USP - Universidade de São Paulo

IME - Instituto de Matemática e Estatística

MAC0332

Engenharia de Software

Testes

Marco Aurélio Gerosa

gerosa@ime.usp.br

Conceitos Básicos

- O que é qualidade de software?
- Como garantir a qualidade de software?
- Validação x verificação (V&V)
  - Validação – Atende aos requisitos do usuário? Estamos construindo o produto certo?
  - Verificação – Atende às especificações? Estamos construindo certo o produto?
- Prevenção de defeitos é melhor do que remoção de defeitos (Mays, 1990)
- “Inspecionar para prevenir defeitos é bom; Inspecionar para encontrar defeitos é desperdício” - Shigeo Shingo
- Um desenvolvedor não é indicado para testar seu próprio código (Weinberg, 1971)
- Aproximadamente 80% dos feitos são provenientes de 20% dos módulos (Pareto, 1897)(Endres, 1975)

Manutenção (evolução)

- Entendimento clássico: (IEEE 1990)
  - Alterações depois da entrega e instalação
  - Se a falha ou requisito é descoberto durante o desenvolvimento é tratado como parte do desenvolvimento, caso contrário é manutenção
- Entendimento moderno: (ISO/IEC 1995 e IEEE 1998)
  - O processo que ocorre quando um artefato de software é modificado por conta de um problema ou por necessidade de melhoria ou adaptação
  - (Ocorre sempre o software é modificado, independente da fase)

- Software é um modelo da realidade, que se altera constantemente

- 50% da manutenção pós-entrega é para correção (Schach et al., 2002 e 2003)

Testes

- Testes é parte das atividades de V&V
- Engano => Imperfeição no código => Resulta em um falha => Manifesta um erro [IEEE 610.12, 1990].
  - Defeito é uma palavra genérica para imperfeição, falha ou erro.
- Verificação no final de cada fase (pode ser tarde demais)
- Validação no final do projeto (mais tarde ainda)
- Atividades contínuas de testes devem ser feitas durante o projeto
  - Teste de unidade
  - Teste de integração
  - Teste de aceitação
- Grupo de qualidade
- Grupo de teste

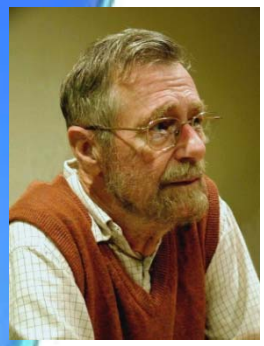
Testes de software

- Prejuízos de aproximadamente $59.5 bilhões na economia dos EUA (Fonte: NIST/2002)
- Tipos: caixa branca, preta ou cinza
- Fases: de unidade, de integração e de sistema
- Tipos: aceitação, performance, stress etc.
- Testes automatizados

Falhas

- Entre 60 a 70% das falhas são nos requisitos, análise ou projeto (Boehm, 1970).
- Exemplo: Jet Propulsion Laboratory inspections (1992)
  - 1.9 falhas por página de especificação
  - 0.9 por página de projeto
  - 0.3 por página de código
- Corrigir uma falha em fases mais avançadas do desenvolvimento demanda:
  - Alterar código e documentação
  - Executar teste de regressão
  - Reinstalar o produto nos clientes

Citações



"Program testing can be used to show the presence of bugs, but never to show their absence."

Edsger W. Dijkstra

“Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead.” Martin Fowler.

“Any program feature without an automated test simply does not exist.”

Kent Beck.

Testes de unidade

- Não é “testes unitários”
- Testes automatizados
  - Possibilita rodar uma bateria grande de testes sem depender da intervenção ou interpretação humana dos resultados
  - Rápido de executar – podem ser executados a cada alteração do sistema
  - Os testes podem cobrir todo código produzido
  - Oferece documentação sobre a funcionalidade do sistema
  - Mais segurança na manutenção

- JUnit - http://www.junit.org
  - Kent Beck, Erich Gamma
  - Open Source

Exemplo de teste

```
import br.usp.ime.mac0332.Calculadora;

public class TesteCalculadora {

    private Calculadora calculadora;

    @Before
    public void setUp() throws Exception {
        calculadora = new Calculadora();
    }

    @After
    public void tearDown() throws Exception {
        calculadora = null;
    }

    @Test
    public void somaPositivos() {
        assertEquals(42, calculadora.soma(11, 31));
    }

    @Test
    public void somaNegativos() {
        assertEquals(-42, calculadora.soma(-11, -31));
    }

    @Test
    public void divide() {
        assertEquals(42, calculadora.divide(84, 2));
    }

    @Test(expected = ArithmeticException.class)
    public void dividePorZero() {
        calculadora.divide(84, 0);
    }
}
```

