UFPB CCAE Centro de Ciências Aplicadas e Educação

CAMPUS IV - LITORAL NORTE

Introdução ao Teste de Software com JUnit e EMMA

Wilkerson de Lucena Andrade

wilkerson.andrade@gmail.com

Sumário

- Introdução ao Teste de Software
- Teste Funcional
- JUnit
- Teste Estrutural
- EMMA

UFPB CCAE Centro de Ciências Aplicadas e Educação

CAMPUS IV - LITORAL NORTE

Introdução ao Teste de Software

Wilkerson de Lucena Andrade

wilkerson.andrade@gmail.com

Introdução

- Por que testar?
  - Avaliar a qualidade ou aceitabilidade
  - Descobrir problemas
- Objetivos:
  - Mostrar que a aplicação faz o esperado
  - Mostrar que a aplicação não faz mais do que o esperado

O que é Teste de Software e o que Não é?

- Processo para descobrir a existência de defeitos em um software
- Um defeito pode ser introduzido em qualquer fase do desenvolvimento ou manutenção como resultado de:
  - Imprecisão
  - Desentendimentos
  - Omissões
  - Direcionamento a soluções particulares
  - Inconsistências
  - Não completude

O que é Teste de Software e o que Não é?

- **Teste é um processo referencial**
  - É necessário existir uma definição precisa do que se quer verificar e quais os resultados esperados
- **Teste não é *debugging***
  - *Debugging* é o processo de encontrar/localizar defeitos

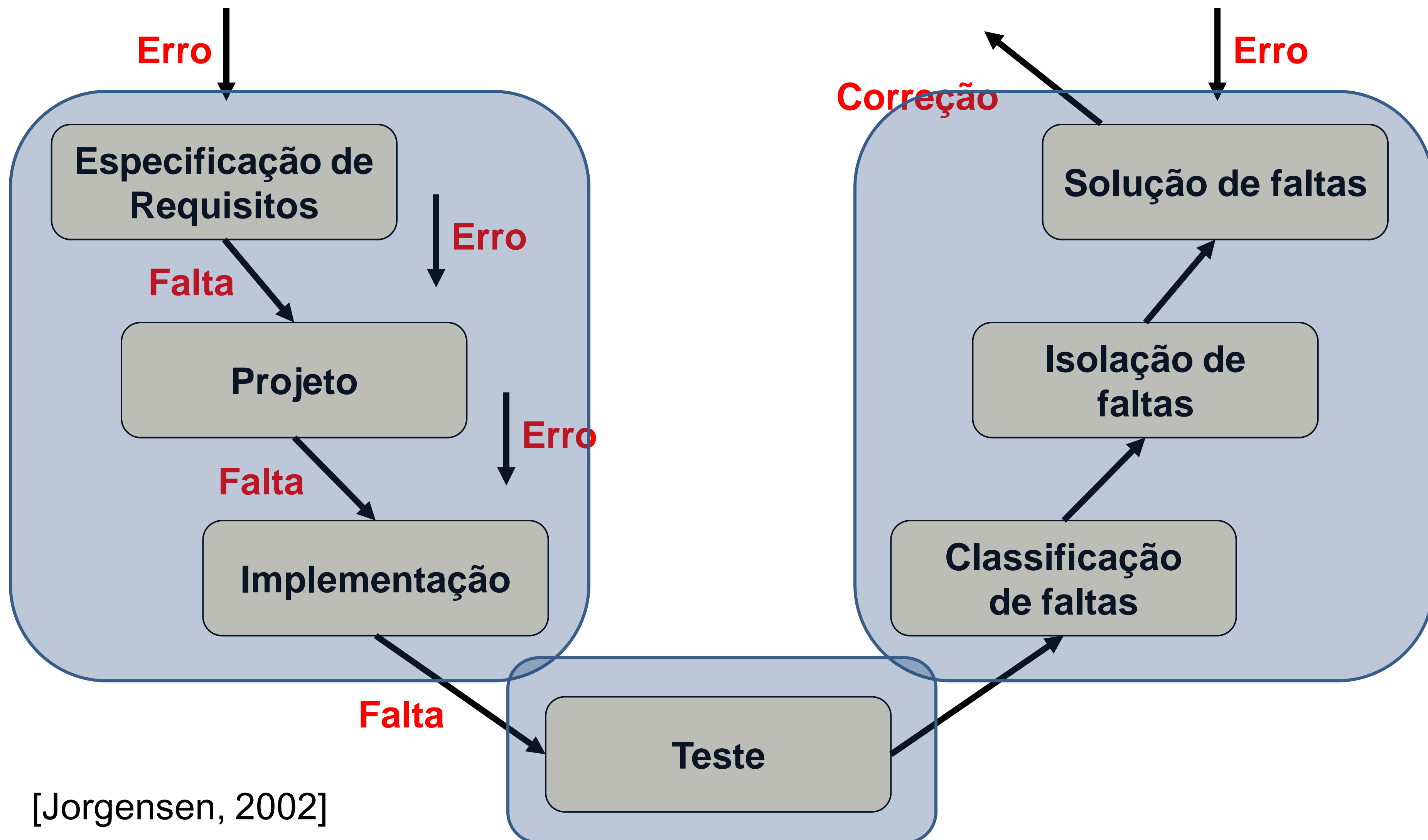
Terminologia

- Erro
  - Engano ou omissão causado por uma ação humana
  - Ocorre durante a codificação
  - Tende a ser propagado
- Falta
  - Representação de um erro
  - Sinônimo de defeito ou *bug*
  - Falta de comissão - representação incorreta
  - Falta de omissão – representação ausente

Terminologia

- Falha
  - Impossibilidade de um sistema ou componente de executar uma função requisitada
  - Ocorre quando um software com falta é executado
  - É evidenciada através de saída incorreta, término anormal, não satisfação de restrições de tempo e espaço

Modelo do Ciclo de Vida do Teste



[Jorgensen, 2002]

Caso de Teste

- Comportamento a ser testado, normalmente definido em termos de estímulos de entrada e respostas esperadas
- Especifica o que se quer testar:
  - Pré-estado da implementação e seu ambiente
  - Condições
  - Entradas de teste
  - Resultados esperados

