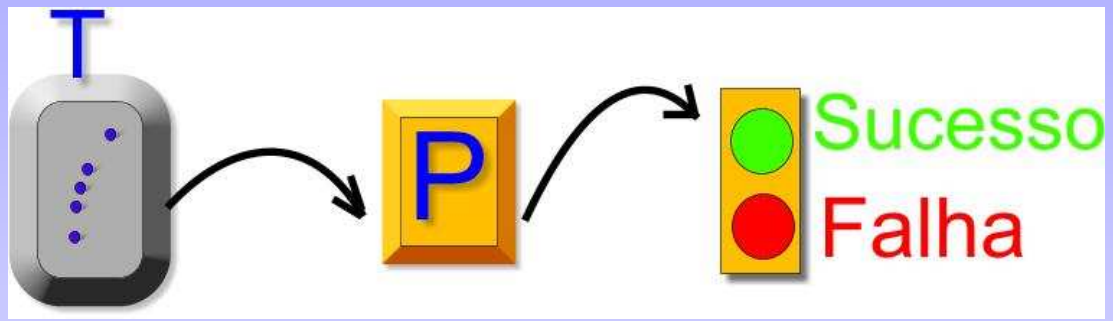
SSC 0721 – Teste e Validação de Software

Conceitos básicos

Prof. Marcio E. Delamaro
delamaro@icmc.usp.br

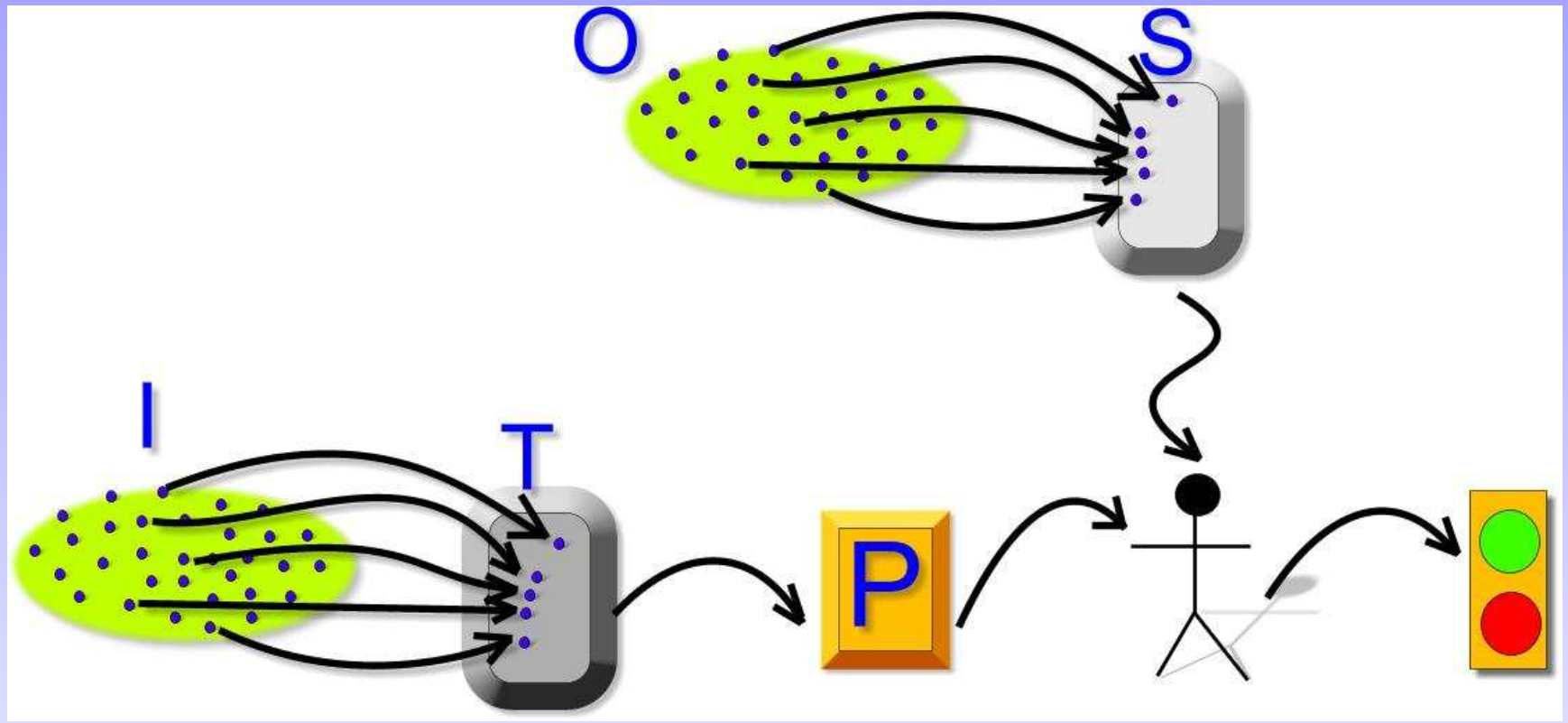
O que é teste

- Atividade de executar um programa e verificar se o seu comportamento é o esperado



