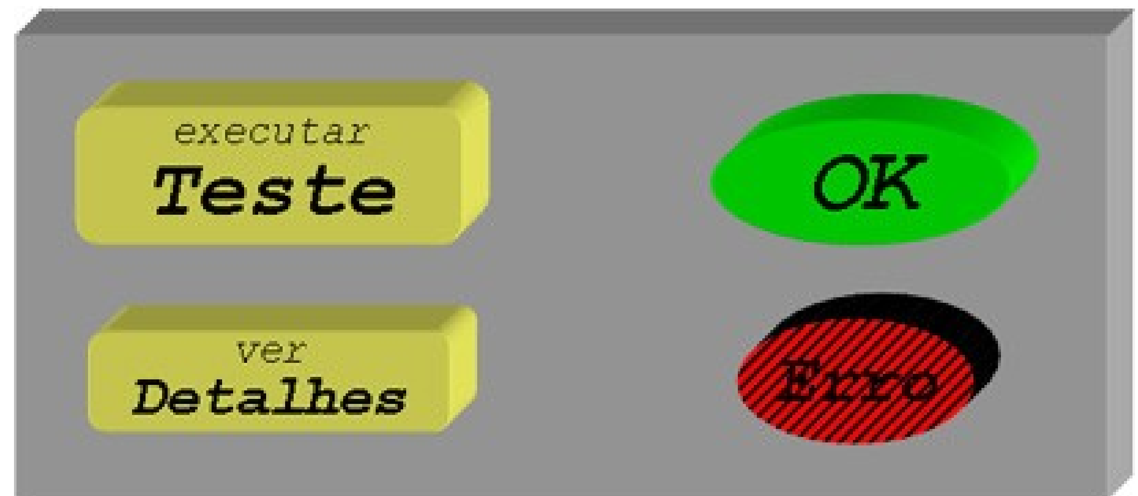
Quando Escrever Testes

- Antes de escrever o código (Test First)
- Durante a implementação
- Antes de uma refatoração
- Quando um erro for encontrado: Escreva um teste que simule o erro e depois corrija-o

Como Executar os Testes

A execução deve ser ágil

- Com ferramentas, scripts ou suite de testes
- Resultado deve ser simples
  - OK ou
  - Lista de Erros



- Não deve demorar

Quando Executar os Testes

- Sempre que uma nova porção de código é criada
- À noite ou em uma máquina dedicada
- Assim que alterações são detectadas no repositório
- **Lembre-se**: É melhor escrever e executar testes incompletos do que não executar testes completos

Tipos de Testes

- Unidade (enfoque desta apresentação)
- Integração
- Aceitação (enfoque desta apresentação)
- Interface
- Desempenho
- Estresse
- Segurança
- …

Testes de Unidade

- Testa um pequeno módulo (geralmente uma classe)
- Não deve se preocupar com as responsabilidades de outros módulos
- Teste “White-box”: É necessário conhecer alguns detalhes do módulo
- **Lembre-se**: Teste a funcionalidade, não a implementação

Exemplo – Uma Classe

```
public class Retangulo {
    private int largura;
    private int altura;

    public Retangulo(int altura, int largura) {
        this.altura = altura;
        this.largura = largura;
    }

    public int calculaArea() {
        return largura * altura;
    }
}
```

Exemplo – Um Teste

```
@Test
public void testaAreaZero() {
    Retangulo r;
    r = new Retangulo(0, 4);
    assertEquals(0, r.calculaArea());
}

@Test
public void testaAreaDeQuadrado() {
    Retangulo r;
    r = new Retangulo(4, 4);
    assertEquals(16, r.calculaArea());
}
```

Set Up e Tear Down

- Os testes devem ser independentes uns dos outros e do número de vezes que é executado
- Set Up: Prepara o ambiente para a execução do teste
- Tear Down: Limpa / Remove o ambiente

Teste com Ambiente

```java
@Test
public final void testBuscarUsuarioPeloNome() throws Exception {
    // Criando ambiente
    UsuarioDAO dao = new UsuarioDAO();
    Usuario usuario1 = dao.criarUsuario("Maria");

    // Testando
    Usuario usuarioEncontrado;
    usuarioEncontrado = dao.buscarUsuarioPeloNome("Maria");
    assertNotNull(usuarioEncontrado);

    usuarioEncontrado = dao.buscarUsuarioPeloNome("José");
    assertNull(usuarioEncontrado);

    // Limpando ambiente
    dao.removerUsuario(usuario1);
}
```