Caso de Teste

- Resultados esperados incluem:
  - Mensagens geradas pela implementação
  - Exceções
  - Valores retornados
  - Estado esperado da implementação e seu ambiente

Caso de Teste

ID do Caso de Teste

Propósito

Pré-condições

Entradas

Saídas esperadas

Pós-condições

Histórico de execução

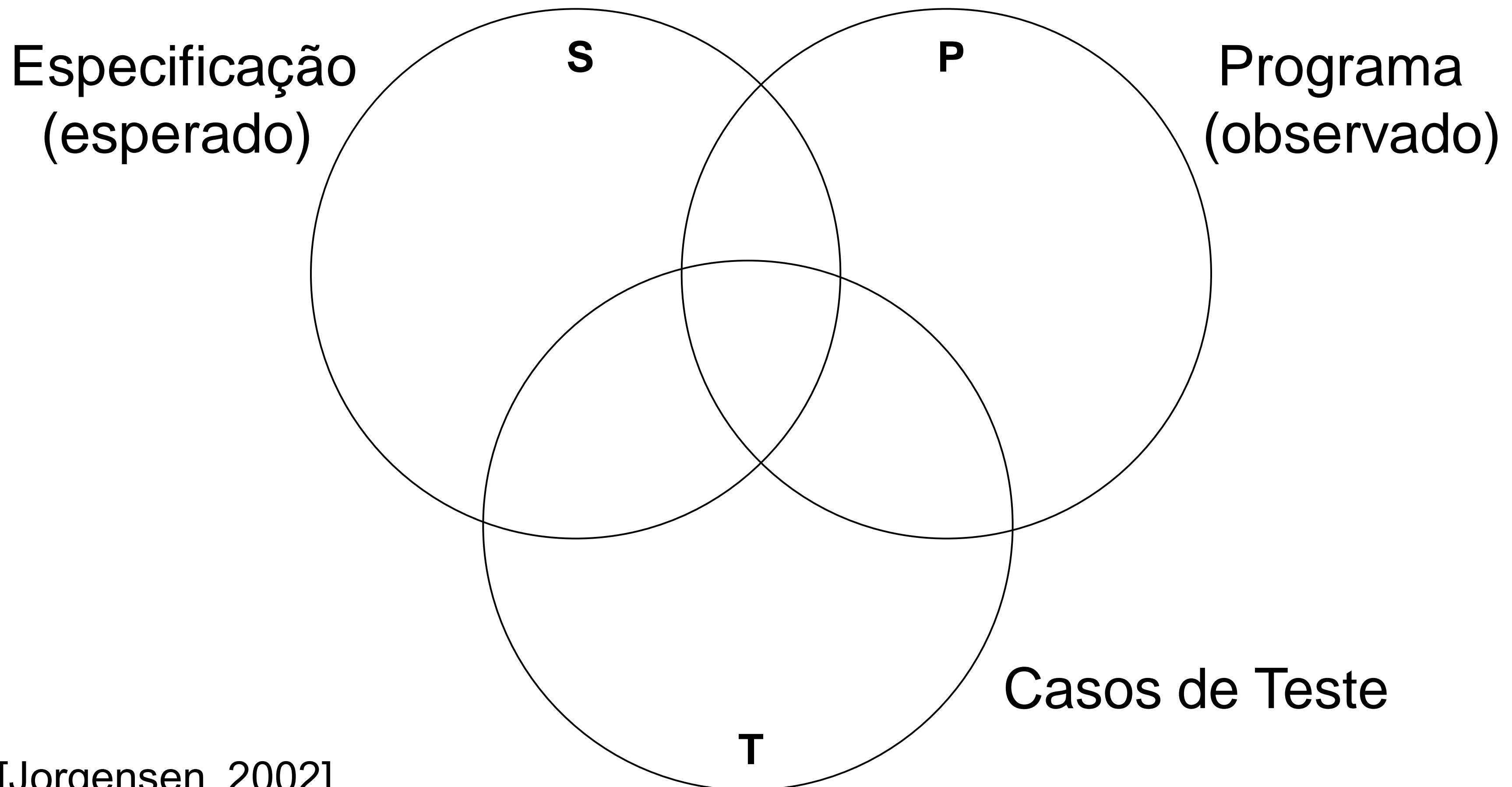
Data Resultado Versão Executado por

[Jorgensen, 2002]

Teste X Comportamento

- Teste é fundamentalmente associado a comportamento
- A visão comportamental é ortogonal em relação a visão estrutural
- Desenvolvedores possuem uma visão estrutural
- Os documentos básicos são escritos por desenvolvedores
- Testadores geram os testes a partir dos documentos gerados

Teste X Comportamento



Intercalando Teste e Desenvolvimento

- Desenvolvimento Iterativo e Incremental:
  - Analise um pouco
  - Projete um pouco
  - Codifique um pouco
  - Teste o que puder
- Filosofia:
  - Teste o quanto antes
  - Teste sempre
  - Teste o suficiente

Intercalando Teste e Desenvolvimento

Análise → Planejamento e Especificação de Testes
Projeto → Refinamento e Projeto de Testes
Implementação → Implementação de Testes

- Casos de teste podem ser identificados mais cedo durante a fase de requisitos
- Analistas e projetistas podem expressar e entender melhor requisitos e assegurar que são testáveis

Intercalando Teste e Desenvolvimento

Análise → Planejamento e Especificação de Testes
Projeto → Refinamento e Projeto de Testes
Implementação → Implementação de Testes

- Defeitos podem ser detectados mais cedo
  - São mais fáceis e baratos de consertar
- Casos de teste podem ser revisados
  - Desentendimentos podem levar a aceitar programas incorretos e rejeitar programas corretos

Testabilidade

- Propriedade que indica a facilidade e precisão na avaliação dos resultados de um teste
- Um produto é testável se oferece suporte a:
  - Geração de testes
  - Implementação
  - Verificação de seus resultados de forma precisa

Testabilidade

- Requisitos constituem a fonte básica para a geração de testes de sistema e de aceitação
- Testadores devem garantir que os documentos gerados propiciam um nível suficiente de entendimento para a geração de testes e que sejam corretos, completos, consistentes e não-ambíguos

Dimensões de Teste

- Quem executa os testes?
  - Equipe dedicada e/ou desenvolvedores
- Que partes serão testadas e que tipos de testes serão considerados?
  - Unidades, componentes, sistemas – todos ou seletivos?
  - Funcionalidade, interface, desempenho, usabilidade, robustez, etc.