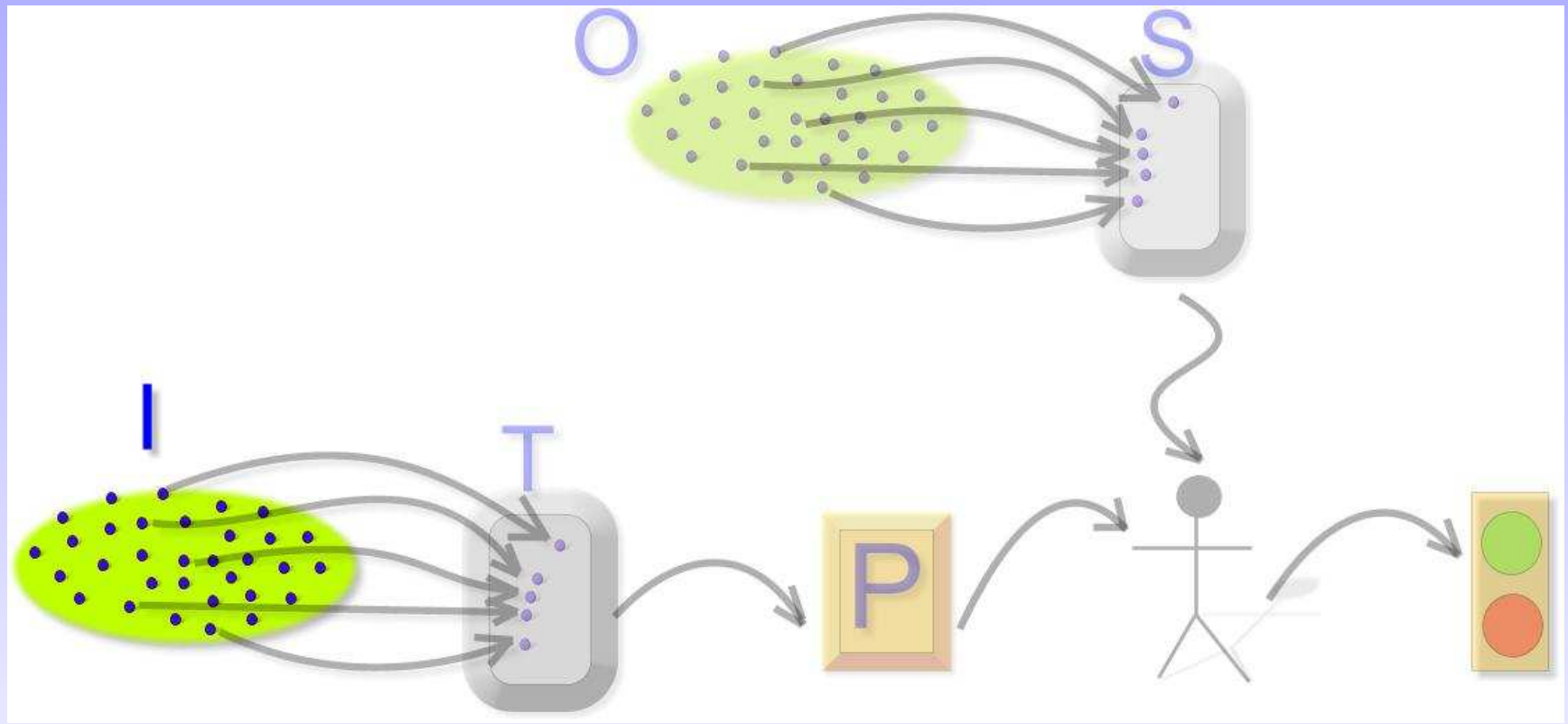
- Objetivo: revelar defeitos

O que é teste – detalhes



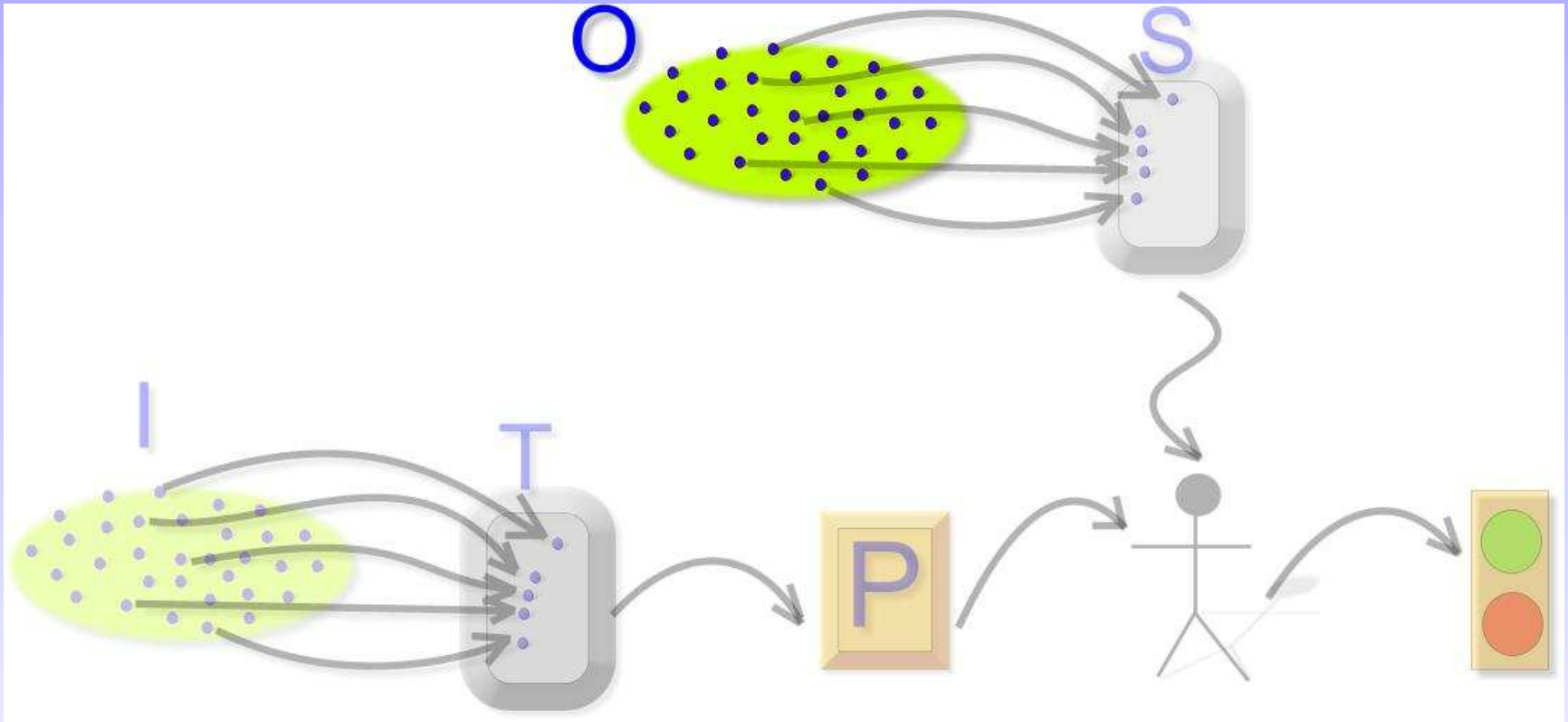
Domínio de entrada

Conjunto de todos os dados que o programa deve tratar.