Asserts

- assertEquals
- assertTrue / assertFalse
- assertSame / assertNotSame
- assertNull / assertNotNull
- assertArrayEquals

JUnit 3 x Junit 4

JUnit 3

```
import junit.framework.*;

public class MultiplicationTest extends TestCase {
   public void testMultiplication() {
     assertEquals("Multiplication", 6, 3 * 2);
   }
}
```

JUnit 4

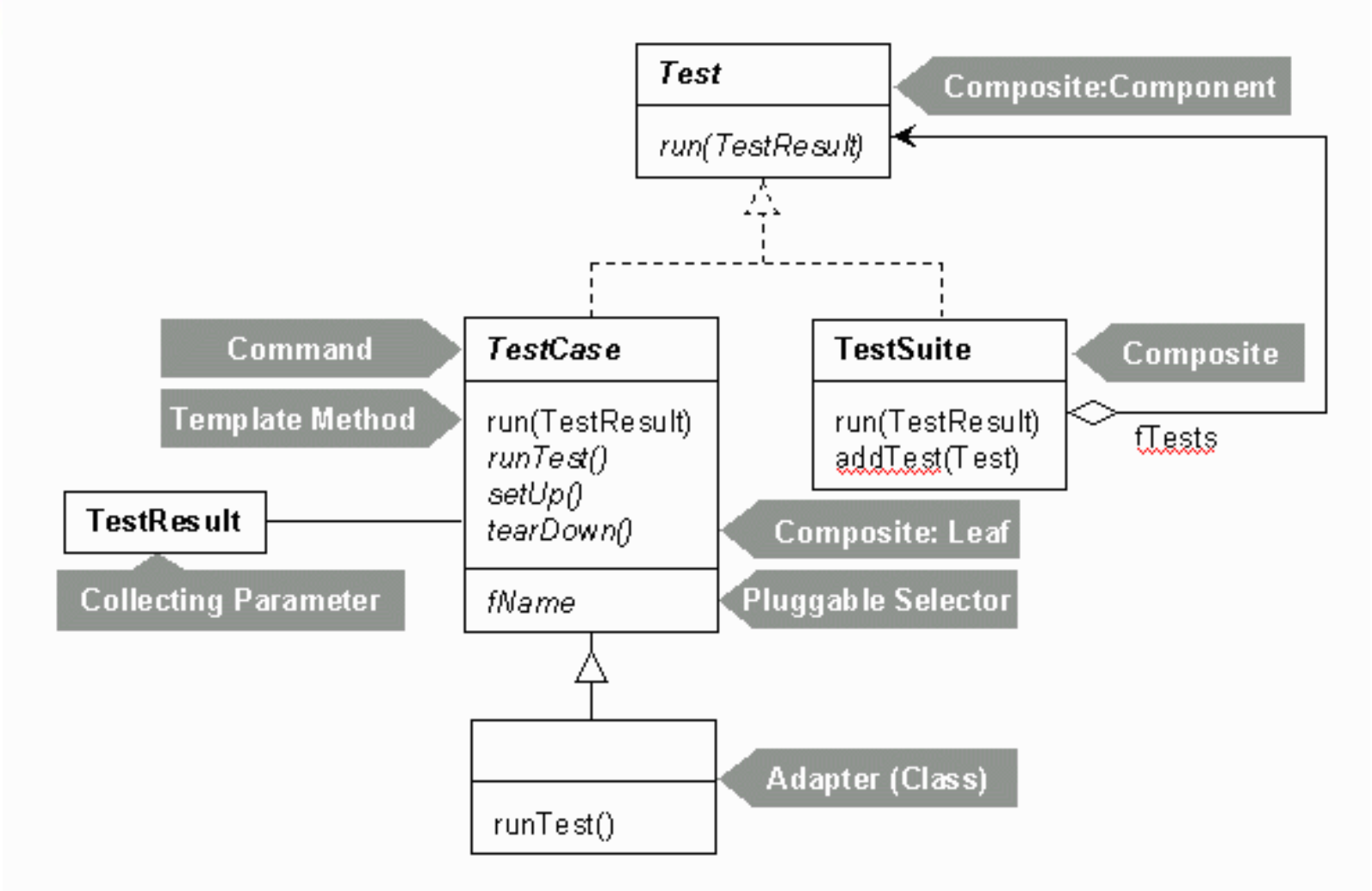
```
import org.junit.*;

public class MultiplicationTest {
   @Test
   public void multiplication() {
     Assert.assertEquals("Multiplication", 6, 3 * 2);
   }
}
```