Dimensões de Teste

- Quando o teste será executado?
  - Escalonamento dentro do processo de desenvolvimento
- Como o teste será executado?
  - Visão Funcional X Visão Estrutural
- Qual a quantidade adequada de casos de teste?
  - Critérios de Aceitação e de Cobertura

Requisitos para um Bom Testador

- Ter um bom entendimento do processo de desenvolvimento, de tecnologias empregadas e do produto sendo gerado, além da habilidade de indicar possíveis falhas e erros
- Ter uma atitude de questionar todos os aspectos relacionados com o software:
  - **Cético** – Quer prova de qualidade
  - **Objetivo** – Não se baseia em suposições
  - **Cuidadoso** – Não deixa passar detalhes importantes
  - **Sistemático** – Buscas são reproduzíveis

Vantagens de Teste

- Se conduzidos de forma rigorosa:
  - Contribuem para aumentar a confiabilidade do software
  - Evidenciar características de qualidade
  - Verificar o software no ambiente operacional

Limitações de Teste

- Número de possíveis combinações é muito grande ou infinito:
  - Espaço de Estado/Entrada
  - Sequências de Execução (branching e dynamic binding)
  - **Sensibilidade a Falta** – habilidade do código esconder faltas
  - **Correção coincidental** – um código correto pode produzir resultados corretos para algumas entradas

Limitações de Teste

- Prova de Correção = Teste Exaustivo
- Certos aspectos podem ser impossíveis de implementar
  - Situações em que o sistema não pode produzir uma resposta – indecibilidade
- Deve ter um ponto de referência (especificação):
  - Não pode verificar requisitos diretamente
  - Testes com pouco valor podem ser produzidos se requisitos estão incorretos ou incompletos
- Não podemos garantir que uma aplicação esteja correta

UFPB CCAE Centro de Ciências Aplicadas e Educação

CAMPUS IV – LITORAL NORTE

Teste Funcional

Wilkerson de Lucena Andrade

wilkerson.andrade@gmail.com

Teste Funcional

- Também conhecido como Teste *Black-Box*
- Parte do pressuposto que qualquer programa pode ser visto como uma função que mapeia valores do domínio de entrada em valores do contradomínio
- O principal objetivo é verificar se uma dada implementação está de acordo com a sua respectiva especificação
- A única informação usada é a especificação do software

Teste Funcional

- Vantagens
  - Os casos de teste são independentes de implementação
  - O desenvolvimento dos casos de teste podem ocorrer paralelamente com o desenvolvimento do software
- Desvantagens
  - Dificuldade em quantificar a atividade de teste
    - Não se pode garantir que partes essenciais ou críticas do software foram executadas

Teste Funcional

- As principais técnicas de teste funcional:
  - Testes derivados de especificação
  - Partição por Equivalência
  - Análise de Valores Limites
  - Teste Baseado em Estado-Transição

Teste Funcional

- Testes Derivados de Especificação
  - Baseado na especificação, os testes são gerados de acordo com as várias expressões contidas na mesma
  - Não se pode precisar que as expressões contidas na mesma refletirão as expressões contidas no código, mas isso tende a ser uma prática comum em vários trechos

Testes Derivados de Especificação

- Exemplo – Função para cálculo de Raiz Quadrada

Input Número Real

Output Número Real

Para uma dada entrada maior ou igual a 0, a raiz positiva do número será retornada. Para uma dada entrada menor que 0, a mensagem “Erro – Entrada inválida” deverá ser mostrada e o valor 0 deverá ser retornado. A rotina “PrintLine” deverá ser usada para mostrar a mensagem.

Testes Derivados de Especificação

- Exemplo – Função para cálculo de Raiz Quadrada
  - Caso 1: Entrada 4.0, Retorno 2.0
    - Testa a primeira expressão da especificação
  - Caso 2: Entrada -10.0, Retorno 0.0, Saída “Erro – Entrada inválida” através de PrintLine
    - Testa a segunda e terceira expressão da especificação

Teste Funcional

- Partição por Equivalência
  - Significa identificar partições dos domínios das entradas e saídas onde os elementos, supostamente, fariam com que o sistema se comportasse da mesma forma
  - Partições são identificadas não somente em parâmetros de funções, métodos, etc., mas também em dados acessados, tempo, seqüência de entradas e saídas bem como em estados

Partição por Equivalência

- Exemplo – Função para cálculo de Raiz Quadrada

Partições de Entrada		Partições de Saída	
i	< 0	a	>= 0
ii	>= 0	b	Error

Partição por Equivalência

- Exemplo – Função para cálculo de Raiz Quadrada
  - Caso 1: Entrada 4.0, Retorno 2.0
    - Testa ii e a
  - Caso 2: Entrada -10.0, Retorno 0.0, Saída “Erro – Entrada inválida” através de PrintLine
    - Testa i e b

Teste Funcional

- Análise de Valores Limites
  - Esta técnica se baseia na hipótese de que erros geralmente são encontrados nas regiões limites das partições
    - Ex.: uma função que trabalha no domínio dos inteiros possui como um valor limite o 0
  - Neste ponto estamos interessados em procurar por erros