Domínio de saída

Todos os possíveis resultados que o programa deve fornecer.



Domínios – exemplo

- Fatorial(x)

Domínios – exemplo

- Fatorial(x)
- I = {0, 1, 2, 3, 4,...}

Domínios – exemplo

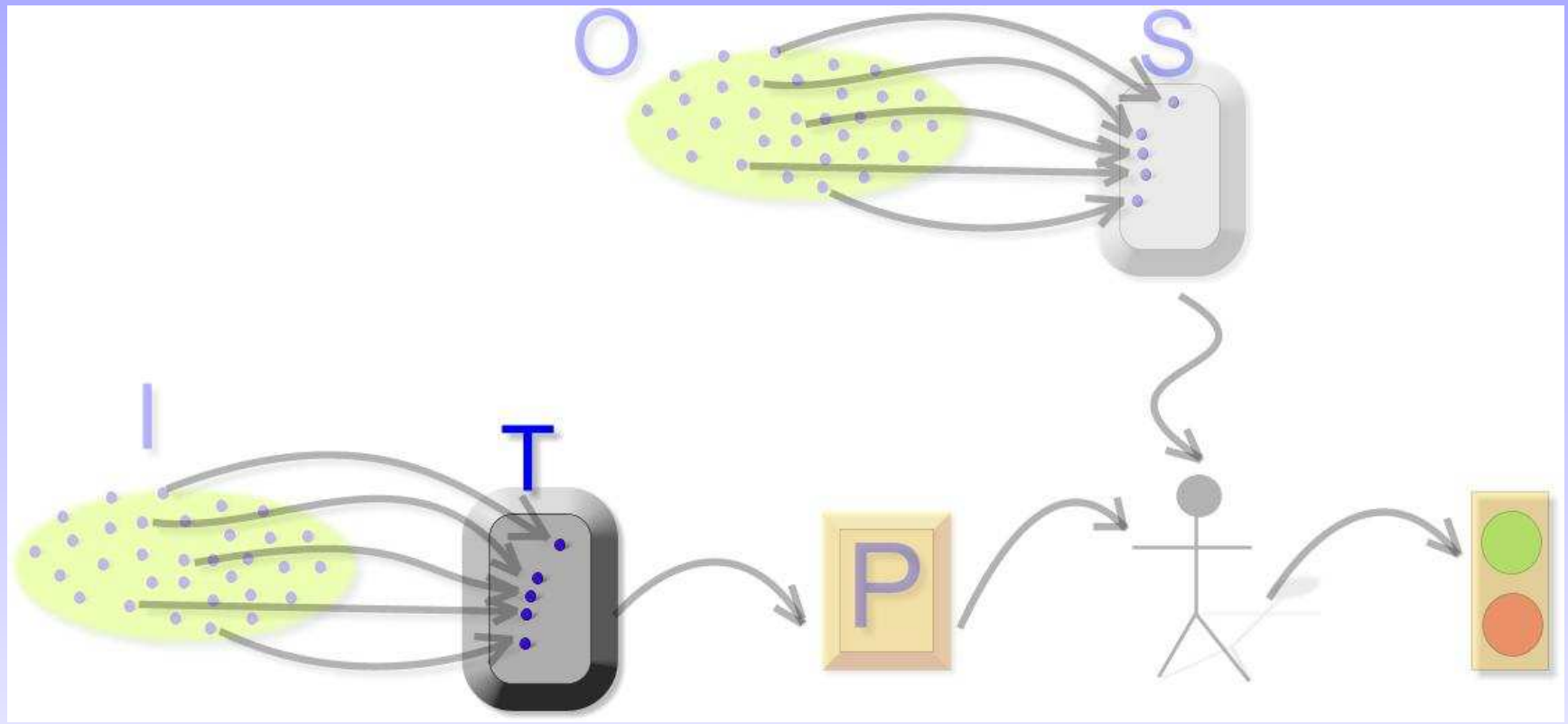
- Fatorial(x)
- I = {0, 1, 2, 3, 4,...}
- O = {1, 2, 6, 24, ...}

Domínios – exemplo

- Fatorial(x)
- I = {0, 1, 2, 3, 4,...}
- O = {1, 2, 6, 24, ...}
- Fatorial(-3) – devo testar meu programa com elementos que não pertencem ao domínio de entrada?