JUnit 3 x JUnit 4

JUnit 3	JUnit 4
public void testX()	@Test
public void setUp()	@Before
public void tearDown()	@After
assertEquals()	Assert.assertEquals()

Arquitetura JUnit



Princípios para escrever testes

- Código dos testes deve ser simples
- Testes podem conter erros
- Não devem exigir intervenção humana
- Devem documentar o sistema

Mock Objects

- Simulam objetos reais
- Úteis para isolar o teste de um objeto do outro ou quando temos objetos que são difíceis de criar, reproduzir, lerdos, que ainda não existem etc.
- Você configura a expectativa do comportamento do objeto
- Frameworks
  - JMock
  - EasyMock
  - Mockito

JUnit 3 x Junit 4

JUnit 3

```
import junit.framework.*;

public class MultiplicationTest extends TestCase {
   public void testMultiplication() {
     assertEquals("Multiplication", 6, 3 * 2);
   }
}
```

JUnit 4

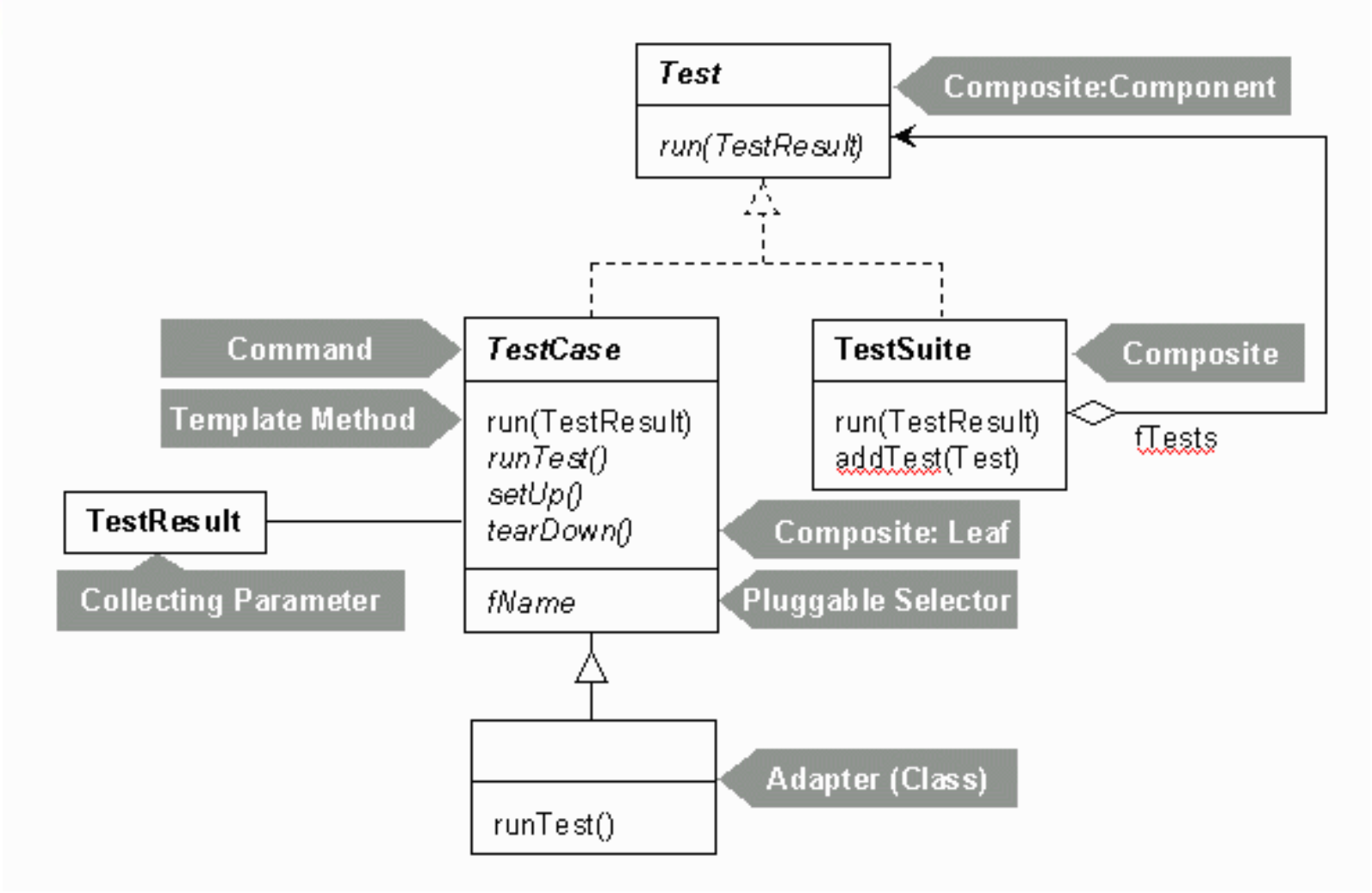
```
import org.junit.*;

public class MultiplicationTest {
   @Test
   public void multiplication() {
     Assert.assertEquals("Multiplication", 6, 3 * 2);
   }
}
```