Análise de Valores Limites

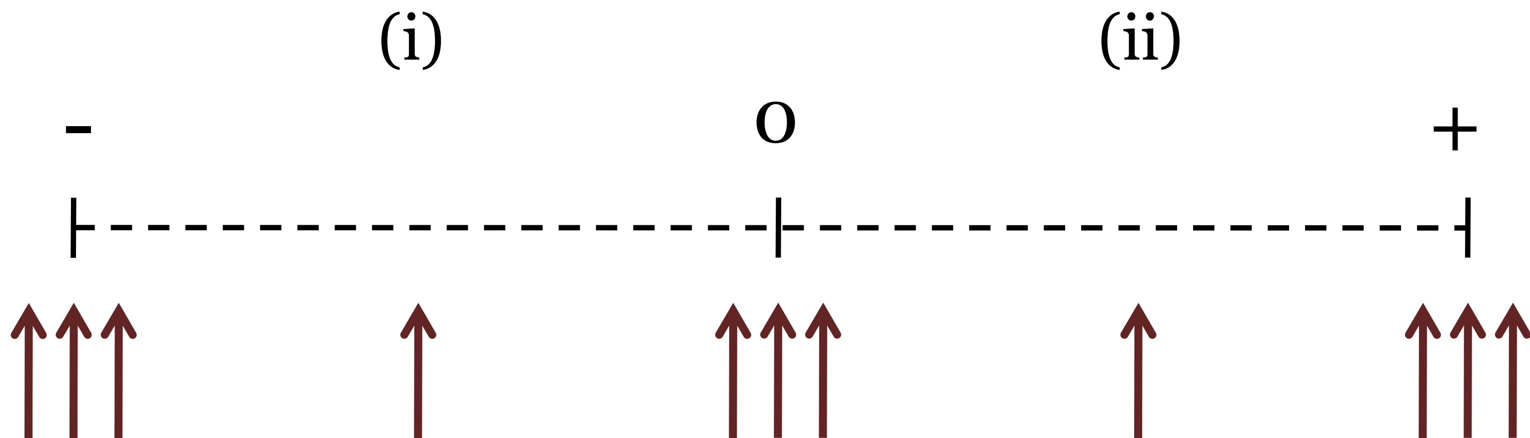
- Análise de Valores Limites foca nos limites do espaço de entrada para identificar casos de teste
- Erros tendem a ocorrer próximo aos valores extremos das variáveis de entrada
- Exemplo
  - Condições de *loop*
    - Testar < quando deveria ser <=

Análise de Valores Limites

- Valores das variáveis utilizados no teste:
  - O Valor mínimo
  - O valor mínimo + 1
  - Um valor qualquer
  - O valor máximo - 1
  - O valor máximo
- Um teste mais robusto deve considerar:
  - O Valor mínimo - 1
  - O Valor máximo + 1

Análise de Valores Limites

- Exemplo – Função para cálculo de Raiz Quadrada



Análise de Valores Limites

- Exemplo – O problema do Triângulo
  - Há entrada três inteiros a, b e c:
  - a, b e c são os lados do triângulo e devem satisfazer as seguintes condições:
    - 1 <= a <= 200
    - 1 <= b <= 200
    - 1 <= c <= 200
    - a < b + c
    - b < a + c
    - c < a + b

Análise de Valores Limites

- Exemplo – O problema do Triângulo
  - A saída do programa é o tipo do triângulo determinado pelos lados:
    - Equilateral
    - Isosceles
    - Scalene
    - NotATriangle
  - Se alguma entrada falha em alguma das condições citadas, o programa deve mostrar uma mensagem de erro

Casos de Teste - Problema do Triângulo

Análise de Valores Limites

Teste	a	b	c	Saída Desejada
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a Triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	100	100	Equilateral
9	100	199	100	Isosceles
10	100	200	100	Not a Triangle

Casos de Teste - Problema do Triângulo

Análise de Valores Limites

Teste	a	b	c	Saída Desejada
11	1	100	100	Isosceles
12	2	100	100	Isosceles
13	100	100	100	Equilateral
14	199	100	100	Isosceles
15	200	100	100	Not a Triangle

Dados em Teste Funcional

- Como dito anteriormente os dados são um fator crucial em testes funcionais
- Com dados ruins, os testes podem não produzir os resultados esperados
  - Os dados podem não refletir o contexto real efetivamente
- Bons dados permitem maior fidelidades dos testes
  - Testes precisam ser repetidos com o mesmo resultado ou variações precisam ser diagnosticadas

Considerações

- Teste funcional se preocupa com a “satisfação de contratos” (especificações)
  - Dependendo do tipo e do nível de teste
- Testes funcionais são executados nas fases de desenvolvimento, principalmente, e de *deployment* (aceitação)
- É possível automatizar a geração tanto dos testes quanto dos casos de teste, desde que se tenha especificações formalizadas
- O ferramental existente para automatização de testes funcionais é bastante abrangente

UFPB CCAE Centro de Ciências Aplicadas e Educação

CAMPUS IV – LITORAL NORTE

JUnit

Wilkerson de Lucena Andrade

wilkerson.andrade@gmail.com

Introdução

- JUnit é um framework open source usado para o desenvolvimento e execução de testes escritos em Java
- Desenvolvido por Eric Gamma e Kent Beck
- A maioria das IDEs incorporam o JUnit dentro de seu ambiente de desenvolvimento
  - JBuilder
  - JDeveloper
  - Netbeans
  - Eclipse

Como usar o JUnit?

- Depende da metodologia de testes que está sendo usada:
  - Código existente
  - Desenvolvimento guiado por testes (TDD)
- Onde obter o JUnit?
  - **www.junit.org**
- Como instalar?
  - Incluir o arquivo **junit.jar** no classpath

Testando código com JUnit

- Crie uma classe de teste para cada classe a ser testada

```
Public class MyClassTest {

...

}
```

Testando código com JUnit

- Para cada método `xxx(args)` a ser testado defina um método `@Test public void xxx()`:
  - MyClass:

```
public String setObject(Object o) {
...
}
```

  - MyClassTest:

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

O que colocar em um teste?

- Cada método criado na sua classe de teste pode ser um caso de teste
  - Escreva o código para verificar o correto funcionamento da unidade de código dentro deste método
- Use asserções do JUnit para verificar os resultados do teste e para causar falhas se o resultado não for o esperado

O que colocar em um teste?

Método	Descrição
assertTrue	Verifica se uma condição é verdade
assertFalse	Verifica se uma condição é falsa
assertEquals	Verifica se dois objetos são iguais
assertNotNull	Verifica se um objeto não é null
assertNull	Verifica se um objeto é null
assertSame	Verifica se dois objetos apontam para um mesmo objeto
assertNotSame	Verifica se dois objetos não apontam para um mesmo objeto
fail	Faz com que um teste falhe

Como executar um teste?

