Instituto de Computação

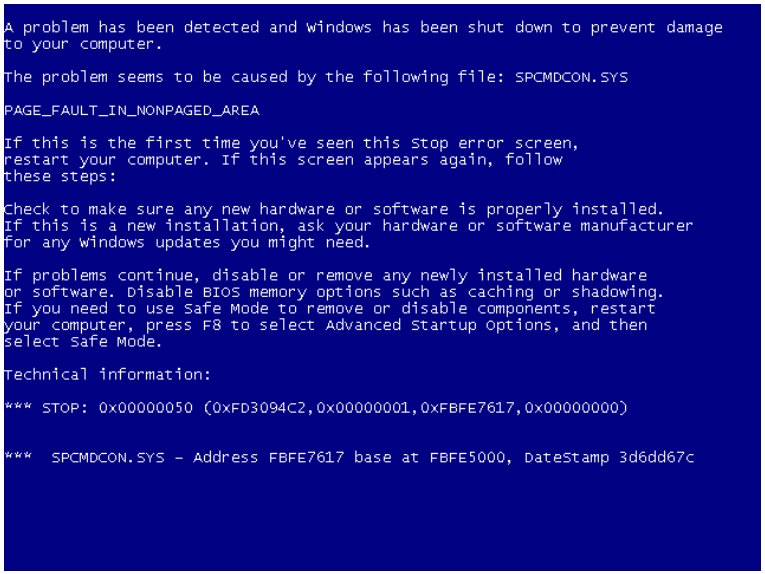
Verificação, Validação e Testes

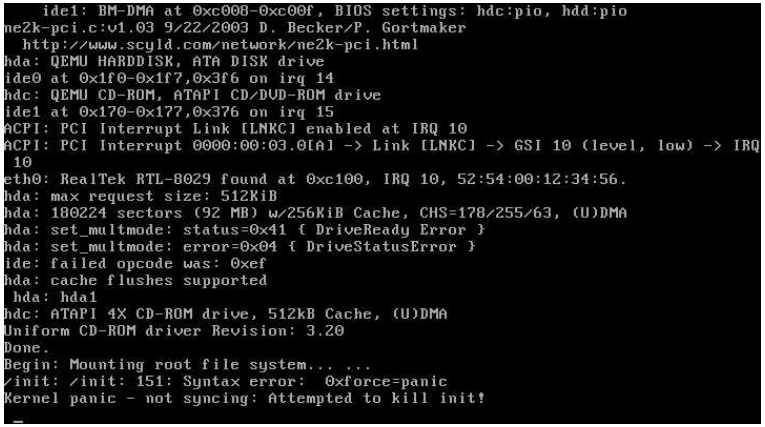
Leonardo Gresta Paulino Murta

leomurta@ic.uff.br

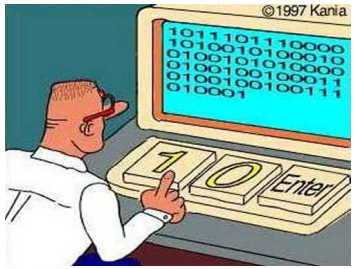
O que é?

- *"Herrar é Umano!!!"*
  - Mas nossos usuários não tem culpa
  - Precisamos fazer o máximo para entregar software de qualidade
- Objetivos de VV&T
  - Assegurar que estamos fazendo de forma correta o produto correto

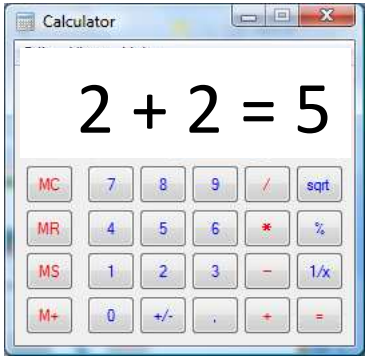




Ciclo de propagação

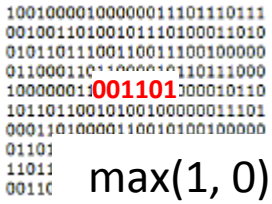


1 – Desenvolvedor comete um **engano** (*mistake*)



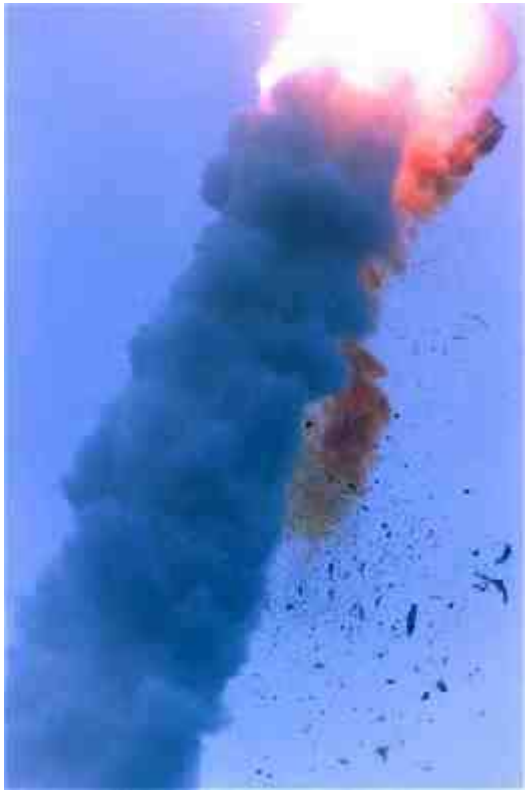
4 – Software **falha** (*failure*)

2 – Software com **defeito** (*fault*)



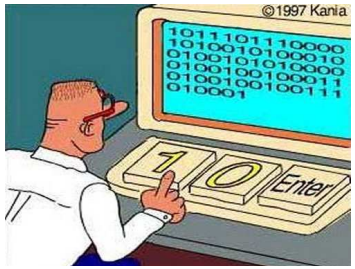
3 – Defeito é exercitado e gera um **erro** (*error*)

max(1, 0) → 0

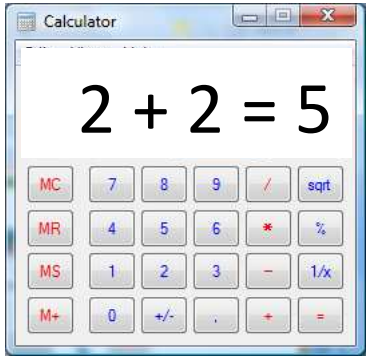


5 – Usuário sofre as consequências

Teste x Depuração

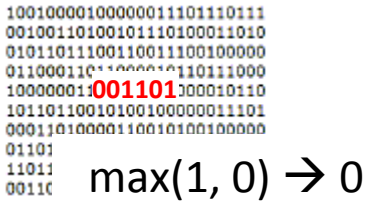


1 – Desenvolvedor comete um **engano** (*mistake*)



4 – Software **falha** (*failure*)

2 – Software com **defeito** (*fault*)

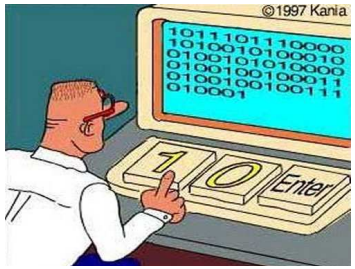


3 – Defeito é exercitado e gera um **erro** (*error*)

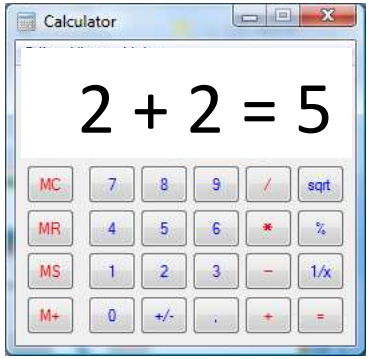
max(1, 0) → 0

Teste busca por **falhas** ou **erros** exercitando o software como um todo ou partes dele

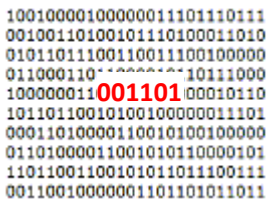
Teste x Depuração



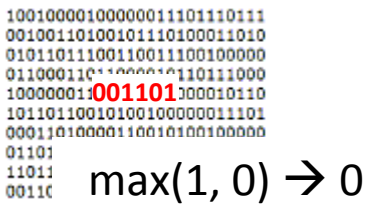
1 – Desenvolvedor comete um **engano** (*mistake*)



4 – Software **falha** (*failure*)



2 – Software com **defeito** (*fault*)



max(1, 0) → 0

3 – Defeito é exercitado e gera um **erro** (*error*)

Depuração busca e corrige **defeitos** que são responsáveis por falhas ou erros do software

Vamos então fazer testes para todas as possibilidades existentes!

O que vocês acham dessa proposta?

Vamos então fazer testes para todas as possibilidades existentes!

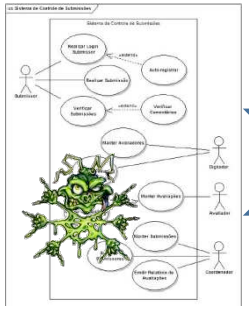
- Infelizmente, isso é **impossível**
- Exemplo
  - Programa simples, com 2 *loops* aninhados que executam 6 vezes cada e 1 *if-then-else* dentro
  - Aproximadamente 2^{36} caminhos de execução possíveis
  - Assumindo que a máquina executa 1 teste por milissegundo
  - Seriam necessários **2 anos** ininterruptos de processamento
- Imagine testar exaustivamente o Debian GNU/LINUX 4, com suas 283 MLOCs!!!

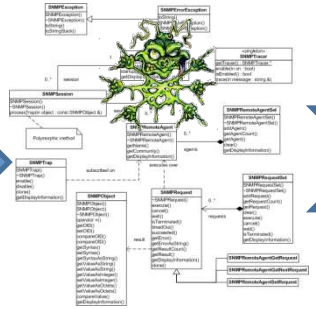
Verificação x Validação

- Verificação
  - Estamos fazendo corretamente o software?
  - Aderência aos requisitos especificados
- Validação
  - Estamos fazendo o software correto?
  - Aderência aos requisitos desejados do usuário

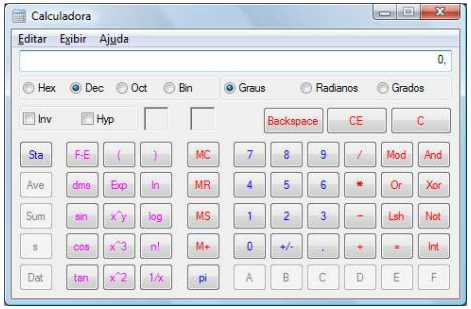
- Não adianta fazer com perfeição um software que não seja o que o usuário deseja!

Verificação x Validação



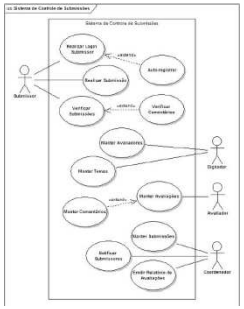


```
C
C     PARTS 1 AND 2, DEFINE SEQUENCE A SECONDS AND CLOUD NA DURING A
C
      KNA=0
      K1=0
230   CALL RANDOM_NUMBER(X1)
      A=-DELTA * LOG(X1)
      IF(A.LE.ALIM) GO TO 250
      IF (K1.GE.KT2) GO TO 240
      K1=K1+1
      GO TO 230
240   A=ALIM/2.0
      X1=0.0
250   K2=0
260   CALL RANDOM_NUMBER(X2)
      IF (JW.GT.1) GO TO 280
270   UX=R*X2
      GO TO 310
280   CALL RANDOM_NUMBER(TEMP)
      IF (TEMP.GE.0.5) GO TO 290
      UX=UPR + R * (1.0-SQRT(X2))
      GO TO 300
290   UX=UPR - R * ( 1.0-SQRT(X2))
300   IF ((UX.GE.0.0).AND.(UX.LE.R)) GO TO 310
      IF (K2.GE.KT2) GO TO 270
      K2=K2+1
      GO TO 260
310   D=UX
      DA=V3 * EXP(U)
```

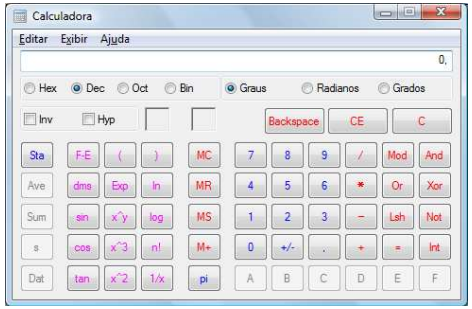


Requisitos Arquitetura Código

Verificação



=

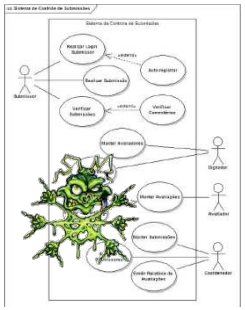


?