JUnit 3 x JUnit 4

JUnit 3	JUnit 4
public void testX()	@Test
public void setUp()	@Before
public void tearDown()	@After
assertEquals()	Assert.assertEquals()

Arquitetura JUnit



Sobre a escrita de nomes

What's in a name

- Don't use the word 'test' in your test names

http://www.slideshare.net/wakaleo/junit-kung-fu-getting-more-out-of-your-unit-tests

What's in a name

- *Do* use the word 'should' in your test names

```
testBankTransfer()

testWithdraw()
```

```
tranferShouldDeductSumFromSourceAccountBalance()

transferShouldAddSumLessFeesToDestinationAccountBalance()

depositShouldAddAmountToAccountBalance()
```

What's in a name

- Your test class names should represent context

When is this behaviour applicable?

```
WhenYouCreateACell
    aDeadCellShouldBePrintedAsADot() : void
    aDeadCellShouldBeRepresentedByADot() : void
    aDeadCellSymbolShouldBeADot() : void
    aLiveCellShouldBePrintedAsAnAsterisk() : void
    aLiveCellShouldBeRepresentedByAnAsterisk() : void
    aLiveCellSymbolShouldBeAnAsterisk() : void
```

What behaviour are we testing?

What's in a name

- Write your tests consistently
  - 'Given-When-Then' or 'Arrange-Act-Assert' (AAA)

```
@Test
public void aDeadCellWithOneLiveNeighbourShouldRemainDeadInTheNextGeneration() {
    String initialGrid = "...\n" +
                         ".*.\n" +
                         "...";

    String expectedNextGrid = "...\n" +
                              "...\n" +
                              "...\n";

    Universe theUniverse = new Universe(seededWith(initialGrid));

    theUniverse.createNextGeneration();
    String nextGrid = theUniverse.getGrid();

    assertThat(nextGrid, is(expectedNextGrid));
}
```

Prepare the test data ("arrange")

Do what you are testing ("act")

Check the results ("assert")

What's in a name

- Tests are deliverables too - respect them as such
  - Refactor, refactor, refactor!
  - Clean and readable

TDD

- Sugerido por Kent Beck [Bec04], a prática sugere ao desenvolvedor que escreva o código de teste antes do código de produção.
- Ciclo em 5 etapas: escreva um teste que falha, veja-o falhar, faça o teste passar da maneira mais simples possível, veja-o passar, refatora para remover duplicação de dados e código.

Efeitos da prática

- É comum relacionar a prática de TDD com efeitos positivos na qualidade externa do software.
  - Muitos trabalhos nem mencionam possíveis efeitos da prática no projeto de classes.
- Mas muitos autores também afirmam que a prática pode influenciar positivamente no projeto de classes [Bec02], [Mar6], [eNP09], [Ast03].
  - Grande parte delas se baseia na ideia de que uma classe difícil de ser testada, é uma classe que possivelmente apresenta problemas de projeto.

Feedback mais rápido

- Praticantes afirmaram que TDD dá feedback constante ao desenvolvedor.