- Para executar digite:
  - java -classpath .;dir/junit-4.4.jar **org.junit.runner.JUnitCore** [classes de teste]

Como funciona?

- Para cada método de teste `public void xxx()`, a ferramenta executa:
  - O método anotado com `@Before`
  - O próprio método `xxx()`
  - O método anotado com `@After`
- Um teste pode **terminar**, **falhar** ou causar uma **exceção**

Anotações

Anotação	Descrição
@BeforeClass	Métodos invocados antes da execução da suíte de teste
@AfterClass	Métodos invocados após a execução da suíte de teste
@Before	Métodos que são executados antes de todos os testes
@After	Métodos que são executados depois de todos os testes
@Test	Métodos reais de teste
@Ignore	Testes que ainda não foram implementados podem ser desabilitados temporariamente

Fixture

- São os dados utilizados por vários testes

```
public class CollectionNamesTest {
  protected Collection<String> stringCollection;
  @Before public void setUp() throws Exception {
    stringCollection = new ArrayList<String>();
    stringCollection.add("Maria");
  }

  @Test public void testLength(){
    assertEquals(1, stringCollection.size());
  }

  @Test public void testToString(){
    assertEquals("[Maria]", stringCollection.toString());
  }
...
```

Teste de situações de falha

```
@Test(expected=ProductException.class)
public void testInvalidCode() {
    Product product = new Product(-2);
}

public void testInvalidCode() {
  try {
    Product product = new Product(-2);
    fail("Should have caused Exception!");
  } catch (Exception e) {
    assertNotNull(e.getMessage());
}
```

TestSuite

- Representa uma composição de testes
- Boa prática: crie uma classe **`AllTests`** em cada pacote de testes

```
@RunWith(Suite.class)
@SuiteClasses({CelsiusTemperatureTest.class,
                    FahrenheitTemperatureTest.class})
public class AllTests {
}
```

TestSuite

- Boa prática: crie uma classe para a execução de todos os testes da sua aplicação
  - Inclua nesta classe as suites de teste de cada pacote

```
@RunWith(Suite.class)
@SuiteClasses({tempconverter.app.AllTests.class,
                    tempconverter.scales.AllTests.class})
public class AllTests {
}
```

Teste Estrutural

Wilkerson de Lucena Andrade
wilkerson.andrade@gmail.com

Teste Estrutural

- O principal objetivo deste tipo de teste é testar detalhes procedimentais
- Os requisitos de teste são extraídos de uma implementação em particular
- Os critérios desta técnica utilizam grafo de fluxo de controle (grafo de programa)
- É também conhecido como teste *White-Box*

Teste Estrutural

- Vantagens
  - Testa partes do software que não estão na especificação
- Desvantagens
  - Não reconhece comportamentos que estão na especificação mas não foram implementados

Teste Estrutural

- As principais técnicas de teste estrutural:
  - Baseada em Fluxo de Controle
    - Teste de Comandos
    - Teste de Ramos
    - Teste de Condição
    - Teste de Condição Múltipla
  - Baseada em Fluxo de Dados
  - Baseada na Complexidade
    - Critério de MacCabe (Caminhos Base)

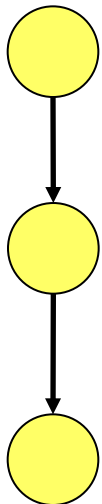
Teste Estrutural

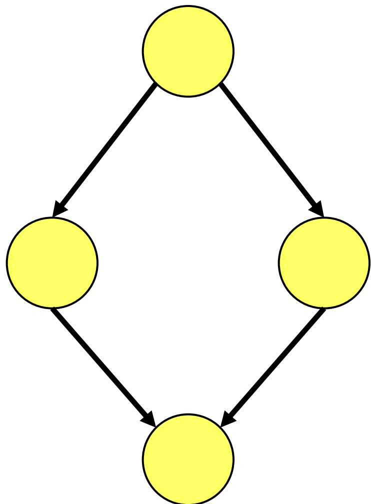
- Grafo de Fluxo de Controle
  - Consiste de um grafo direcionado
  - Os nós representam blocos de comandos
    - Um bloco de comando é um conjunto de expressões tal que se a primeira expressão for executada, todas as demais também o são
  - Os arcos representam precedência ou transferência de controle
  - A representação de fluxo de controle permite uma análise independente da função

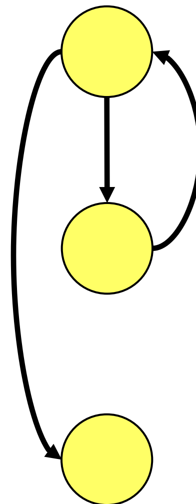
Teste Estrutural

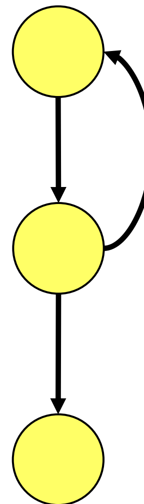
- Grafo de Programa
  - Representações básicas

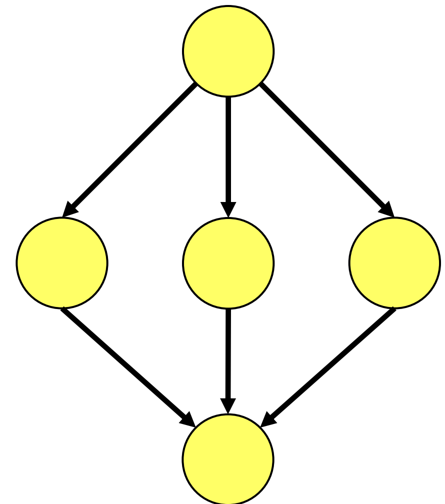
seqüência **if** **for/while** **do...while** **switch**











Teste Estrutural

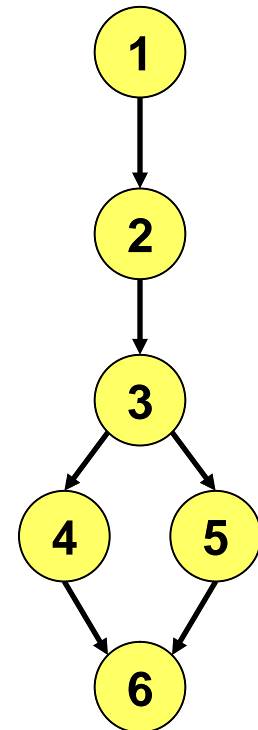
- Exemplo 1: programa com um caminho

```
... {
    double x = 10.0;    1
    double r = sqr(x);   2
    return r;        3
}
```