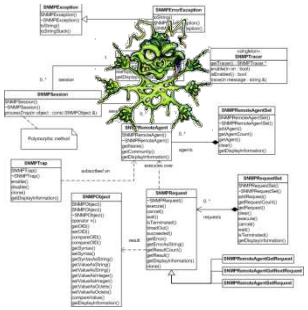
Verificação

Desenvolvimento

Requisitos



Arquitetura



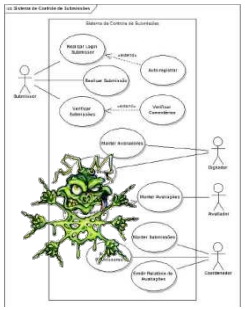
Código

```
C
C     PARTS 1 AND 2, DEFINE SEQUENCE A SECONDS AND CLOUD NA DURING A
C
      KNA=0
      K1=0
  230 CALL RANDOM_NUMBER(X1)
      A=-DELTA * LOG(X1)
      IF(A.LE.ALIM) GO TO 250
      IF (K1.GE.KT2) GO TO 240
      K1=K1+1
      GO TO 230
  240 A=ALIM/2.0
      X1=0.0
  250 K2=0
  260 CALL RANDOM_NUMBER(X2)
      IF (JW.GT.1) GO TO 280
  270 UX=R*X2
      GO TO 310
  280 CALL RANDOM_NUMBER(TEMP)
      IF (TEMP.GE.0.5) GO TO 290
      UX=UPR + R * (1.0-SQRT(X2))
      GO TO 300
  290 UX=UPR - R * ( 1.0-SQRT(X2))
  300 IF ((UX.GE.0.0).AND.(UX.LE.R)) GO TO 310
      IF (K2.GE.KT2) GO TO 270
      K2=K2+1
      GO TO 260
  310 U=UX
      DA=V3 * EXP(U)
```

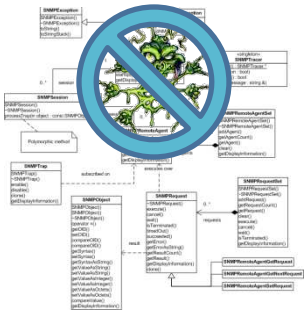
Verificação

Desenvolvimento

Requisitos



Arquitetura



Código

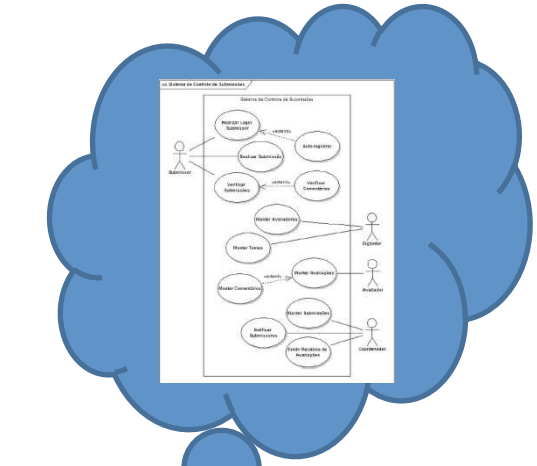
```
C
C     PARTS 1 AND 2, DEFINE SEQUENCE A SECONDS AND CLOUD NA DURING A
C
      KNA=0
      K1=0
  230 CALL RANDOM_NUMBER(X1)
      A=-DELTA * LOG(X1)
      IF(A.LE.ALIM) GO TO 250
      IF (K1.GE.KT2) GO TO 240
      K1=K1+1
      GO TO 230
  240 A=ALIM/2.0
      X1=0.0
  250 K2=0
  260 CALL RANDOM_NUMBER
      IF (JW.GT.1) GO TO
  270 UX=R*X2
      GO TO 310
  280 CALL RANDOM_NUMBER(TE
      IF (TEMP.GE.0.5) GO TO
      UX=UPR + R * (1.0-SQRT(X2
      GO TO 300
  290 UX=UPR - R * ( 1.0-SQRT(X2))
  300 IF ((UX.GE.0.0).AND.(UX.LE.R)) GO TO 310
      IF (K2.GE.KT2) GO TO 270
      K2=K2+1
      GO TO 260
  310 U=UX
      DA=V3 * EXP(U)
```

Teste de Integração

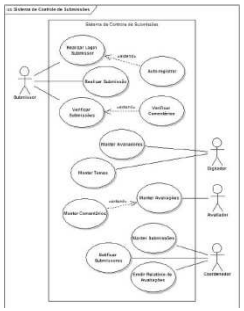
Teste de Unidade

Verificação

Validação

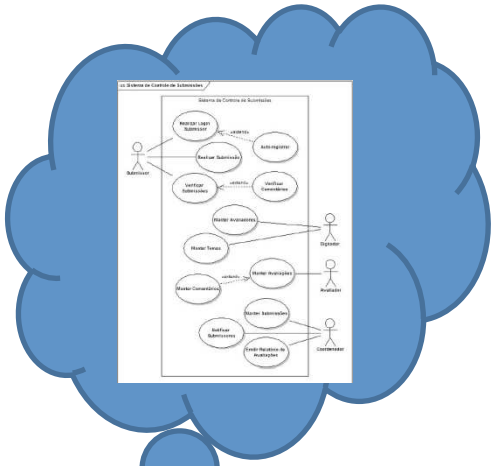


=



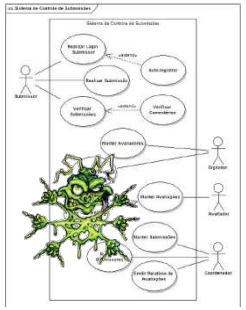
?

Validação

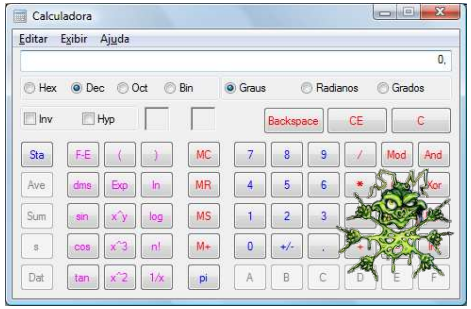


Especificação

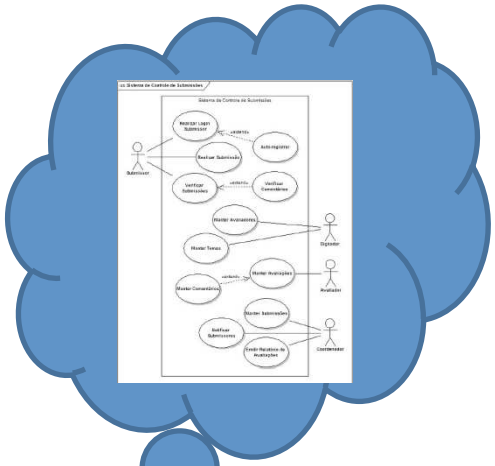
Requisitos



Validação

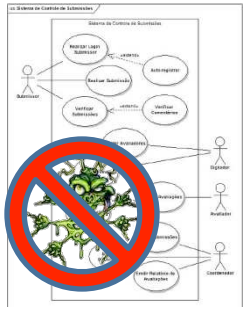


Validação

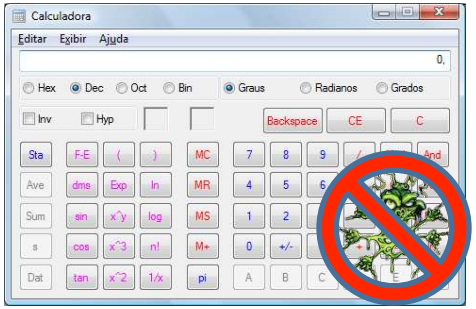


Especificação

Requisitos



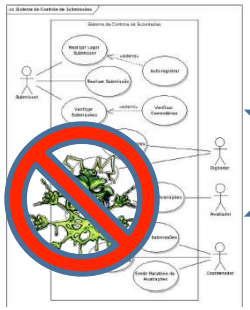
Validação

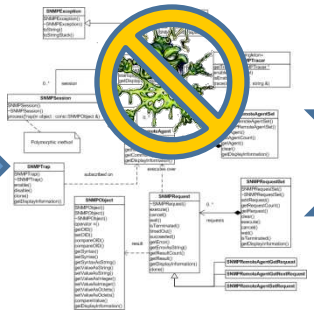


Teste de Aceitação (homologação)

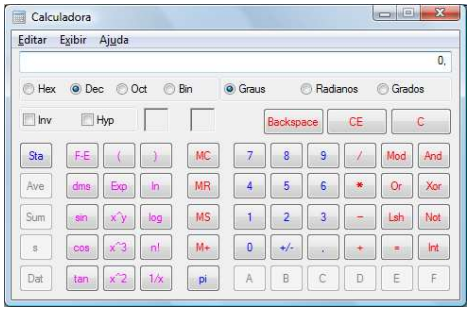
Verificação x Validação

Validação



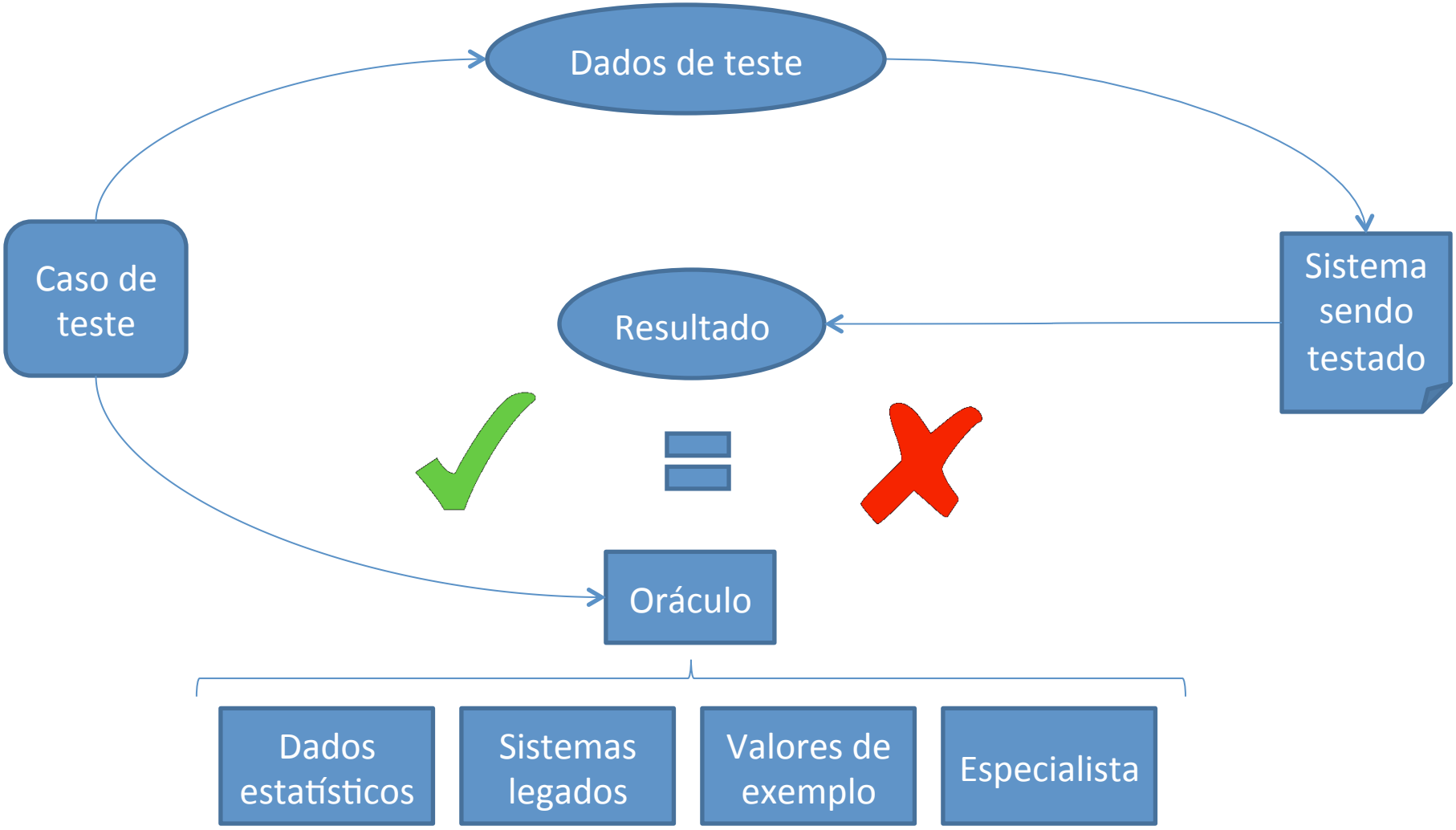


```
C
C     PARTS 1 AND 2, DEFINE SEQUENCE A SECONDS AND CLOUD NA DURING A
C
      KNA=0
      K1=0
230   CALL RANDOM_NUMBER(X1)
      A=-DELTA * LOG(X1)
      IF(A.LE.ALIM) GO TO 250
      IF (K1.GE.KT2) GO TO 24
      K1=K1+1
      GO TO 230
240   A=ALIM/2.0
      X1=0.0
250   K2=0
260   CALL RANDOM_NUMBE
      IF (JW.GT.1) GO TO
270   UX=R*X2
      GO TO 310
280   CALL RANDOM_NUMBER(X
      IF (TEMP.GE.0.5) GO TO
      UX=UPR + R * (1.0-SQRT(X
      GO TO 300
290   UX=UPR - R * ( 1.0-SQRT(X2))
300   IF ((UX.GE.0.0).AND.(UX.LE.R)) GO TO 310
      IF (K2.GE.KT2) GO TO 270
      K2=K2+1
      GO TO 260
310   U=UX
      DA=V3 * EXP(U)
```