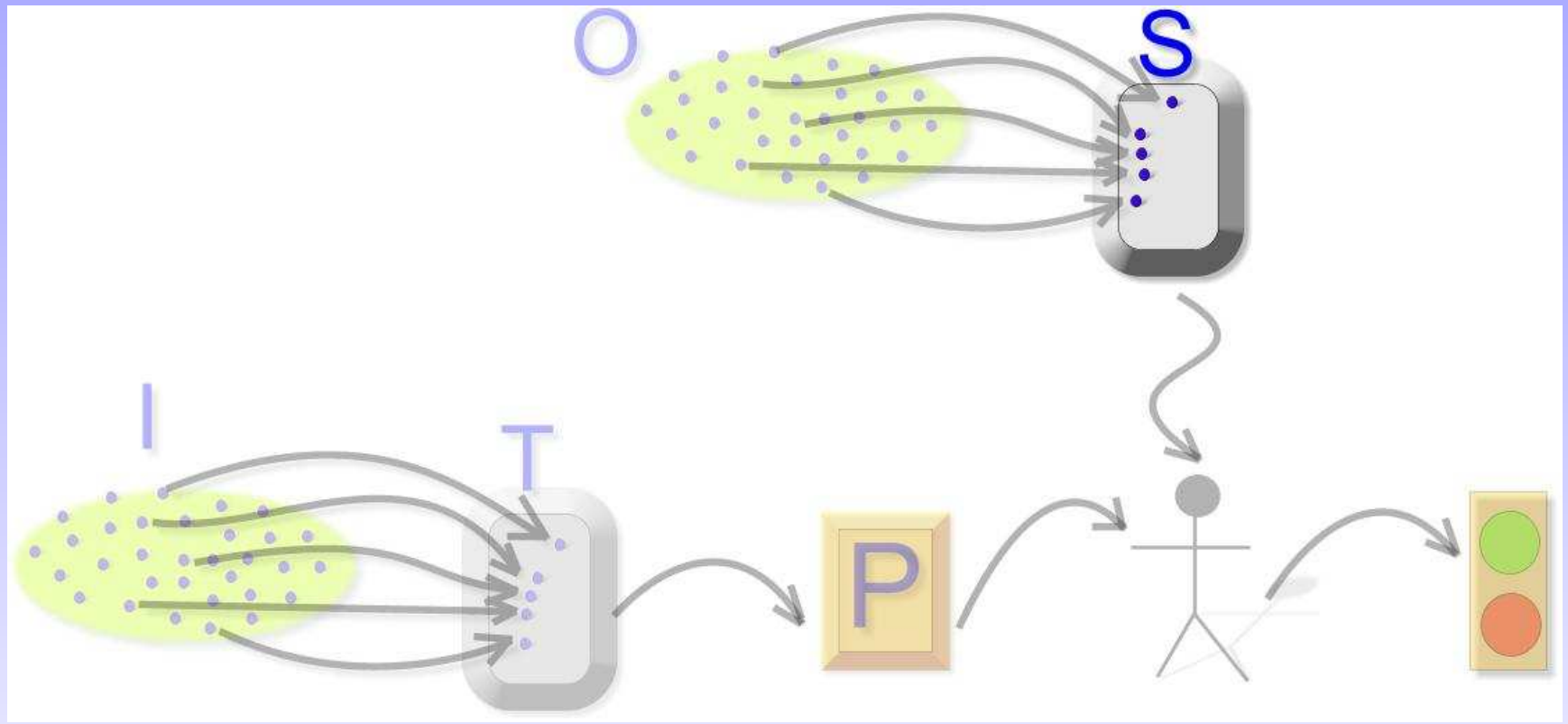
Dados de teste

Subconjunto do domínio de entrada.



Resultados esperados

Subconjunto correspondente do domínio de saída.

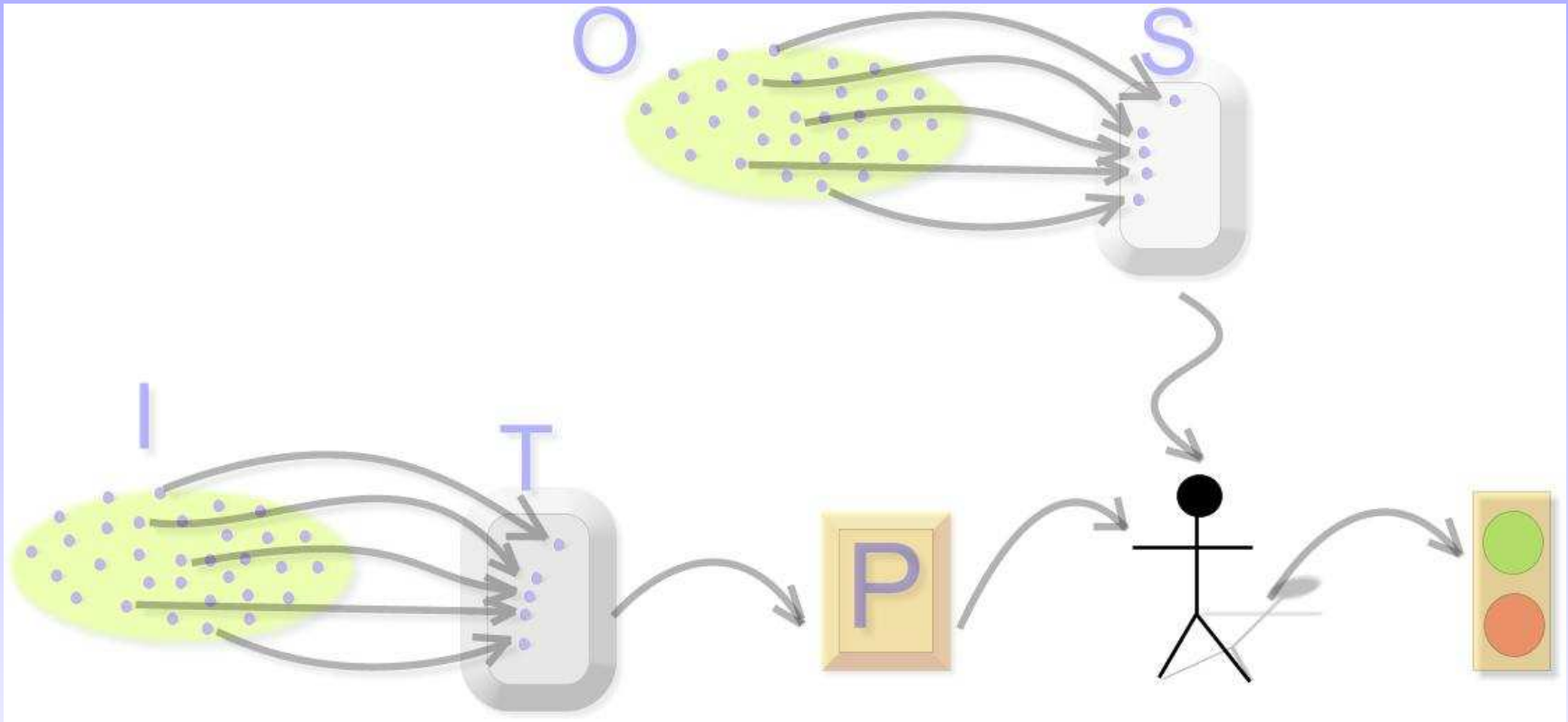


Casos de teste – exemplo

- Um caso de teste é um par: ⟨dado de entrada, resultado esperado⟩
- T = {-3, 3, 4}
- S = {`"Erro"`, 6, 24}
- Conj teste = {⟨-3,`"Erro"`⟩, ⟨3,6⟩, ⟨4,24⟩}

Oráculo

Mecanismo que decide sobre a correção de uma execução.