Teste Estrutural

- Exemplo 2: programa com dois caminhos

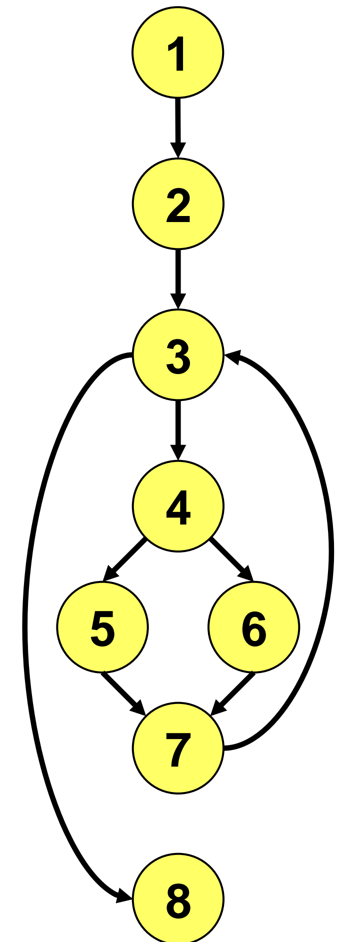
```
... {
    double x = readDouble(); ①
    String msg = ""; ②
    if(x >= 0){ ③
        msg = "sqr(x) = " + sqr(x); ④
    } else {
        msg = "Error"; ⑤
    }
    System.out.println(msg); ⑥
}
```



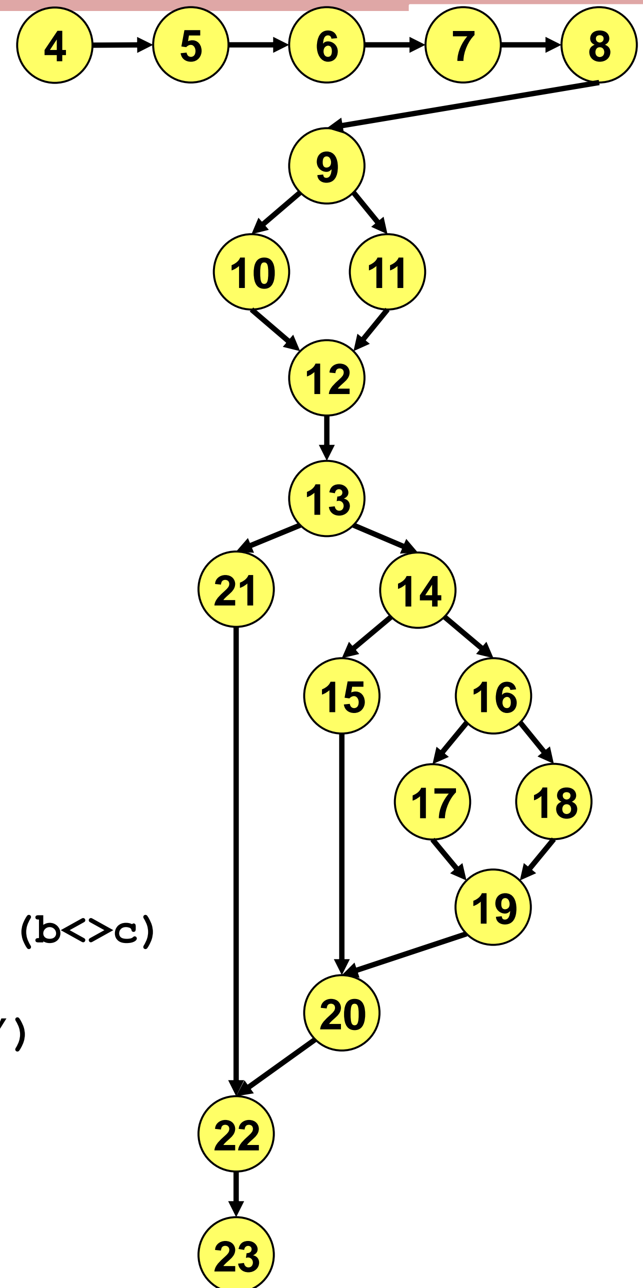
Teste Estrutural

- Exemplo 3: programa com loop

```
... {
    double x = readDouble();  1
    String msg = "";  2
    while(x != 0.0){  3
        if(x >= 0){  4
            msg = "sqr(x) = " + sqr(x);  5
        } else {
            msg = "Error";  6
        }
        System.out.println(msg);  7
    }
    System.out.println("FIM");  8
}
```



```
1. Program Triangle
2. Dim a,b,c AS Integer
3. Dim IsATriangle As Boolean
     #Step 1: Get Input
4. Output(“Enter 3 integers”)
5. Input(a,b,c)
6. Output(“Side A is ”, a)
7. Output(“Side B is ”, b)
8. Output(“Side C is ”, c)
     #Step 2: Is A Triangle?
9. If (a<b+c) AND (b<a+c) AND (c<a+b)
10.   Then IsATriangle = True
11.   Else IsAtriangle = False
12.EndIf
     #Step 3: Determine Triangle Type
13.If IsATriangle
14.   Then If (a=b) AND (b=c)
15.          Then Output(“Equilateral”)
16.          Else If (a<>b) AND (a<>c) AND (b<>c)
17.                 Then Outpput(“Scalene”)
18.                 Else Output(“Isosceles”)
19.               EndIf
20.        EndIf
21.   Else Output(“Not a Triangle”)
22.EndIf
23.End Triangle
```



Teste Estrutural

- Teste de Comandos
  - O critério é que todos os comandos do programa sejam executados pelo menos uma vez
  - Em outras palavras, deve-se percorrer todos os nós do grafo pelo menos uma vez