Verificação

Anatomia de testes



Pontos importantes

- Verificação não precisa ser feita somente quando existe código
  - Inspeções são técnicas efetivas para identificação de defeitos, mesmo antes de ter código
- Testes devem ser aplicados nas partes, para só então ser aplicado no todo
  - Facilita o isolamento e a localização posterior de defeitos
- Quem faz os testes?
  - O próprio desenvolvedor, em relação às partes (testes de unidade)
  - Uma equipe própria e independente de testes, em relação ao todo (testes de integração)
  - O usuário (testes de aceitação)

Pontos importantes

- Testes não substituem produtos de qualidade
  - Produtos de baixa qualidade, ao serem submetidos a testes, precisarão ser refeitos (**retrabalho**!!!)
- Para que testes sejam efetivos, planejamento é fundamental
  - É necessário estabelecer um **objetivo claro de testes**
  - É importante o alinhamento dos testes com os **perfis dos usuários**

Características de um bom caso de teste

- Ter alta probabilidade de encontrar erros
  - Conhecer o produto e **explorar aspectos diferenciados**
- Não ser redundante
  - **Estabelecer claramente o propósito** de cada teste (planejamento)
- Não ser demasiadamente complexo
  - Decompor os testes de forma que **cada teste foque em somente um objetivo**

Exercício

- Etapa 1
  - Codifique no papel um algoritmo para ordenação
  - Defina alguns testes para esse algoritmo
- Etapa 2
  - Insira propositalmente um defeito no seu algoritmo
- Etapa 3
  - Passe o seu algoritmo para outro grupo e receba o algoritmo de outro grupo
  - Aplique seus testes sobre o algoritmo recebido
  - Seus testes foram capazes de detectar a falha no algoritmo?

Teste caixa branca x caixa preta

- Duas estratégias distintas para elaboração de testes
- Teste caixa branca
  - Também conhecido como teste estrutural
  - Conhece o interior do produto
  - Utiliza esse conhecimento na definição da estratégia de teste
  - Encontra erros
- Teste caixa preta
  - Também conhecido como teste funcional
  - Não conhece o interior do produto
  - Utiliza somente os requisitos na definição da estratégia de teste
  - Encontra falhas

Meta do teste caixa branca

- Garantir que **todos os caminhos independentes foram exercitados** ao menos uma vez
  - Um caminho independente é um caminho que exercita alguma nova instrução ou condição do programa
- Garantir que **todas as decisões lógicas foram exercitadas** do nos dois sentidos (V/F)
- Garantir que **todos os *loops* foram exercitados** nos seus valores de fronteira
- Garantir que as **estruturas de dados internas foram exercitadas** para assegurar a sua integridade

Teste de fumaça

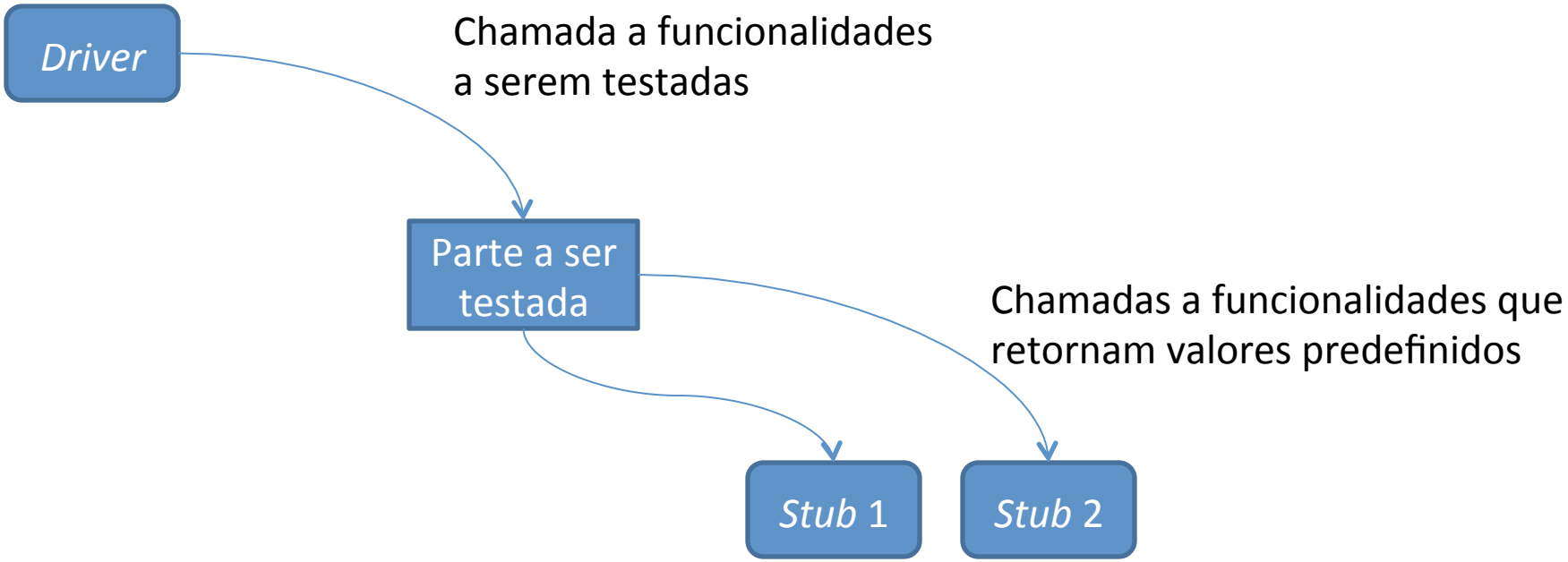
- Metáfora a fumaça gerada por circuito eletrônico com defeito na sua primeira execução
- Consiste em fazer um teste superficial que indica a viabilidade de rodar os demais testes
  - Todas as partes são combinadas e é gerado um *build* do software
  - Esse *build* é submetido a testes básicos
  - Esse processo é repetido frequentemente (e.g., diariamente)

Testes de unidade

- Foco em testar caminhos específicos do produto (caixa branca)
- Visa ter 100% de cobertura
  - Neste caso, 100% de cobertura representando a execução de todas as linhas do código
  - Já vimos que é impossível ter 100% de cobertura para todos os caminhos possíveis de execução!!!
- Normalmente captura erros de cálculos, comparações e fluxo de controle
  - É fundamental **testar as fronteiras**
  - Ex.: valores -1, 0, n, (n+1) para um loop de 0 a n

Testes de unidade (*Drivers* & *stubs*)

- Para viabilizar o teste de unidade, é necessário construir *drivers* e *stubs*



Testes de unidade

- E se não der tempo para fazer todos os testes de unidade?
  - Se concentre ao menos nas partes com maior complexidade
  - Para isso é necessário calcular a **complexidade ciclomática** das partes
- Complexidade ciclomática é o número de caminhos independentes de um programa

Projeto de testes

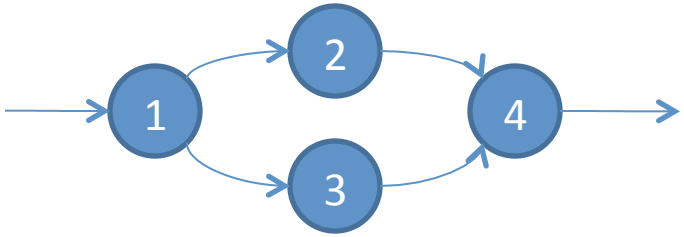
- Método de conjunto básico (*basis set*)
  - Visa exercitar ao menos uma vez todas as sentenças de um programa
  - Se baseia no grafo de fluxo (*flow graph*) do programa e na sua complexidade ciclomática
  - Permite encontrar quantos são os caminhos independentes de teste
  - Permite encontrar possíveis conjuntos mínimos de caminhos independentes de teste

Projeto de testes

1. Gerar o grafo de fluxos (*flow graph*) para cada parte (método, procedimento, função, etc.)

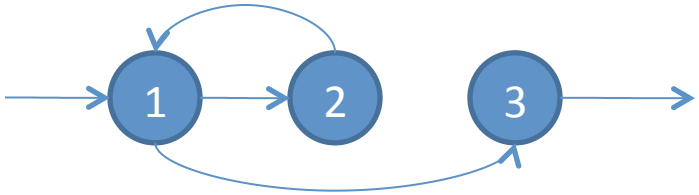
- sentença a; sentença b; sentença c; ... (1)

- if a (1) then b (2) else c (3) endif (4)

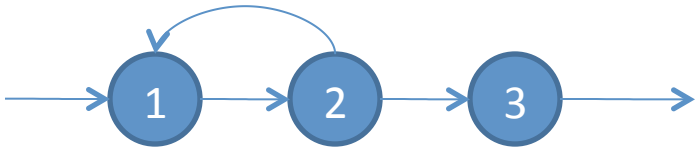


Projeto de testes

- while a (1) do b (2) endwhile (3) [ou]
- for a (1) do b (2) endfor (3)

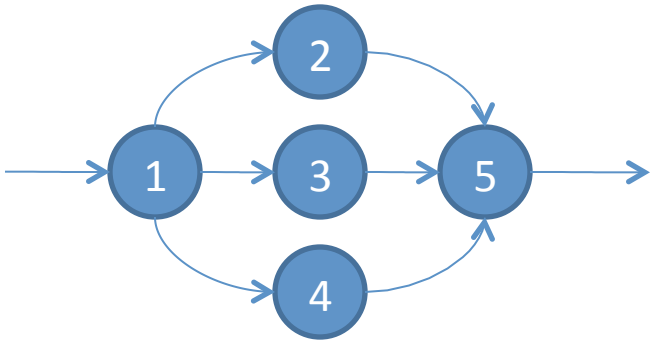


- do a (1) while b (2) enddo (3) [ou]
- repeat a (1) until b (2) enddo (3)