Oráculos complicados

- Resultado esperado desconhecido
  - Qual a décima milésima casa decimal de Π?
- Resultado conhecido mas difícil de ser avaliado
  - Formatos não convencionais: imagens, som etc
- Grande quantidade de dados

Erro, defeito, falha

- Depende da “comunidade” envolvida
- Defeito (fault): um passo, processo ou definição de dados incorreto
- Erro (error): se caracteriza por um estado inconsistente ou inesperado
- Falha (failure): um comportamente que difere do comportamento esperado.

Defeito – exemplo

`last` deveria ser inicializada com o valor 1.

```
public static void bubbleSort(int[] x) {
    int n = x.length;
    for (int last = 0; last < n; last++) {
        for (int i = 0; i < n - last; i++) {
            if (x[i] > x[i+1]) {
                int aux = x[i];
                x[i] = x[i+1];
                x[i+1] = aux;
            }
        }
    }
}
```

Erro – exemplo

Conceito dinâmico. Após a execução da atribuição `last = 0` o programa está em um estado inconsistente

```
public static void bubbleSort(int[] x) {
    int n = x.length;
    for (int last = 0; last < n; last++) {
        for (int i = 0; i < n - last; i++) {
            if (x[i] > x[i+1]) {
                int aux = x[i];
                x[i] = x[i+1];
                x[i+1] = aux;
            }
        }
    }
}
```

Falha – exemplo

Ao executar `if (x[i] > x[i+1])` ocorre uma falha. (Nem sempre).

```
public static void bubbleSort(int[] x) {
    int n = x.length;
    for (int last = 0; last < n; last++) {
        for (int i = 0; i < n - last; i++) {
            if (x[i] > x[i+1]) {
                int aux = x[i];
                x[i] = x[i+1];
                x[i+1] = aux;
            }
        }
    }
}
```

Exemplo

```
1  public static int numZero (int[] x) {
2  // Funcao: se x == null lanca NullPointerException
3  // senao returna o número de ocorrências de 0 em x
4      int count = 0;
5      for (int i = 1; i < x.length; i++)
6      {
7          if (x[i] == 0)
8          {
9              count++;
10         }
11     }
12     return count;
13 }
```

Código fonte: `./src/defeito.java`

Exemplo

- Defeito: inicialização da variável $i = 1$
- Não falha: $x = [2, 7, 0]$
- Falha: $x = [0, 7, 2]$
- Estado dado pelo valor de: x, i, count e PC
- Erro: $x = [0, 7, 2], i = 1, count = 0$, PC = if
- Erro (???): $x = [2, 7, 0], i = 1, count = 0$, PC = if

Para uma falha ocorrer (RIP)

- O ponto do programa que contém um defeito deve ser executado (alcançabilidade).
- Após a execução deste ponto, o estado da execução deve ser incorreto (infecção).
- O estado infectado deve se propagar de modo a produzir uma saída incorreta (propagação).
- RIP (Reachability, Infection, Propagation).

Exercícios

```
1  public static int lastZero (int[] x) {
2  // Funcao: se x==null lanca NullPointerException
3  // senao returna o indic do ultimo 0 em x
4  // Returna -1 se 0 nao ocorre em x
5
6      for (int i = 0; i < x.length; i++) {
7          if (x[i] == 0) {
8              return i;
9          }
10     }
11     return -1;
12 }
13 // teste: x=[0, 1, 0]
14 // esperado = 2
```

Código fonte: `src/lastzero.java`

lastZero

- a) Identifique o defeito
- b) Se possível, identifique um caso de teste que não executa o defeito.
- c) Se possível, identifique um caso de teste que executa o defeito mas não resulta em um estado de erro.
- d) Se possível, identifique um caso de teste que resulta em um estado de erro mas não em uma falha.
- e) Para o caso de teste dado, identifique o primeiro estado de erro. Descreva o estado por completo.
- f) Corrija o defeito e verifique que o caso de teste

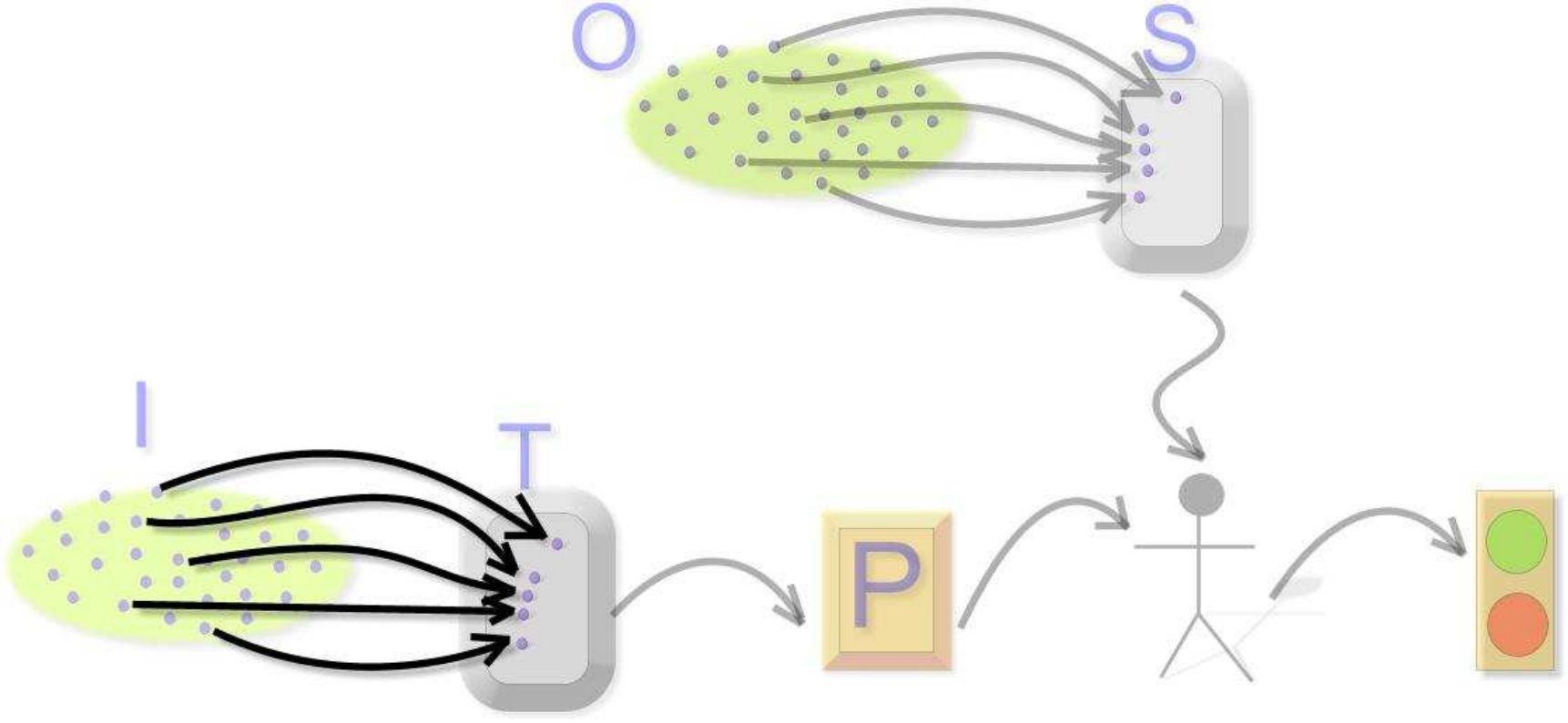
oddOrPos

```
1  public static int oddOrPos(int[] x) {
2  // Funcao: se x==null lanca NullPointerException
3  // senao returna o numero de elementos em x que
4  // sao impar ou positivo (ou ambos)
5      int count = 0;
6      for (int i = 0; i < x.length; i++) {
7          if (x[i]%2 == 1 || x[i] > 0) {
8              count++;
9          }
10     }
11     return count;
12 }
13 // test: x=[-3, -2, 0, 1, 4]
14 // Expected = 3
```