Teste Estrutural

- Teste de Ramos
  - O critério de teste é exercitar todas as saídas *verdadeiro* e *falso* de todas as decisões
  - Em outras palavras, deve-se percorrer todos os arcos do grafo pelo menos uma vez
  - Cobre o Teste de Comandos

Teste Estrutural

- Teste de Condição
  - O critério de teste é que todas as condições de uma decisão requeiram os valores *verdadeiro* e *falso* pelo menos uma vez (se possível)

Teste Estrutural

- Teste de Fluxo de Dados
  - Estabelece requisitos de teste que seguem o modelo de dados usados dentro do programa
  - Requerem que sejam testadas as interações que envolvam definições de variáveis e subseqüentes referências a estas definições
  - Torna os casos de teste mais rigorosos

Teste Estrutural

- Teste de Fluxo de Dados
  - Cada ocorrência de uma variável dentro de um programa pode ser classificada como sendo uma das seguintes:
    - *def:* definição
    - *c-use:* uso-computacional
    - *p-use:* uso-predicativo

Teste Estrutural

- Teste de Fluxo de Dados
  - **Definição:** quando uma variável é definida através de uma leitura ou quando ela aparece do lado esquerdo de um comando de atribuição, isto é, é dado um valor à variável
  - **Uso-computacional:** quando a variável é usada na avaliação de uma expressão ou em um comando de saída
  - **Uso-predicativo:** quando a variável ocorre em um predicado e portanto, afeta o fluxo de controle do programa

Teste Estrutural

- Critérios do Teste de Fluxo de Dados:
  - **Todas-Definições:** requer que cada definição de variável seja exercitada pelo menos um vez, seja por c-uso ou por p-uso
  - **Todos-Usos:** requer que todas as associações entre uma definição de variável e seus sub-seqüentes usos (*c-usos* e *p-usos*) sejam exercitadas pelos casos de teste, através de pelo menos um caminho livre de definição, ou seja, um caminho onde a variável não é redefinida

Considerações

- É uma técnica mais propensa a automação
- Um problema é a impossibilidade, em geral, de determinar se um caminho é executável e quais valores para fazê-lo
- Problemas triviais podem não ser detectados por critérios de fluxo de controle
- Teste Estruturais devem ser usados com Testes Funcionais

Considerações

- Analisadores de cobertura podem ser usados para unir as duas abordagens de teste
- Essas ferramentas geram métricas de cobertura dos casos de teste
- Quando deveríamos parar de testar?
  - Quando o tempo esgotar-se
  - Quando os testes não encontram mais faltas
  - Quando não se consegue identificar mais casos de teste
  - Quando a métrica de cobertura escolhida for alcançada

UFPB CCAE Centro de Ciências Aplicadas e Educação
CAMPUS IV – LITORAL NORTE

Wilkerson de Lucena Andrade
wilkerson.andrade@gmail.com

Introdução

- EMMA é uma ferramenta open source usada para medir e gerar relatórios de cobertura de código Java
- Verifica quais partes da aplicação estão sendo exercitadas pelo seu conjunto de testes
- Desenvolvida por Vlad Roubtsov

Características

- EMMA instrumenta classes de maneira *offline* e *on the fly*
- Tipos de cobertura suportados:
  - Classe
  - Método
  - Linha
  - Bloco
- EMMA detecta se uma linha de código foi coberta parcialmente

Características

- Tipos de relatório:
  - Texto simples
  - HTML
  - XML
- EMMA não precisa acessar o código fonte
- A instrumentação pode ser realizada em um **.class** individual ou em um **.jar** inteiro
- Pode ser integrada ao ANT

Como usar EMMA?

- Há duas formas de funcionamento:
  - Instrumentação *on the fly*: comando **emmarun**
  - Instrumentação *offline*: comando **emma**
- Onde obter a ferramenta EMMA?
  - **http://emma.sourceforge.net**
- Como instalar?
  - Incluir o arquivo emma.jar no *classpath*

Usando EMMA *on the fly*

- Assumindo que estamos no diretório **examples** da distribuição do EMMA, vamos começar compilando o código:

```
>mkdir out
>javac -d out src/*.java src/search/*.java
```

- Agora podemos executar o exemplo:

```
>java -cp out Main
main(): running doSearch()...
main(): done
```

Usando EMMA *on the fly*

- Para executar o mesmo programa com a coleta de informações de cobertura de código, basta acrescentar **`emmarun`** depois do comando **`java`**:

```
>java emmarun -cp out Main
main(): running doSearch()...
main(): done
EMMA: writing [txt] report to
                             [...coverage.txt] ...
```

Usando EMMA *on the fly*

```
[EMMA report, generated Sun Jan 11
-----------------------------------------------
OVERALL COVERAGE SUMMARY:
[method, %]      [block, %]       [name]
100% (7/7)       95%  (116/122)   all classes

OVERALL STATS SUMMARY:

total classes:   3
total methods:   7

COVERAGE BREAKDOWN BY PACKAGE:

[method, %]      [block, %]       [name]
100% (4/4)       91%  (64/70)     search
100% (3/3)       100% (52/52)     default package
-------------------------------------------------
```