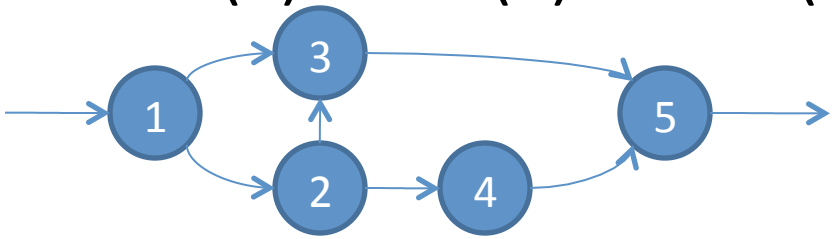


Projeto de testes

- switch a (1) case b (2); case c (3) case d (4) ... endswitch (5)



- Tratamento especial para expressões booleanas
  - if a (1) OU b (2) then c (3) else d (4) endif (5)



Projeto de testes

2. Calcular a complexidade ciclomática do grafo de fluxos

- V(G) = E – N + 2
  - G é o grafo de fluxo
  - V(G) é a complexidade ciclomática do grafo de fluxo G
  - E é o número de arestas do grafo G
  - N é o número de vértices do grafo G

Projeto de testes

3. Identificar V(G) caminhos independentes que formem o conjunto básico
   - Fazer uma busca em profundidade pelos caminhos possíveis, sempre adicionando alguma aresta nova
4. Elaborar casos de teste com valores de entrada que exercitem cada um dos caminhos independentes do conjunto básico

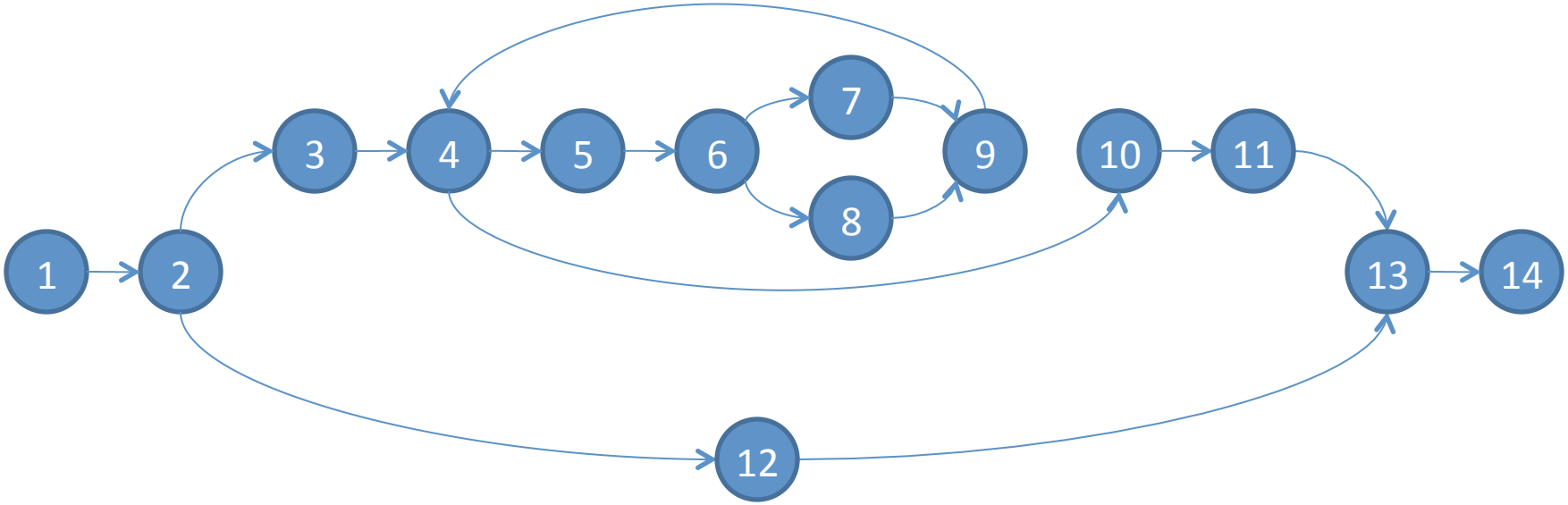
Exemplo (quicksort)

```
public List<String> ordena(List<String> listaDesordenada) {
    List<String> listaOrdenada = new ArrayList<String>();
    if (listaDesordenada.size() > 1) {
       String pivo = listaDesordenada.get(0);
       List<String> listaMenoresDesordenada = new ArrayList<String>();
       List<String> listaMaioresDesordenada = new ArrayList<String>();
       for (int i = 1; i < listaDesordenada.size(); i++) {
          String elemento = listaDesordenada.get(i);
          if (elemento.compareTo(pivo) < 0) { listaMenoresDesordenada.add(elemento); }
          else { listaMaioresDesordenada.add(elemento); }
       }
       listaOrdenada.addAll(ordena(listaMenoresDesordenada));
       listaOrdenada.add(pivo);
       listaOrdenada.addAll(ordena(listaMaioresDesordenada));
    } else { listaOrdenada.addAll(listaDesordenada); }
    return listaOrdenada;
  }
```

Extração do grafo de fluxo...

```
public List<String> ordena(List<String> listaDesordenada) {
1   List<String> listaOrdenada = new ArrayList<String>();
2   if (listaDesordenada.size() > 1) {
        String pivo = listaDesordenada.get(0);
3       List<String> listaMenoresDesordenada = new ArrayList<String>();
        List<String> listaMaioresDesordenada = new ArrayList<String>();
4     for (int i = 1; i < listaDesordenada.size(); i++) {
5       String elemento = listaDesordenada.get(i);
6       if (elemento.compareTo(pivo) < 0) { listaMenoresDesordenada.add(elemento) } 7
8       else { listaMaioresDesordenada.add(elemento); } 9
10    }
        listaOrdenada.addAll(ordena(listaMenoresDesordenada));
11      listaOrdenada.add(pivo);
        listaOrdenada.addAll(ordena(listaMaioresDesordenada));
12  } else { listaOrdenada.addAll(listaDesordenada); } 13
14  return listaOrdenada;
  }
```

Grafo de fluxo e complexidade ciclomática



- V(G) = E – N + 2 = 16 – 14 + 2 = 4
  - Teste 1: 1, 2, 12, 13, 14
  - Teste 2: 1, 2, 3, 4, 10, 11, 13, 14 (**impossível**)
  - Teste 3: 1, 2, 3, 4, 5, 6, 7, 9, 4, 10, 11, 13, 14
  - Teste 4: 1, 2, 3, 4, 5, 6, 8, 9, 4, 10, 11, 13, 14

Exemplo
(teste 1 com JUnit)

```
@Test
  public void teste1() {
     List<String> listaDesordenada = Arrays.asList("abc");
     List<String> oraculo = Arrays.asList("abc");

     List<String> resultado = quicksort.ordena(listaDesordenada);
     assertEquals(oraculo, resultado);
  }
```

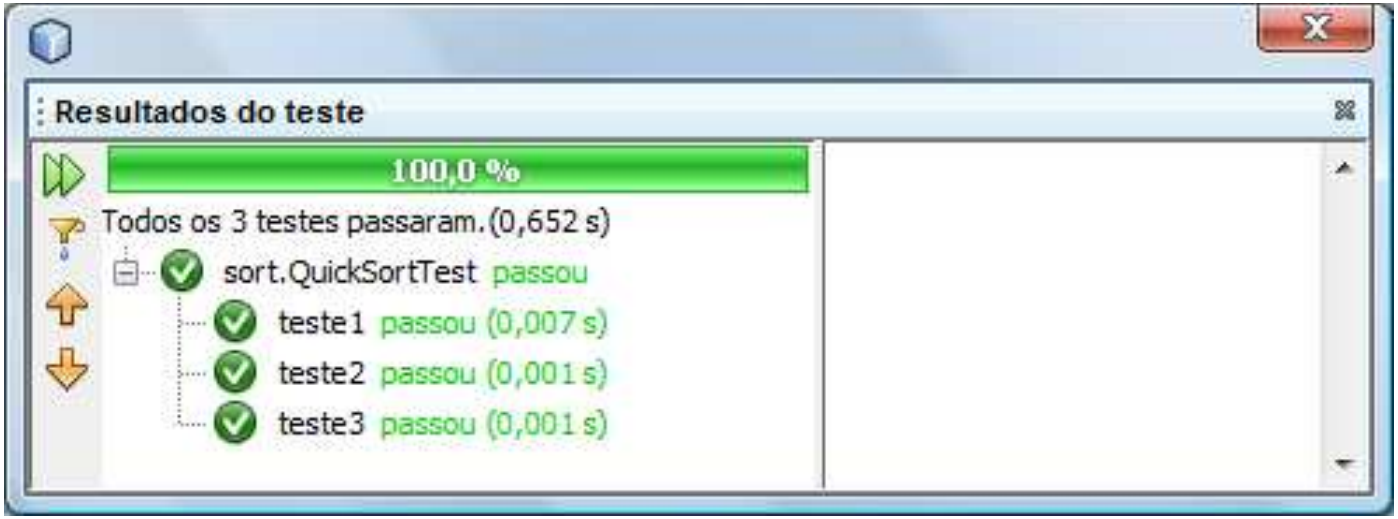
Exemplo (teste 2 com JUnit)

```
@Test
  public void teste2() {
     List<String> listaDesordenada = Arrays.asList("def", "abc");
     List<String> oraculo = Arrays.asList("abc", "def");

     List<String> resultado = quicksort.ordena(listaDesordenada);
     assertEquals(oraculo, resultado);
  }
```

Exemplo (teste 3 com JUnit)

```
@Test
  public void teste3() {
     List<String> listaDesordenada = Arrays.asList("abc", "def");
     List<String> oraculo = Arrays.asList("abc", "def");

     List<String> resultado = quicksort.ordena(listaDesordenada);
     assertEquals(oraculo, resultado);
  }
```

Exemplo
(resultado no JUnit)

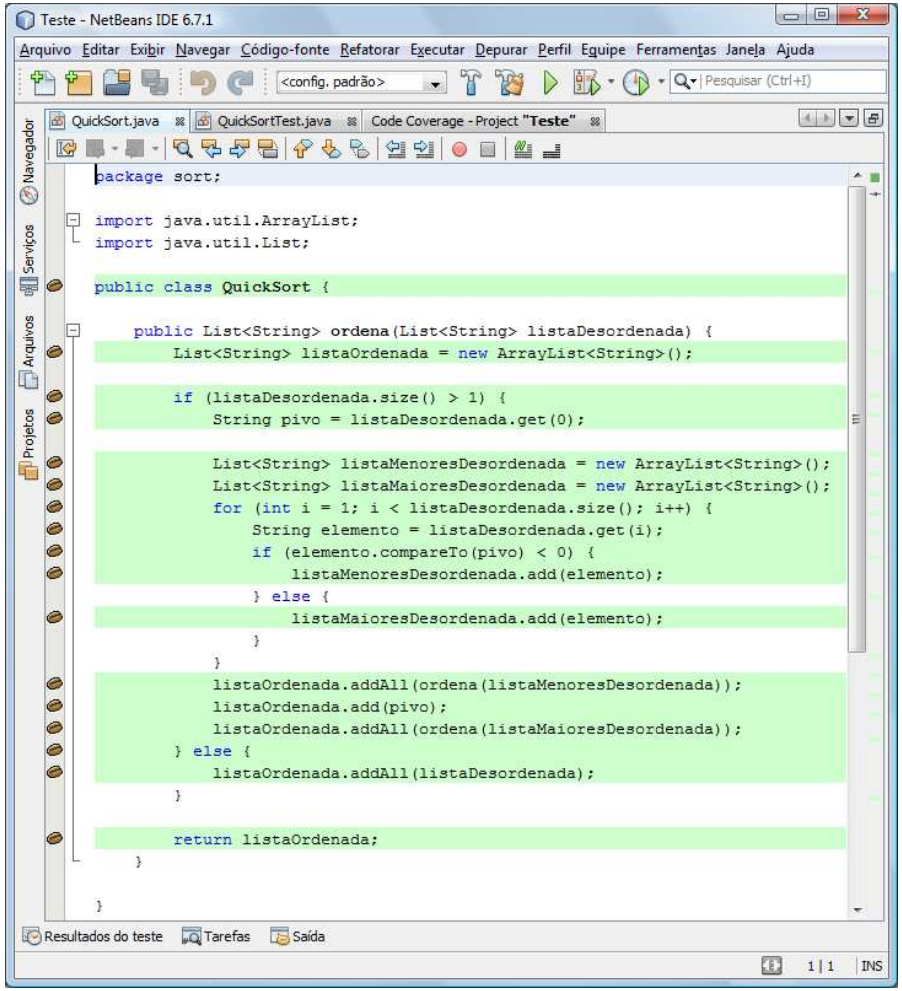


Exemplo (quicksort com defeito)