Exemplo Set Up / Tear Down

```
public class UsuarioAdminTest {
    private static UsuarioDAO dao;
    private static Usuario usuario1;

    @BeforeClass
    public static void setUp() throws Exception {
        dao = new UsuarioDAO();
        usuario1 = dao.criarUsuario("Maria");
    }

    @AfterClass
    public static void tearDown() throws Exception {
        dao.removerUsuario(usuario1);
    }

    @Test
    public final void testBuscarUsuarioPeloNome() throws Exception {
        Usuario usuarioEncontrado = dao.buscarUsuarioPeloNome("Maria");
        assertNotNull(usuarioEncontrado);
    }

    @Test
    public final void testBuscarUsuarioDesconhecido() throws Exception {
        Usuario usuarioEncontrado = dao.buscarUsuarioPeloNome("José");
        assertNull(usuarioEncontrado);
    }
}
```

Testando Métodos

- Protegidos: Coloque o código dos testes no mesmo pacote da classe que está sendo testada (boa prática)
- Privados (Um alerta para o design): Utilize reflexão ou arcabouços próprios para isso
- Triviais (ex: get/set): Podem não ser testados

Testando Classes

- Internas: Testar a classe que usa é suficiente
- Abstratas: Instanciar uma subclasse e chamar os métodos da classe mãe
- Classes simples de armazenamento de dados não precisam ser testadas

O Que Testar?

- Valores positivos, negativos, muito pequenos, muito grandes e zero
- Valores Limite
- Strings: nula, vazias, pequenas e grandes
- Coleção: nula, vazia, com vários elementos e com elementos iguais

Dicas

- Teste casos de sucesso
- Teste casos de erro:
  - Tratamento do erro
  - Exceção esperada

```
@Test
public final void testCPFValido() throws Exception {
    CPF cpf = new CPF("11122233344"); // Recebe 11 números
    assertEquals("111.222.333-44", cpf.formatar());
}

@Test(expected = CPFInvalidoException.class)
public final void testCPFInvalido() throws Exception {
    CPF cpf = new CPF("entradaInvalida");
    cpf.formatar();
}
```

Objetos Falsos (Mock Objects)

- Uma unidade pode depender de outra que colabora com seu funcionamento
- Objetos Falsos simulam o comportamento de objetos mais complicados
- Vantagens:
  - Testes mais simples
  - Independentes
  - Eficientes

Exemplo de Objetos Falsos

```
@Test
public final void testMeuMetodo() throws Exception {
    // Crio um objeto Mock
    ClasseColaboradora classeColaboradora = createMock(ClasseColaboradora.class);
    // Digo a ele como eu quero que ele se comporte
    expect(classeColaboradora.metodoColaborador()).andReturn("");
    // Gravo o comportamento do Mock
    replay(classeColaboradora);
    // Seto o Objeto Mock como a classe colaboradora
    MinhaClasse minhaClasse = new MinhaClasse(classeColaboradora);
    // Executo o Método
    Object obj = minhaClasse.meuMetodo();
    // Verifico se o Mock se comportou como o esperado
    verify(classeColaboradora);
    // Faço as verificações
    assertEquals("Uma saida esperada", obj);
}
```

Exemplo - Servlet

```
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

    String rua = (String) req.getParameter("rua");
    String numero = (String) req.getParameter("numero");
    String complemento = (String) req.getParameter("complemento");
    String bairro = (String) req.getParameter("bairro");
    String cep = (String) req.getParameter("cep");
    if (ValidacaoUtil.isVazio(rua) || ValidacaoUtil.isVazio(numero)
            || ValidacaoUtil.isVazio(bairro)) {
        resp.sendError(404, "Esse endereço não foi encontrado!");
        return;
    }
    Endereco end = new Endereco(rua, numero, complemento, bairro, cep);
    req.getSession().setAttribute("endereco", end);
    resp.sendRedirect("mostraEndereco.jsp");
}
```

Exemplo – Objetos Falsos

```
@Test
public void testeInsereEnderecoCorretoNaSessao() throws Exception {
    expect(req.getParameter("rua")).andReturn("Av. Paulista");
    expect(req.getParameter("numero")).andReturn("1");
    expect(req.getParameter("bairro")).andReturn("Jardins");
    expect(req.getParameter("complemento")).andReturn("2 andar");
    expect(req.getParameter("cep")).andReturn("01111-111");

    expect(req.getSession()).andReturn(session);
    session.setAttribute(eq("endereco"), isA(Endereco.class));

    resp.sendRedirect(eq("mostraEndereco.jsp"));

    replay(mocks);
    servlet.doPost(req, resp);
    verify(mocks);
}
```

Teste de Integração

- Testa a integração entre unidades ou componentes
- Duas unidades podem funcionar perfeitamente individualmente,
- mas podem haver falhas quando elas se comunicam