Usando EMMA *on the fly*

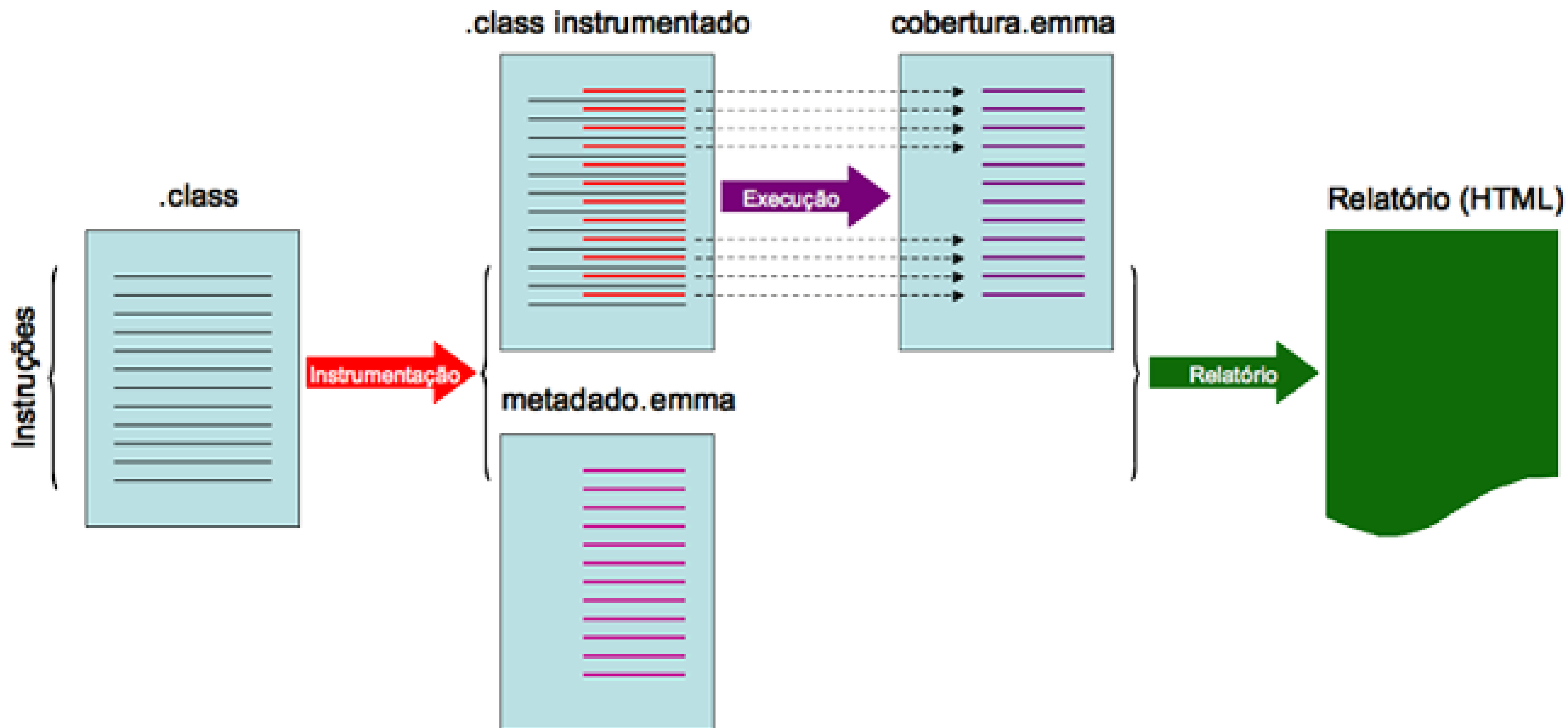
- Quando há a dependência de bibliotecas de terceiros e não queremos incluí-las na análise de cobertura temos duas opções:
  - Colocar a biblioteca no *classpath* da JVM:

```
>java -cp somelib.jar emmarun -cp out Main
```

  - Usar filtros

```
>java emmarun -cp out;somelib.jar
                    -ix +Main,+search.*  Main
```

Como funciona o EMMA *on the fly*?



Como funciona o EMMA *on the fly*?

- EMMA utiliza um *classloader* para instrumentar as classes Java no momento em que elas são carregadas pela JVM
- O *classpath* não é completamente verificado antes do início da execução da aplicação
- A análise é realizada somente nas classes que foram carregadas pela aplicação
- O comando **`–f`** permite analisar a cobertura de todas as classes do *classpath*

Usando EMMA *offline*

- Em alguns casos não conseguimos usar o EMMA *on the fly*
  - Quando queremos excluir classes de teste que estão no mesmo pacote da aplicação e seus nomes não estão padronizados
  - Executar um container J2EE através de um *classloader* de instrumentação é praticamente impossível
  - No desenvolvimento em larga escala há a necessidade de se coletar dados de múltiplas execuções e processos

Usando EMMA *offline*

- As fases de instrumentação, execução e geração de relatórios são separadas
- Usamos o EMMA *offline* através do comando **`emma`**
- Assumindo que estamos no diretório **examples** da distribuição do EMMA, o primeiro passo é compilar o código:

```
>mkdir out
>javac -d out src/*.java src/search/*.java
```

Usando EMMA *offline*

- Agora vamos instrumentar as classes geradas pelo **`javac`** criando um diretório separado para o código instrumentado:

```
>mkdir outinstr
>java emma instr -d outinstr -ip out

EMMA: processing instrumentation path ...
EMMA: instrumentation path processed in 116 ms
EMMA: [3 classes instrumented]
EMMA: metadata merged into [...coverage.em]
```

Usando EMMA *offline*

- Neste momento, a aplicação instrumentada pode ser executada:

```
>java -cp outinstr;out Main

EMMA: collecting runtime coverage data ...
main(): running doSearch()...
main(): done
EMMA: runtime coverage data merged into
                   [...coverage.ec] {in 32 ms}
```

Usando EMMA *offline*

- Finalmente combinamos as informações geradas para a produção do relatório:

```
>java emma report -r txt,html -in
                 coverage.em -in coverage.ec

EMMA: 2 file(s) read and merged in 43 ms
EMMA: writing [txt] report to
                              [...coverage.txt]
EMMA: writing [html] report to
                       [...coverage/index.html]
```

Usando EMMA *offline*

- Podemos utilizar inúmeras fases de instrumentações e execuções
- Com isso, teremos uma quantidade grande de arquivos com informações das instrumentações e execuções
- O comando **`report`** coloca tudo na memória para, só depois, gerar o relatório com as métricas de cobertura

Usando EMMA *offline*

- Com o comando **`merge`** podemos juntar todas as informações geradas e salvar em um só arquivo no disco:

```
>java emma merge -in coverage.em -in
                  coverage.ec -out coverage.es

EMMA: processing input files ...
EMMA: 2 file(s) read and merged in 42 ms
EMMA: merged/compacted data written to
                   [...coverage.es] {in 58 ms}
```

Usando EMMA com JUnit

- Para o exemplo do problema do triângulo usamos o seguinte script:

```
java -cp lib\emma.jar emmarun
-cp .;lib\junit.jar;build
-report html
-sp src
-filter -org.junit*
-filter -junit*
-filter -*Test
-filter -*Tests
org.junit.runner.JUnitCore AllTests
```