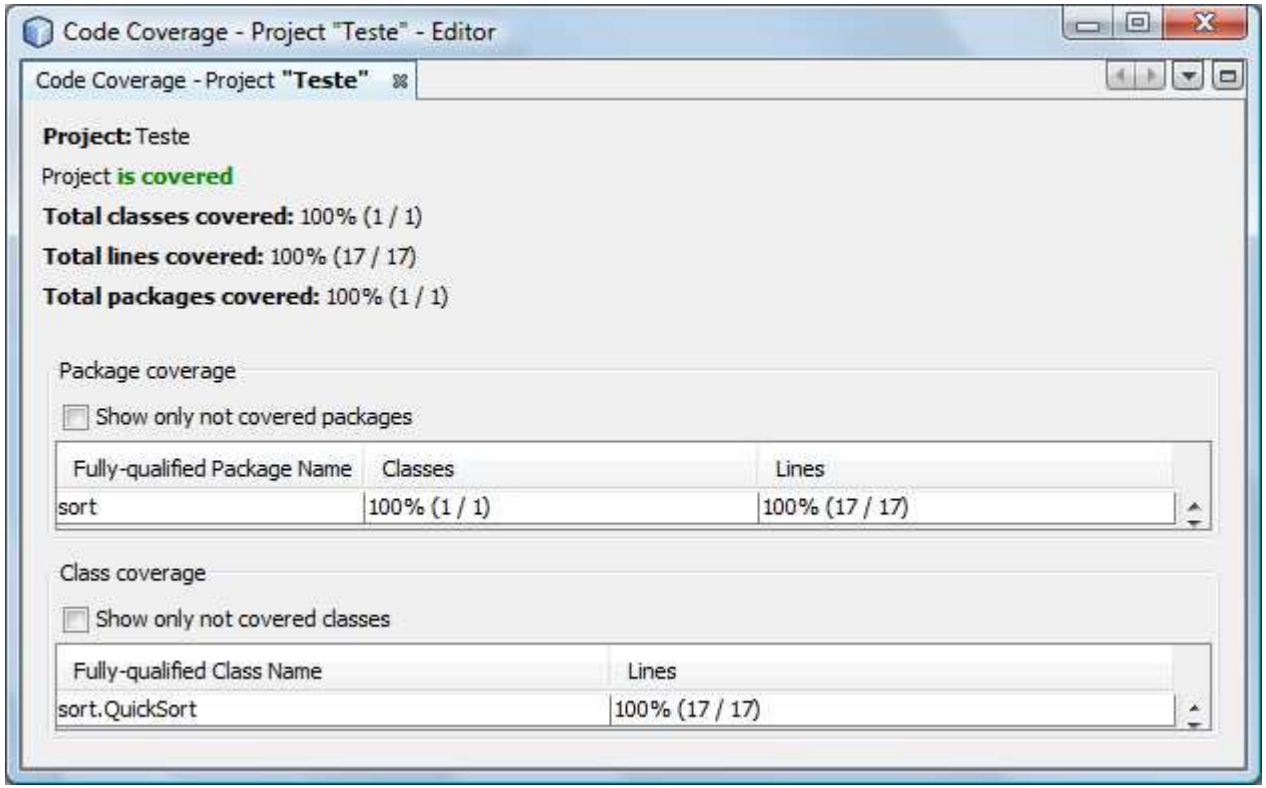
```
public List<String> ordena(List<String> listaDesordenada) {
    List<String> listaOrdenada = new ArrayList<String>();
    if (listaDesordenada.size() > 1) {
        String pivo = listaDesordenada.get(0);
        List<String> listaMenoresDesordenada = new ArrayList<String>();
        List<String> listaMaioresDesordenada = new ArrayList<String>();
        for (int i = 1; i < listaDesordenada.size() - 1; i++) {
            String elemento = listaDesordenada.get(i);
            if (elemento.compareTo(pivo) < 0) { listaMenoresDesordenada.add(elemento) }
            else { listaMaioresDesordenada.add(elemento); }
        }
        listaOrdenada.addAll(ordena(listaMenoresDesordenada));
        listaOrdenada.add(pivo);
        listaOrdenada.addAll(ordena(listaMaioresDesordenada));
    } else { listaOrdenada.addAll(listaDesordenada); }
    return listaOrdenada;
}
```

Exemplo
(resultado no JUnit com defeito)

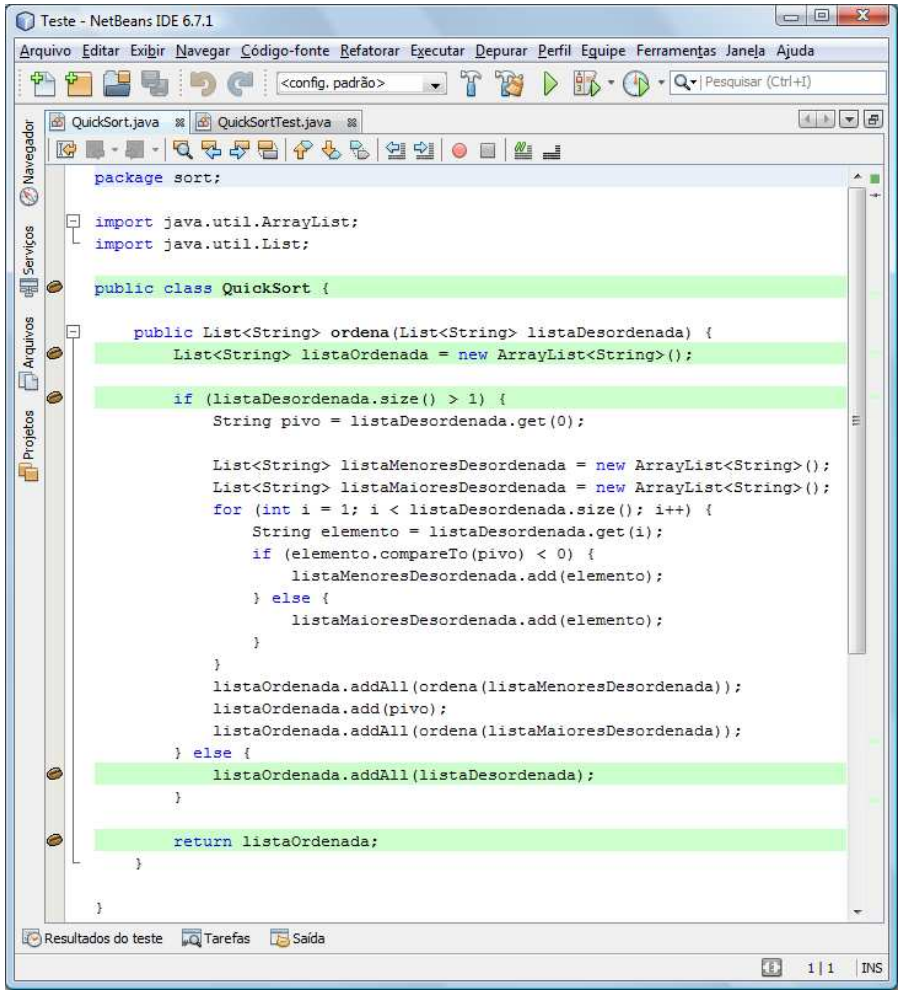
Exemplo
(cobertura dos testes com EMMA)



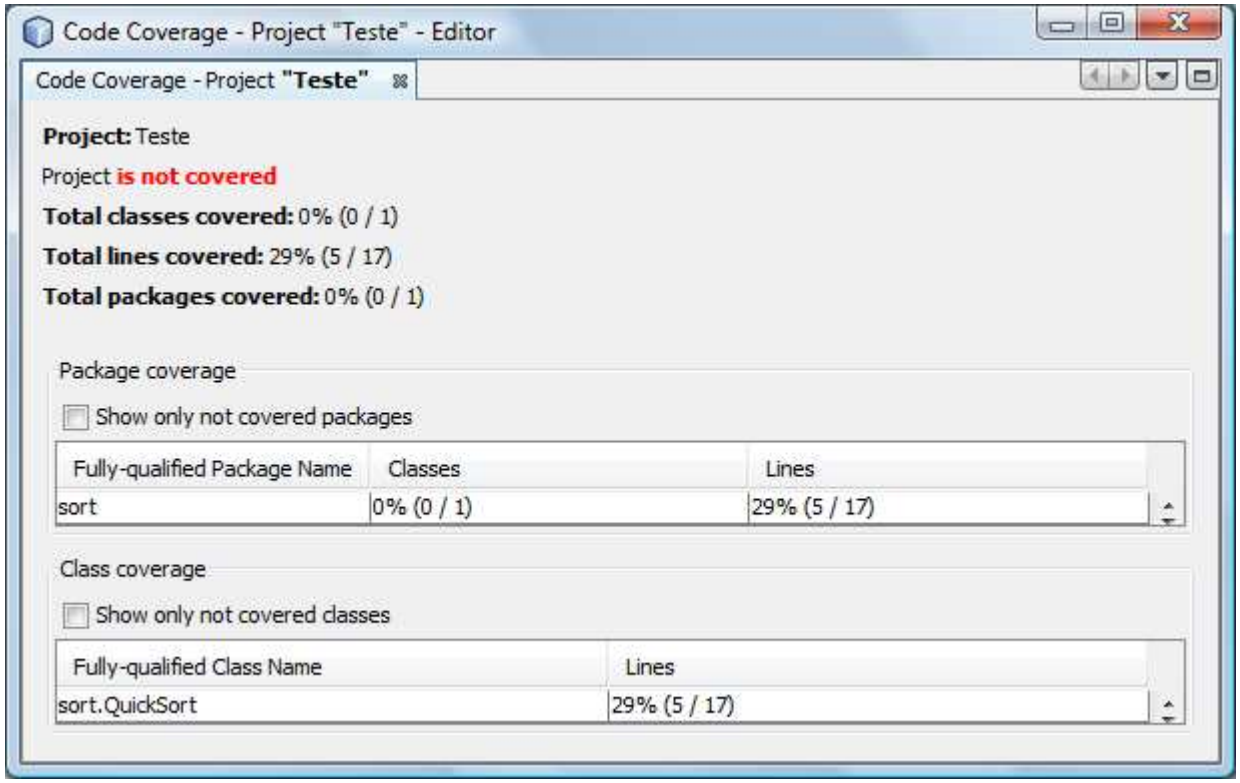
Exemplo
(cobertura dos testes com EMMA)



Exemplo
(cobertura do teste 1 com EMMA)



Exemplo
(cobertura do teste 1 com EMMA)



Projeto de testes (outras estratégias)

- Testes baseados em defeitos
  - Visa identificar os tipos de defeito mais prováveis
  - Projeta testes que são eficazes na descoberta de erros oriundos desses defeitos

- Testes baseados em cenários
  - Projeta testes em função dos principais cenários de uso do sistema, e não nas funcionalidades do sistema
  - É guiado pelo usuário, e não pela estrutura

Tratamento de exceções

- É uma boa prática construir software capaz de tratar as suas próprias exceções (erros)
- Neste caso, o tratamento de exceções precisa também ser testado
  - A mensagem que descreve a exceção é compreensível?
  - A mensagem corresponde ao erro?
  - O mecanismo utilizado para o tratamento é apropriado?
  - A mensagem permitirá que os desenvolvedores localizem o defeito?