Testes de Aceitação

- Testa as funcionalidades do sistema a partir do ponto de vista do cliente
- Teste caixa preta: Não é necessário conhecimento da implementação do módulo
- É um teste de alto nível
- É uma 'prova' para o cliente de que o software funciona
- O cliente pode especificar os testes e os desenvolvedores os implementam

Exemplo - História

- O sistema deve mostrar uma lista de ingredientes que poderão fazer parte da pizza, separando-os por categorias. Inicialmente, os ingredientes deverão ser os seguintes:

  - **Queijos**
  - Muzzarela
  - Muzzarela em dobro
  - ...
  - **Vegetais**
  - Escarola
  - Rúcula
  - ...
  - **Carnes**
  - Calabresa
  - Frango
  - ...
  - **Outros**
  - Ovo
  - Orégano
  - ...

Exemplo - Teste

```java
public class ListaDeIngredientesTest extends SeleniumTestCase {

    @Before
    public void entrarNaPagina() {
        sel.open("/AgilPizza/ingredientes.jsp");
        sel.clickAndWait("listaDeIngredientes");
    }

    @Test
    public void testCategorias() {
        assertTrue(sel.isTextPresent("Carnes"));
        assertTrue(sel.isTextPresent("Queijos"));
        assertTrue(sel.isTextPresent("Vegetais"));
        assertTrue(sel.isTextPresent("Outros"));
    }

    @Test
    public void testIngredientes() {
        assertTrue(sel.isTextPresent("Calabresa"));
        assertTrue(sel.isTextPresent("Mussarela"));
        assertTrue(sel.isTextPresent("Rúcula"));
        assertTrue(sel.isTextPresent("Orégano"));
    }

}
```

Testes de Interface Gráfica do Usuário (GUI)

- Módulos distintos e isolados podem alterar o estado da interface
  - Estado pode ficar inconsistente
  - Sujeira na tela
- Ajuda a desenhar a interface (Test First)
- Quando podemos ver a simulação do teste:
  - Podemos usar o teste como um tutorial do sistema
  - Podemos usar para apresentar o sistema

Testes de Desempenho

- Otimizar somente quando necessário
- Código claro traz mais benefícios do que milésimos de segundo de otimização
- Começar pelos gargalos da aplicação
- Utilizar *Profilers* para detectar os gargalos

Testes de Estresse

- Testar com grandes quantidades de dados gerados automaticamente
- Simulação de muitos usuários simultâneos
- Erros comuns:
  - Overflow
  - Falta de memória
- Alguns benefícios:
  - Encontrar vazamento de memória
  - Avaliar a capacidade de um ambiente

Testes de Segurança

- Útil principalmente para aplicações Web
  - Aplicações expostas a usuários mal-intencionados
- Testar valores inesperados:
  - Null
  - Grande quantidade de dados
  - Tipos de dados não esperados
    ex: colocar uma string em um campo que espera um inteiro
- Testes de estresse para lidar com ataques de negação de serviço (DoS)

Análise de Risco

- Ajuda a planejar quais testes devem ser feitos
- Risco: Ameaça ao sucesso de um projeto
- Riscos do gerenciamento do projeto
  - Testes não ajudam muito
- Riscos do negócio
  - Testes da funcionalidade
- Riscos técnicos
  - Testes de unidades e integração

Análise de Risco

- Quando um teste de aceitação falha, pode ser um sinal que testes de unidade estão faltando
- Quando um usuário ou cliente encontra um erro, é um sinal que estão faltando testes de unidade e de aceitação
- Lembrando: Antes de corrigir o erro, escreva um teste que o reproduza

Algumas Ferramentas

- Testes de Unidade:
  - Família XUnit (e.g.: JUnit)
  - TestNG
  - EasyMock
- Testes de Aceitação:
  - JWebUnit
  - Selenium
  - Fit
- Testes de Desempenho / Estresse:
  - JMeter

Considerações Finais

- Testes automatizados é uma prática fundamental para garantir a qualidade do código, independente da metodologia

- “Qualquer funcionalidade que não possui testes automatizados simplesmente não existe” (Kent Beck)

- Então, Teste!!!!!!

Mais Informações

- http://www.testing.com/
- http://www.junit.org/
- http://www.easymock.org/
- http://www.opensourcetesting.org/
- http://www.openqa.org/selenium/
- http://fit.c2.com/
- Robert V. Binder, “Testing Object-Oriented Systems”, Addison-Wesley Professional, 1999

Perguntas

- Comentários?
- Dúvidas?
- Críticas?
- Sugestões?

agilcoop@gmail.com