Código fonte: `src/oddorpos.java`

Critério/técnica de teste

Maneira de determinar quais dados de teste devem ser usados (de forma a maximizar a chance de revelar um defeito).



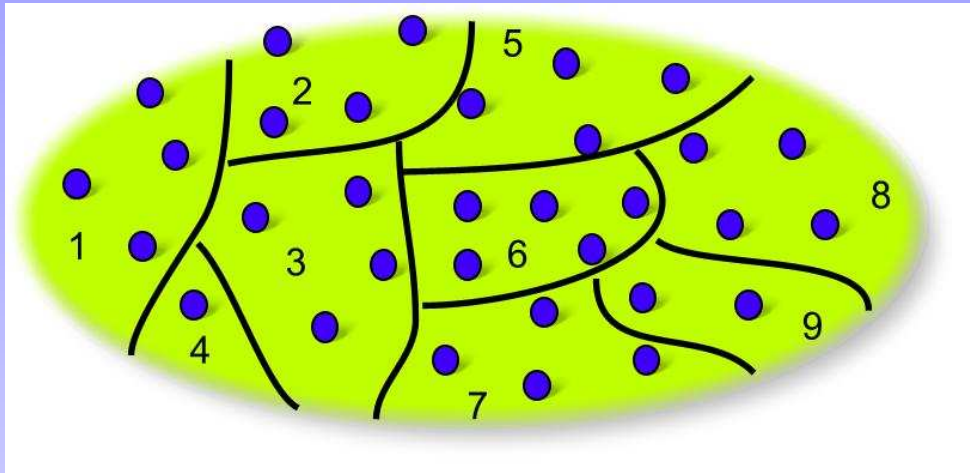
Critérios – exemplo

- ✓ Selecionar aleatoriamente 30 casos de teste
- ✓ Selecionar 1% dos elementos válidos do domíno e 10 elementos não válidos (exceções)
- ✓ Selecionar casos de teste que executem cada um dos comandos da implementação sendo testada

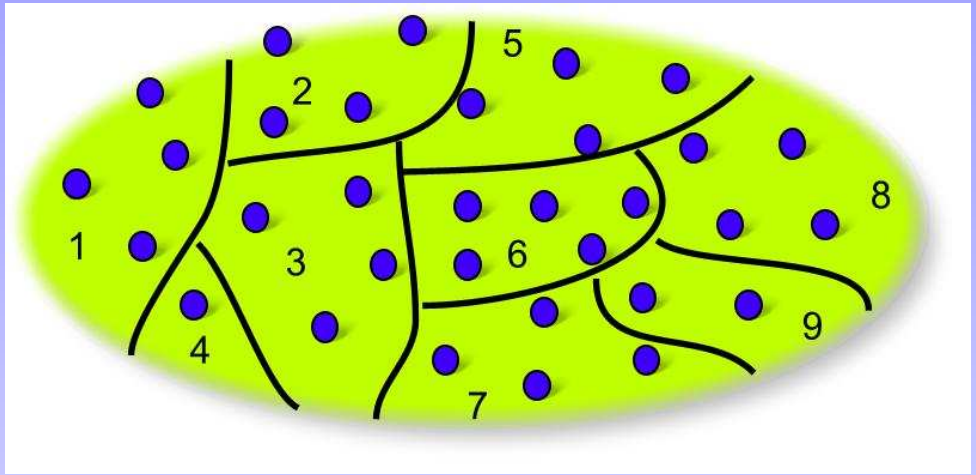
Critérios baseados em subdomínios

- Dividir o domínio de entrada em subdomínios
- De cada subdomínio selecionar n elementos (geralmente 1)
- Dessa forma tem-se uma cobertura do domínio como um todo
- Maximiza-se a chance de revelar defeitos
- Dependendo de como os subdomínios são definidos, garante-se que elementos críticos sejam selecionados