Exercício

- Aplique jUnit (ou qualquer outro xUnit) sobre alguma parte do trabalho do curso
- Exiba a cobertura de testes utilizando EMMA (ou qualquer outra ferramenta)
- Dica: o NetBeans tem ambas as ferramentas
  - http://www.netbeans.org
  - JUnit já vem na distribuição padrão do NetBeans
  - EMMA é um plugin - http://codecoverage.netbeans.org

Exercícios

- Traga um exemplo da criação de stubs com a ferramenta Mockito, EasyMock ou jMock (ou alguma outra que você conheça)
  - http://www.mockito.org
  - http://www.easymock.org
  - http://www.jmock.org

Desenvolvimento Dirigido a Testes (*TDD* – método ágil)

- Inverte a ordem, colocando teste antes da codificação (*test first*)
  1. O teste é construído **antes da implementação** da funcionalidade
  2. O **teste deve falhar** nesse momento (vermelho)
  3. É feito o **código mais simples** capaz de atender ao teste (verde)
  4. O **código é refatorado** com o objetivo de aumentar a qualidade do produto
  5. Retorna ao passo 1 enquanto tiverem novas funcionalidades a serem implementadas

Testes de integração

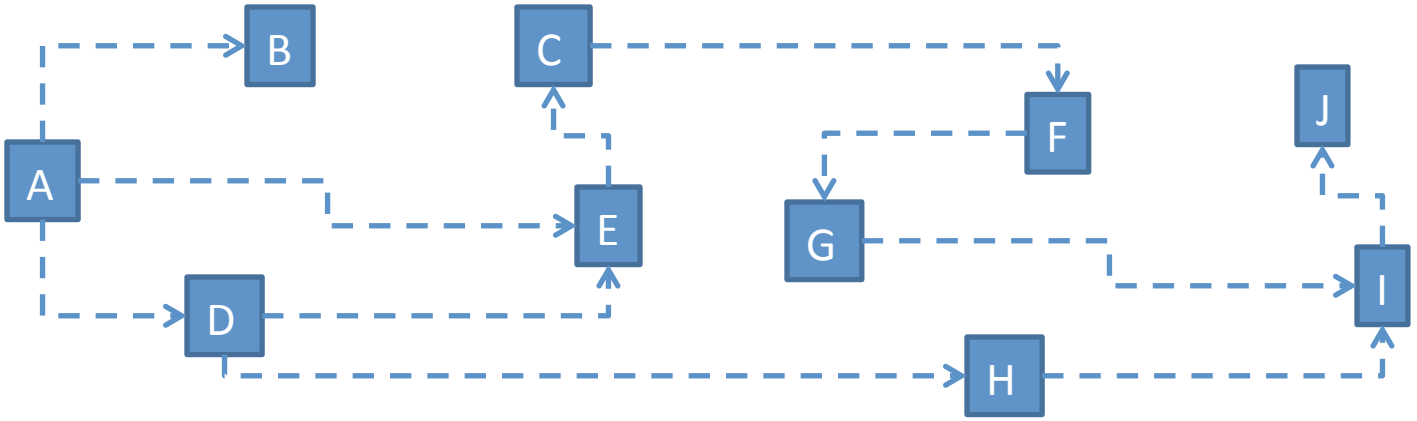
- Foco em combinar as partes do produto e testar as partes em conjunto
- Visa analisar o produto em termos de entradas e saídas (caixa preta)
  - Eventualmente testa também caminhos específicos de grande relevância

Estratégias para integração

- *Big bang*
  - Joga fora os *drivers* e *stubs*, conecta todas as partes, e executa todos os testes de integração
  - Gera normalmente um grande número de erros
  - Torna difícil a atividade de depuração
- Incremental
  - Aos poucos, segundo algum critério, *drivers* e *stubs* são substituídos por partes reais do software, e os testes de integração são executados
  - Os erros aparecem gradativamente
  - O espaço de busca para atividade de depuração é menor

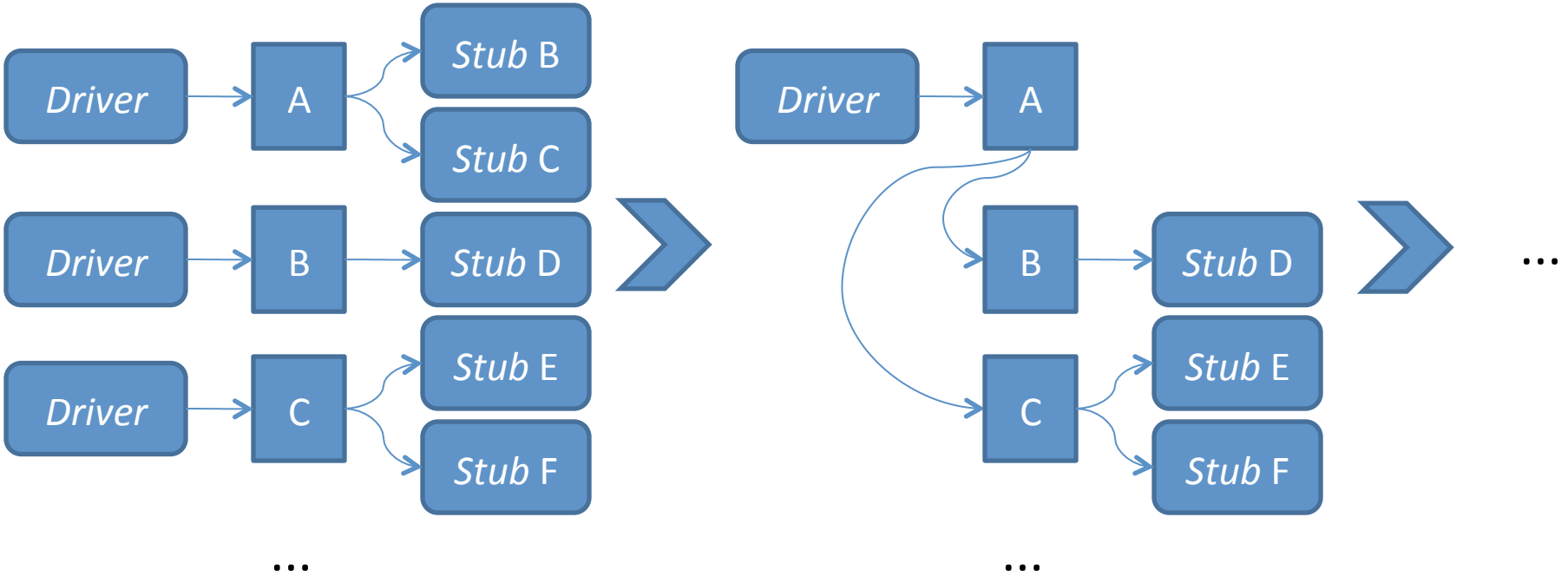
Exercício

- Assuma que os vértices abaixo sejam classes de um sistema orientado a objetos, e as arestas as suas dependências onde A → B significa A depende de B.
  - Qual critério de integração incremental você adotaria?
  - Qual tipo de busca (largura ou profundidade) implementa esse critério?
  - Qual seria uma possível ordem de integração assumindo que a classe A é responsável pela inicialização do sistema (classe raiz)?



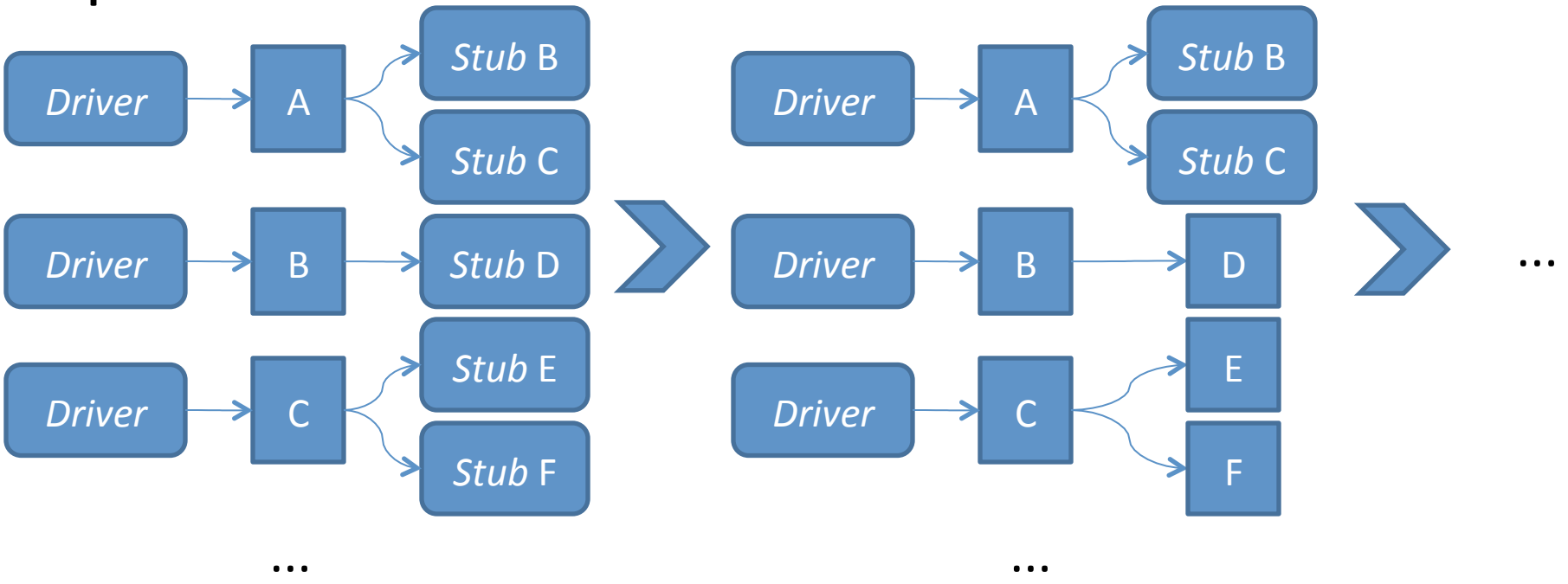
Integração *top-down*

- A parte raiz da árvore de dependências tem seus *stubs* gradativamente substituídos por partes reais do sistema



Integração *bottom-up*

- As partes folha da árvore de dependências têm seus *drivers* gradativamente substituídos por partes reais do sistema



Top-down x *bottom-up*

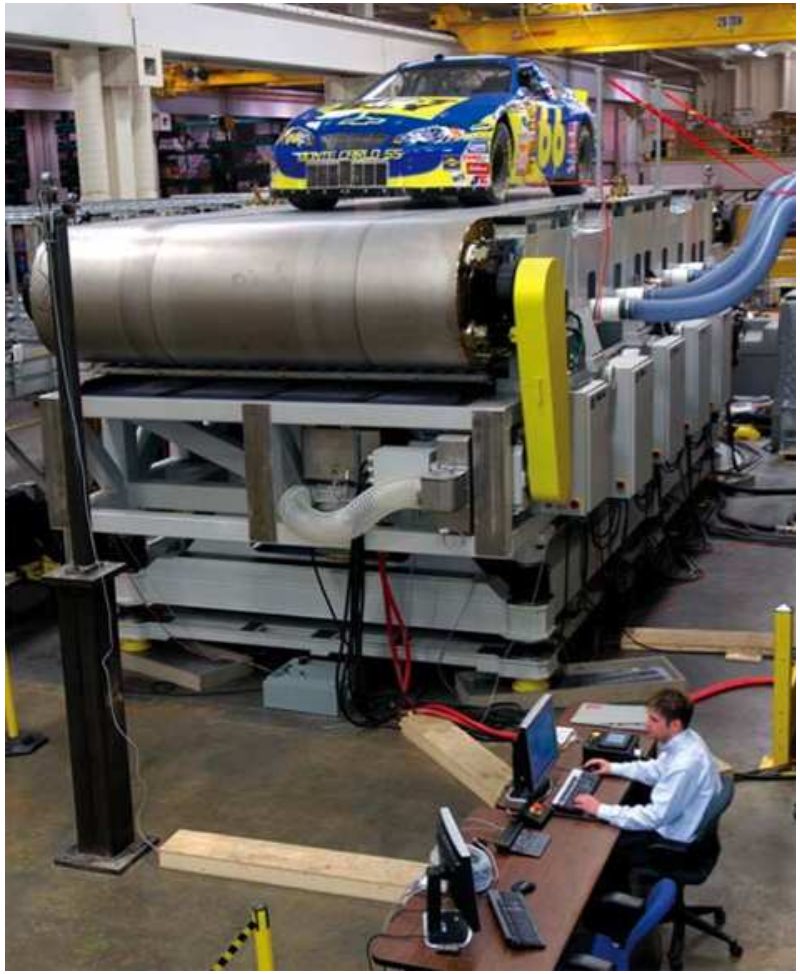
- Integração *top-down*, apesar de fazer sentido, manterá o uso de *stubs* por todos os passos de integração incremental, menos o último
- Integração *bottom-up* de fato substitui gradativamente *stubs* por partes reais
  - Os testes manipulam dados processados e não construídos por *stubs*

Exercício

- Defina a estratégia a ser adotada pelo seu grupo para testes de integração do trabalho do curso

Teste de sistema

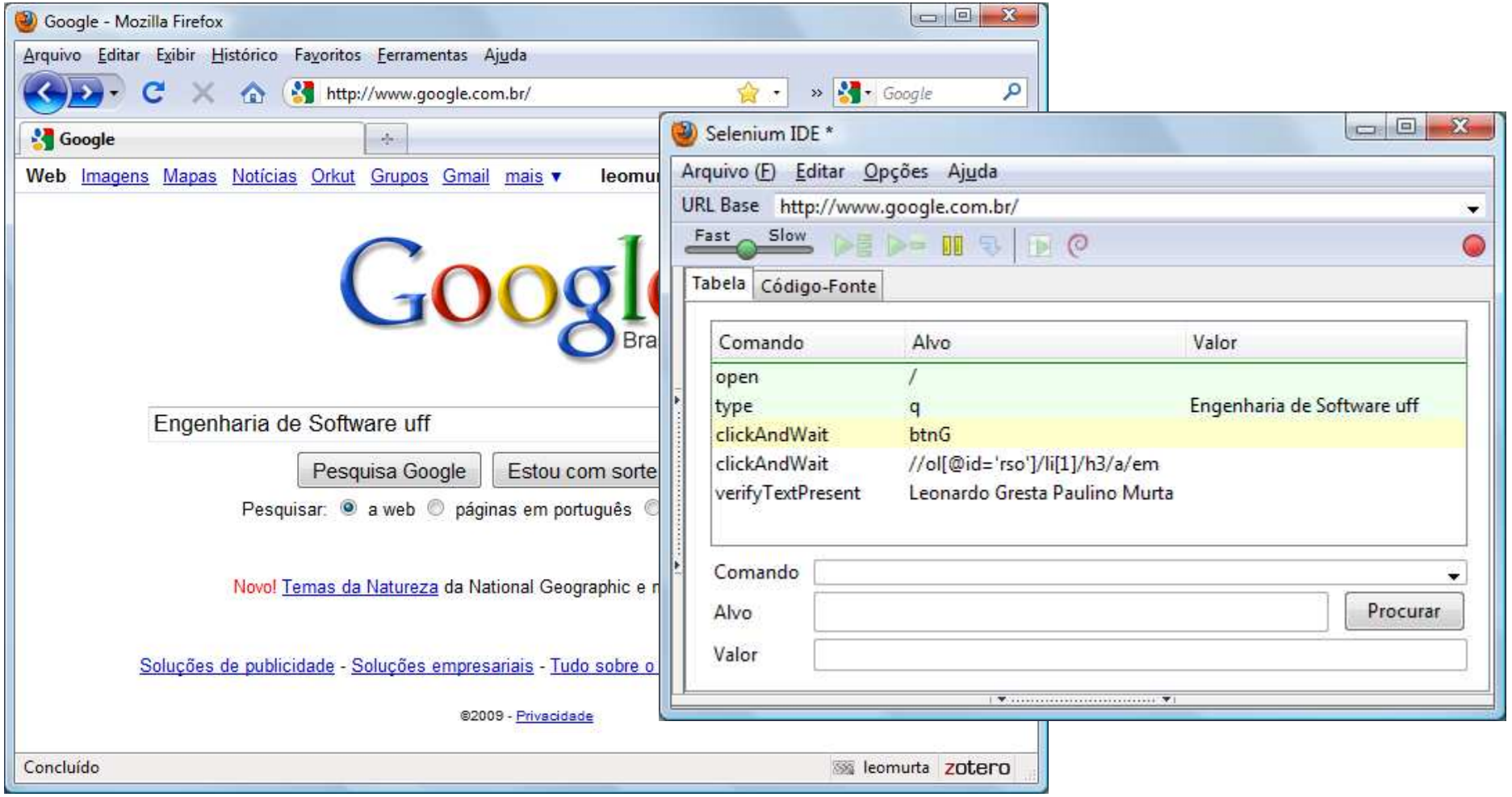
- Transcende o software
- Ocorre depois dos demais teste
- Visa garantir que o software funciona corretamente com os demais elementos do sistema
- Exemplo
  - *Hardware-in-the-loop*



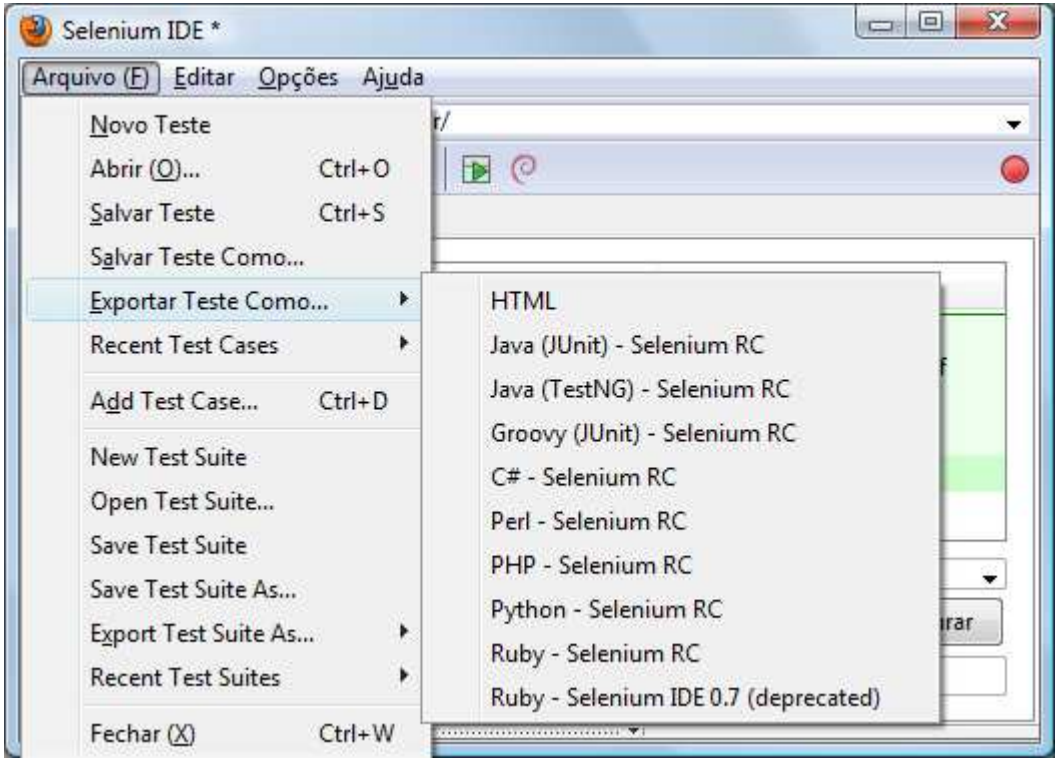
Teste de sistema (tipos)

- Teste de recuperação
  - Força situações extremas
  - Verifica como o sistema se comporta posteriormente
- Teste de segurança
  - Verifica se o sistema tem brechas que possibilitem invasões
  - Em alguns casos, *hackers* são contratados para esse fim
- Teste de estresse (carga)
  - Submete o software a uma elevada demanda de utilização
  - Verifica como a qualidade de serviço varia em função dessa demanda

Exemplo
(Selenium – gravação e reprodução)



Exemplo
(Selenium – exportação)

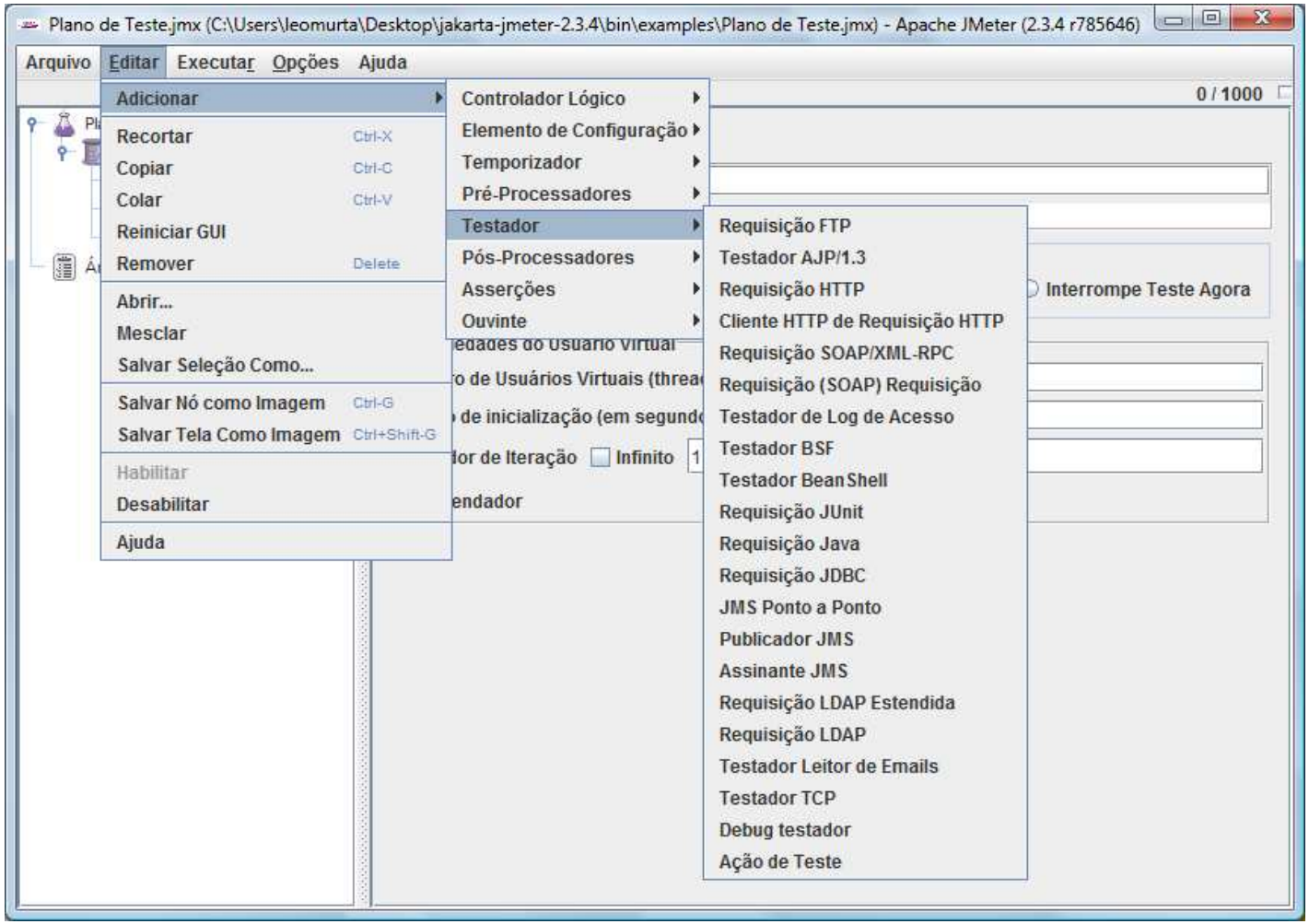


Exemplo (Selenium – exportado para JUnit)