Subdomínios



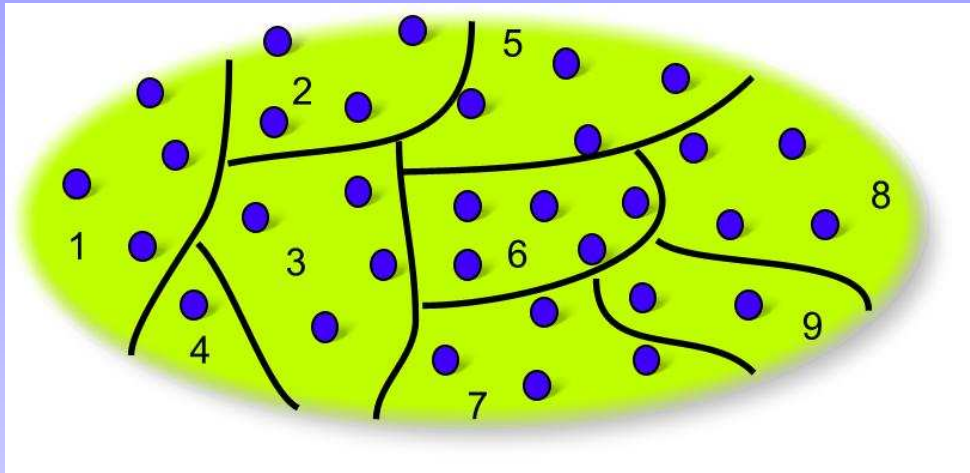
Subdomínios



$$p = 1 - \prod_{i=1}^{k} (1 - \frac{m_i}{d_i})^{n_i}$$

d_i é o tamanho do subdomínio i
m_i é o número de elementos que falham
n_i é número de elementos a serem testados no subdomínio

Subdomínios



$$p = 1 - \prod_{i=1}^{k}(1 - \frac{m_i}{d_i})$$

d_i é o tamanho do subdomínio i
m_i é o número de elementos que falham
1 é número de elementos a serem testados no subdomínio

Probabilidade em revelar defeitos

- Simplificando: $n_i = 1$
- $$p = 1 - \prod_{i=1}^{k}(1 - \frac{m_i}{d_i})$$
- Para garantir que um defeito será revelado devemos ter $\frac{m_i}{d_i} = 1$ para algum i
- Um critério de teste que gere tais subdomínios é **confiável**

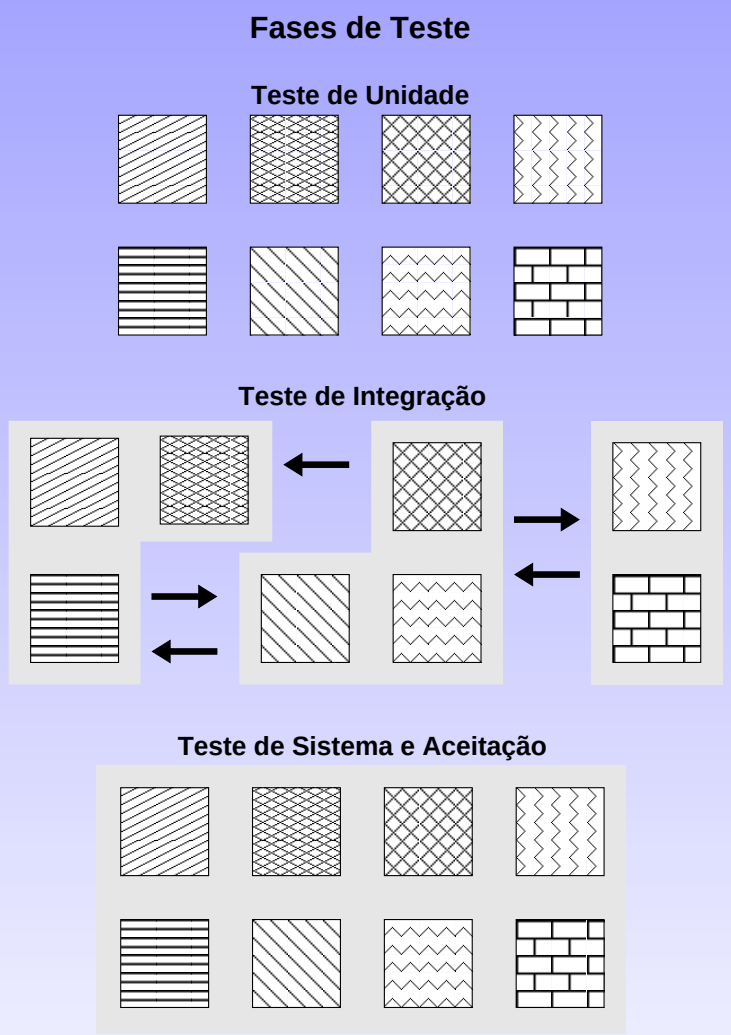
Como gerar subdomínios

- Técnica funcional
  - Apenas a especificação do problema (domínio, função) serve para definir os subdomínios
  - Partição em classes de equivalência, análise de valor limite, grafo cause-efeito
- Técnica estrutural
  - Elementos do código são usados como **requisitos de teste** que dividem o domínio de entrada
- Técnica baseada em defeitos
  - Defeitos típicos ou específicos são usados para dividir o domínio

Fases de teste (1)

- A atividade de teste também é dividida em fases, conforme outras atividades de Engenharia de Software.
- Objetivo é reduzir a complexidade dos testes.
- Conceito de “dividir e conquistar”.
- Começar testar a menor unidade executável até atingir o programa como um todo.

Fases de Teste (2)



Teste Procedimental	Teste Orientado a Objeto
Procedimento ou Sub−rotina	Método
Dois ou mais procedimentos Subsistema	Classe Cluster Componentes Subsistema
Todo Sistema	Todo Sistema

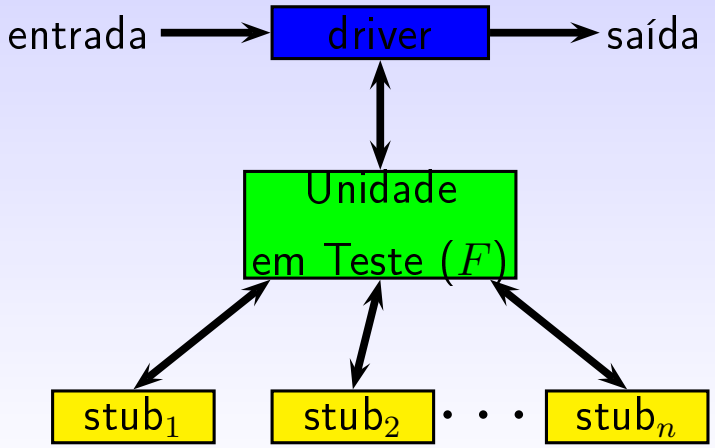
Principais Fases de Teste

Teste de Unidade

- Objetivo é identificar erros de lógica e de programação na menor unidade de programação.
- Diferentes linguagens possuem unidades diferentes.
  - Pascal e C possuem procedimentos ou funções.
  - Java e C++ possuem métodos (ou classes?).
  - Basic e COBOL (o que seriam unidades?).
- Como testar uma unidade que depende de outra para ser executada?
- Como testar uma unidade que precisa receber dados de outra unidade para ser executada?