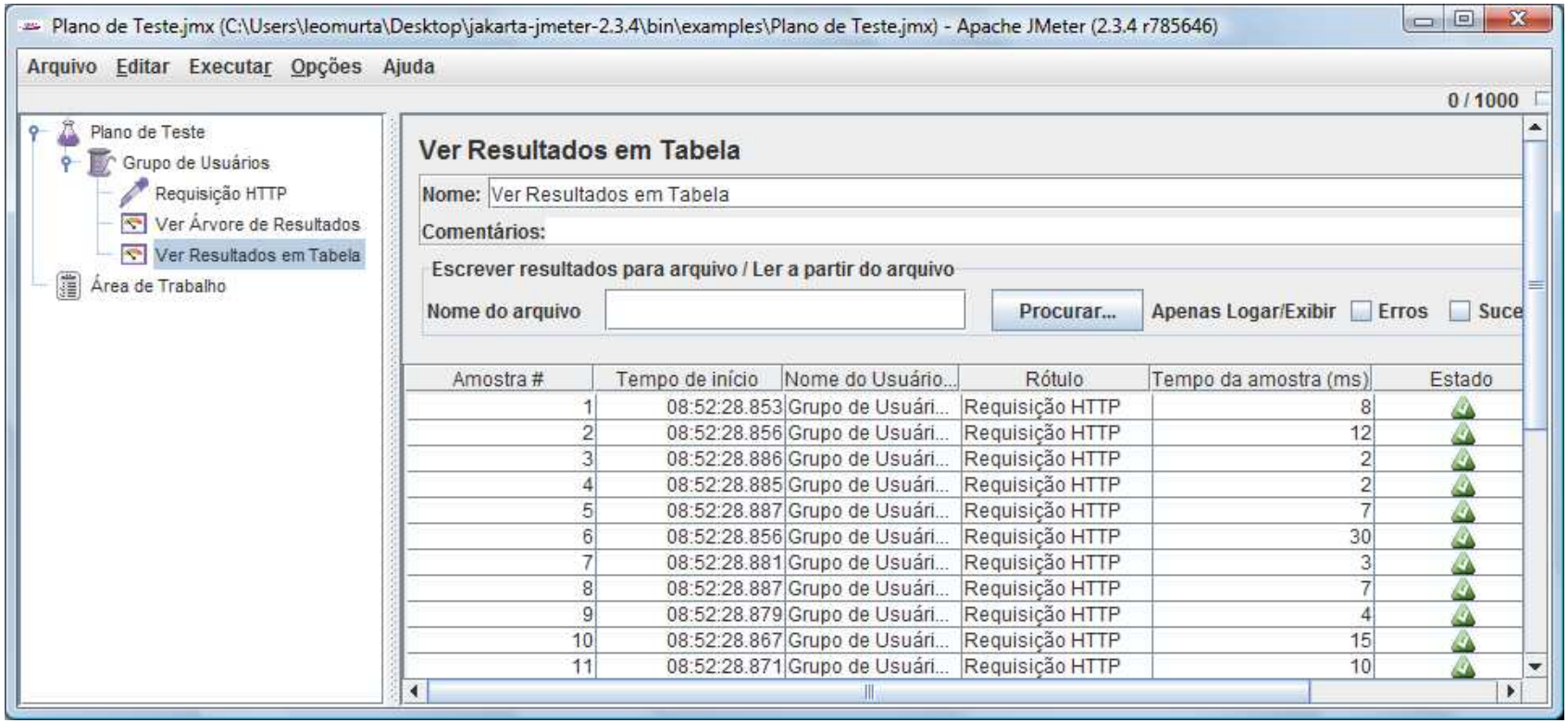
```
package com.example.tests;

import com.thoughtworks.selenium.*;
import java.util.regex.Pattern;

public class Untitled extends SeleneseTestCase {
	public void setUp() throws Exception {
		setUp("http://www.google.com.br/", "*chrome");
	}
	public void testUntitled() throws Exception {
		selenium.open("/");
		selenium.type("q", "Engenharia de Software uff");
		selenium.click("btnG");
		selenium.waitForPageToLoad("30000");
		selenium.click("//ol[@id='rso']/li[1]/h3/a/em");
		selenium.waitForPageToLoad("30000");
		verifyTrue(selenium.isTextPresent("Leonardo Gresta Paulino Murta"));
	}
}
```

Exemplo
(jMeter – configuração da carga)

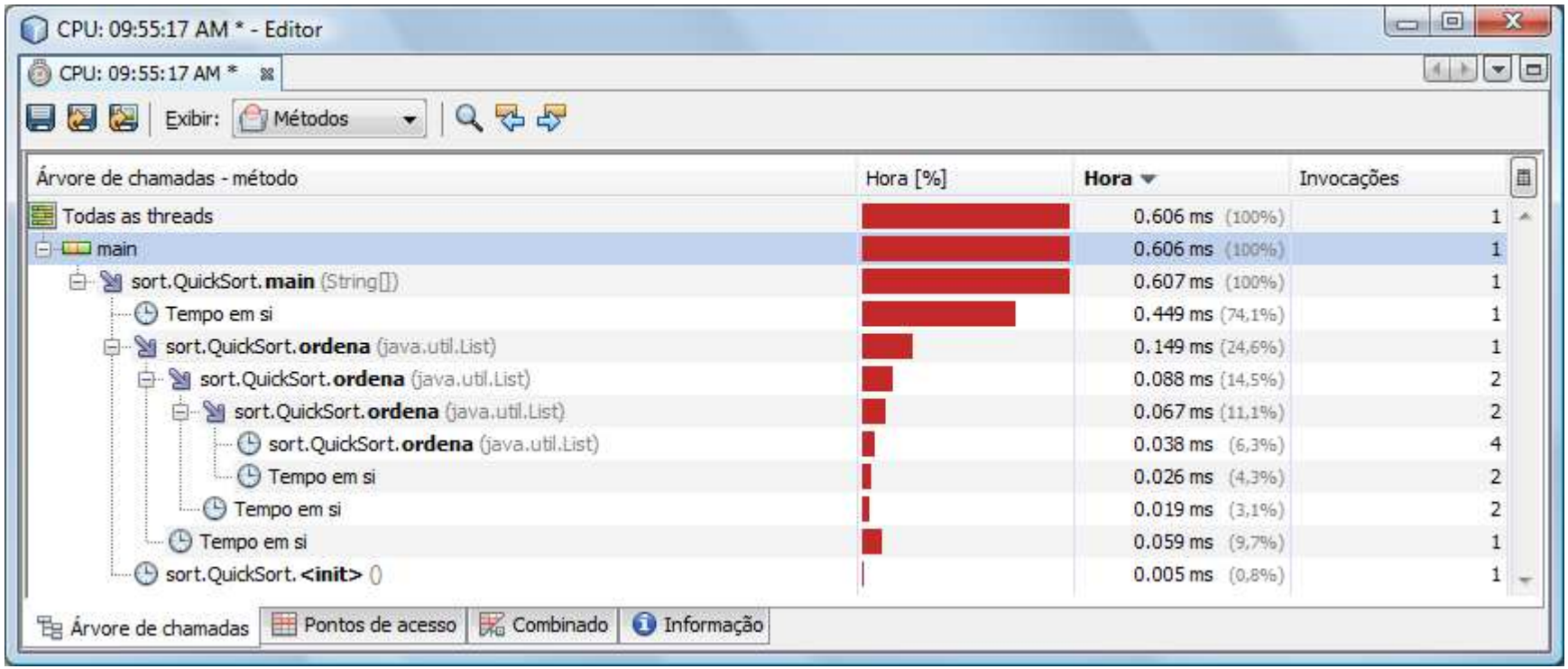
Exemplo
(jMeter – adição de testador)



Exemplo
(jMeter – resultados)



Exemplo (*profiling* de CPU)



Exemplo
(*profiling* de memória)

Exercício

- Utilize Selenium, jMeter ou algum *profiler* sobre o trabalho do curso e apresente os resultados obtidos

Teste de regressão

- Não é mais um tipo de teste, mas sim um **papel** que pode ser empenhado por diferentes tipos de teste
- Visa **evitar que defeitos já corrigidos retornem** ao produto
- Muito usado em testes de integração, onde testes anteriores são aplicados

Testes de aceitação

- Foco em apresentar o produto ao usuário para que o produto seja homologado
- Visa estabelecer critérios para aceitação
  - Funcionais
  - Comportamentais
  - De desempenho
- Tipos de teste de aceitação
  - Alfa
  - Beta

Cenário típico de validação

Semanas ou meses

Testes informais (*ad hoc*)

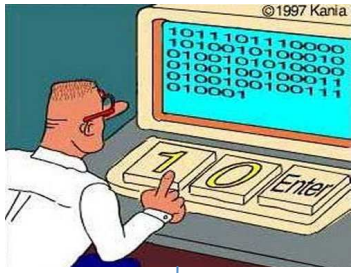
Sistema sendo testado

Utilização em paralelo

Sistema antigo (manual ou automatizado)

Testes formais (critérios, planos, etc.)

Testes alfa



Ambiente controlado (e.g., local do desenvolvimento)

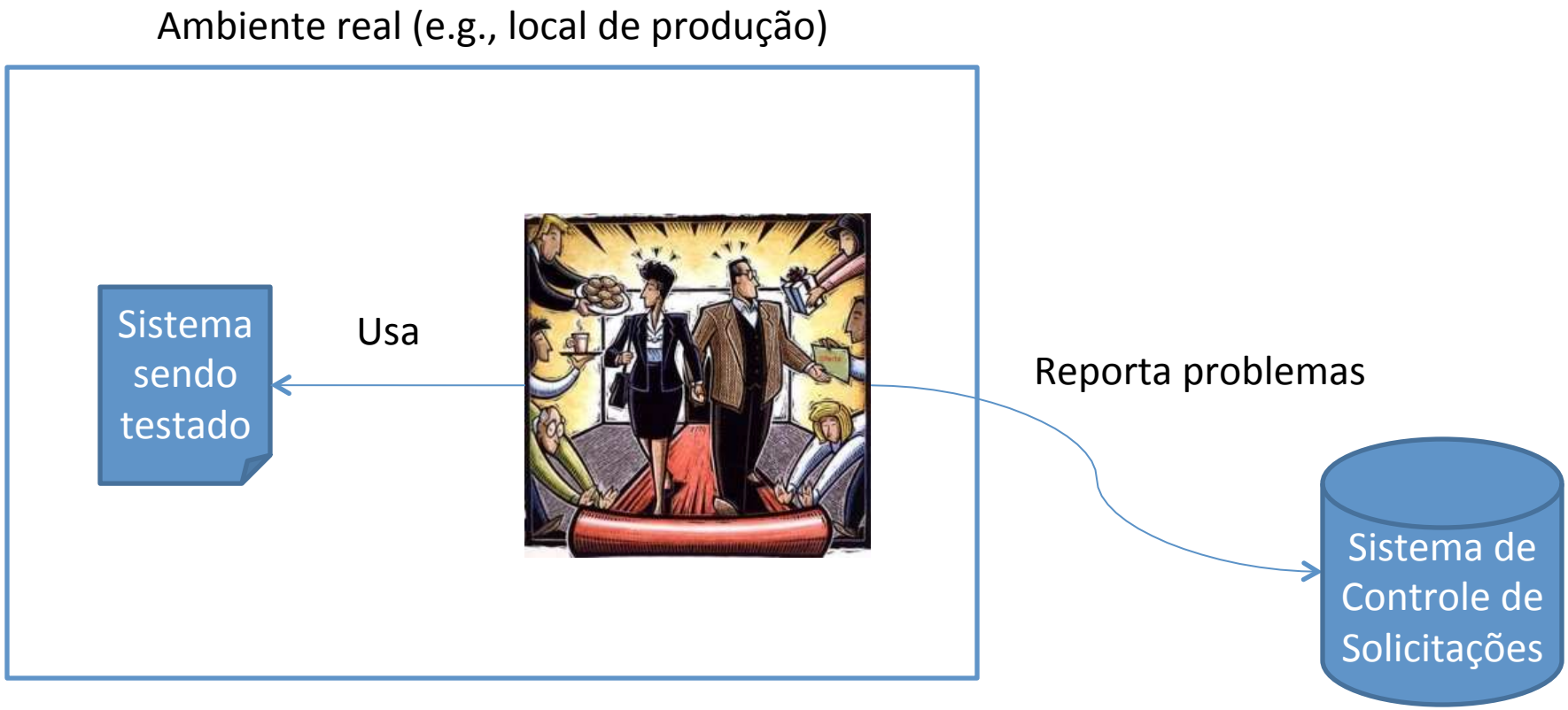
Sistema sendo testado

Usa

(numero reduzido)

Observa

Testes beta


Ambiente real (e.g., local de produção)
Sistema sendo testado
Usa
Reporta problemas
Sistema de Controle de Solicitações

Referências

- Pilone, D.; Miles, R.; 2008. Head First Software Development. O'Reilly Media.
- Pressman, R. S.; 2004. Software Engineering: A Practitioner's Approach. 6 ed. McGraw-Hill.

Instituto de Computação

Verificação, Validação e Testes

Leonardo Gresta Paulino Murta

leomurta@ic.uff.br