Driver e *Stub*

- Para auxiliar no teste de unidade, em geral, são necessários *drivers* e *stubs*.
- O *driver* é responsável por fornecer para uma dada unidade os dados necessários para ela ser executada e, posteriormente, apresentar os resultados ao testador.
- O *stub* serve para simular o comportamento de uma unidade que ainda não foi desenvolvida, mas da qual a unidade em teste depende.



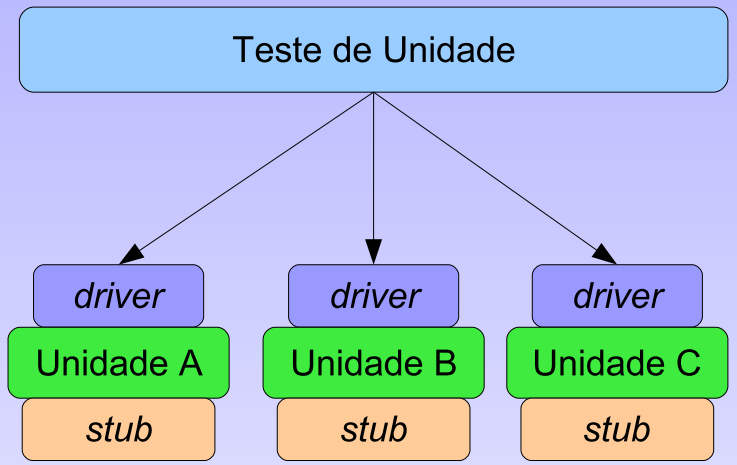
Drivers e Stubs

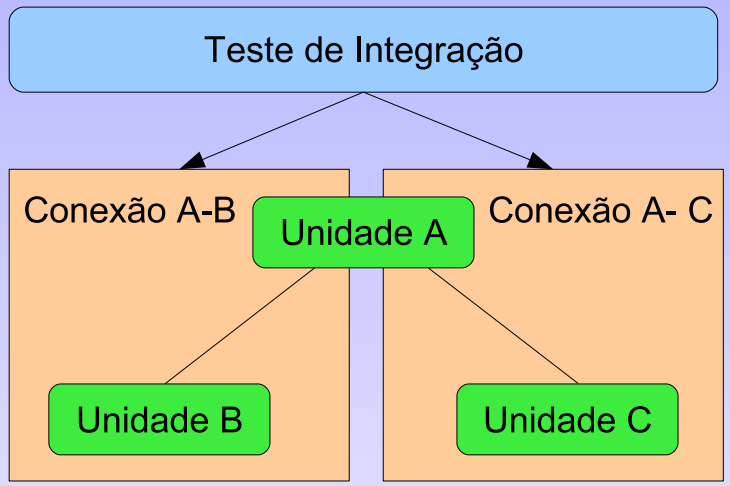
Teste de Integração (1)

- Objetivo é verificar se as unidades testadas individualmente se comunicam como desejado.
- Por que testar a integração entre unidades se as mesmas, em isolado, funcionam corretamente?

Teste de Integração (2)

- ✓ Dados podem se perder na interface das unidades.
- ✓ Variáveis globais podem sofrer alterações indesejadas.





Teste de Unidade X Teste de Integração.

Teste de Sistema

- Objetivo é verificar se o programa em si interage corretamente com o sistema para o qual foi projetado. Isso inclui, por exemplo, o SO, banco de dados, hardware, manual do usuário, treinamento, etc.
- Corresponde a um teste de integração de mais alto nível.
- Inclui teste de funcionalidade, usabilidade, segurança, confiabilidade, disponibilidade, performance, backup/restauração, portabilidade, entre outros (Norma ISO-IEC-9126 para mais informações)

Teste de Aceitação

- Objetivo é verificar se o programa desenvolvido atende as exigências do usuário.

Teste de Regressão

- Mesmo após liberado o software precisa ser testado
- A cada manutenção é preciso verificar se o software mantém suas características
- Se nenhum efeito colateral foi introduzido
- Técnicas de teste de regressão reutilizam subconjuntos do conjunto de teste existente

Limitações do Teste (1)

Observe o exemplo abaixo:

```
1  int blech(int j) {
2      j = j - 1;   // deveria ser j = j + 1
3      j = j / 30000;
4      return j;
5  }
```

Código fonte: `./src/blech.java`

- ✓ Considerando o tipo inteiro com 16 bits (2 bytes) – o menor valor possível seria -32.768 e o maior seria 32.767, resultando em 65.536 valores diferentes possíveis.
- ✓ Haverá tempo suficiente para se criar 65.536 casos de teste? E se os programas forem maiores? Quantos casos de teste serão necessários?

Limitações do Teste (2)

```
1  int blech(int j) {
2      j = j - 1;   // deveria ser j = j + 1
3      j = j / 30000;
4      return j;
5  }
```

Código fonte: `./src/blech.java`

Quais valores escolher?

Entrada (j)	Saída Esperada	Saída Obtida
1	0	0
42	0	0
40000	1	1
-64000	-2	-2

Quais valores de entrada revelam o erro no programa acima?

Limitações do Teste (3)

- Os casos de testes anteriores não revelam o erro.
- Somente quatro valores do intervalo de entrada válido revelam o erro:
- Os valores abaixo revelam o erro:

Entrada (j)	Saída Esperada	Saída Obtida
-30000	0	-1
-29999	0	-1
30000	1	0
29999	1	0

- Qual a chance desses valores serem selecionados???

Automatização

- A atividade de teste é cara e consome muito tempo.
- Automatizar significa:
  - Diminuir custos
  - Diminuir erros
  - Facilitar teste de regressão
- Atividades intelectuais $\times$ atividades braçais.
- Suporte a técnicas de teste
- Geração de dados de teste
- Oráculos