



**PCS2408**

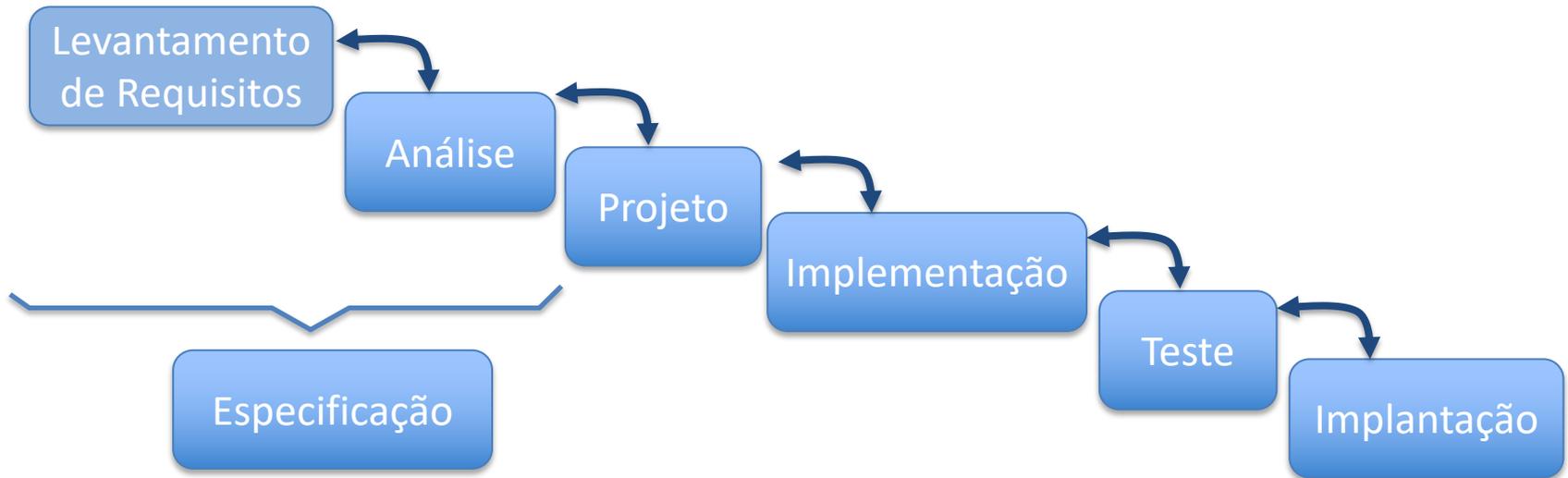
# Fundamentos de Engenharia de Software

**Aula 25**

# Processos de Software

- é um conjunto de atividades e resultados associados que produzem um produto de software
  - então, um processo de software se dá pela estruturação de um conjunto de atividades que resultam num produto de software
  - deve contribuir para redução de custos, aumento da qualidade e melhora da produtividade

# Etapas de Processo de Software

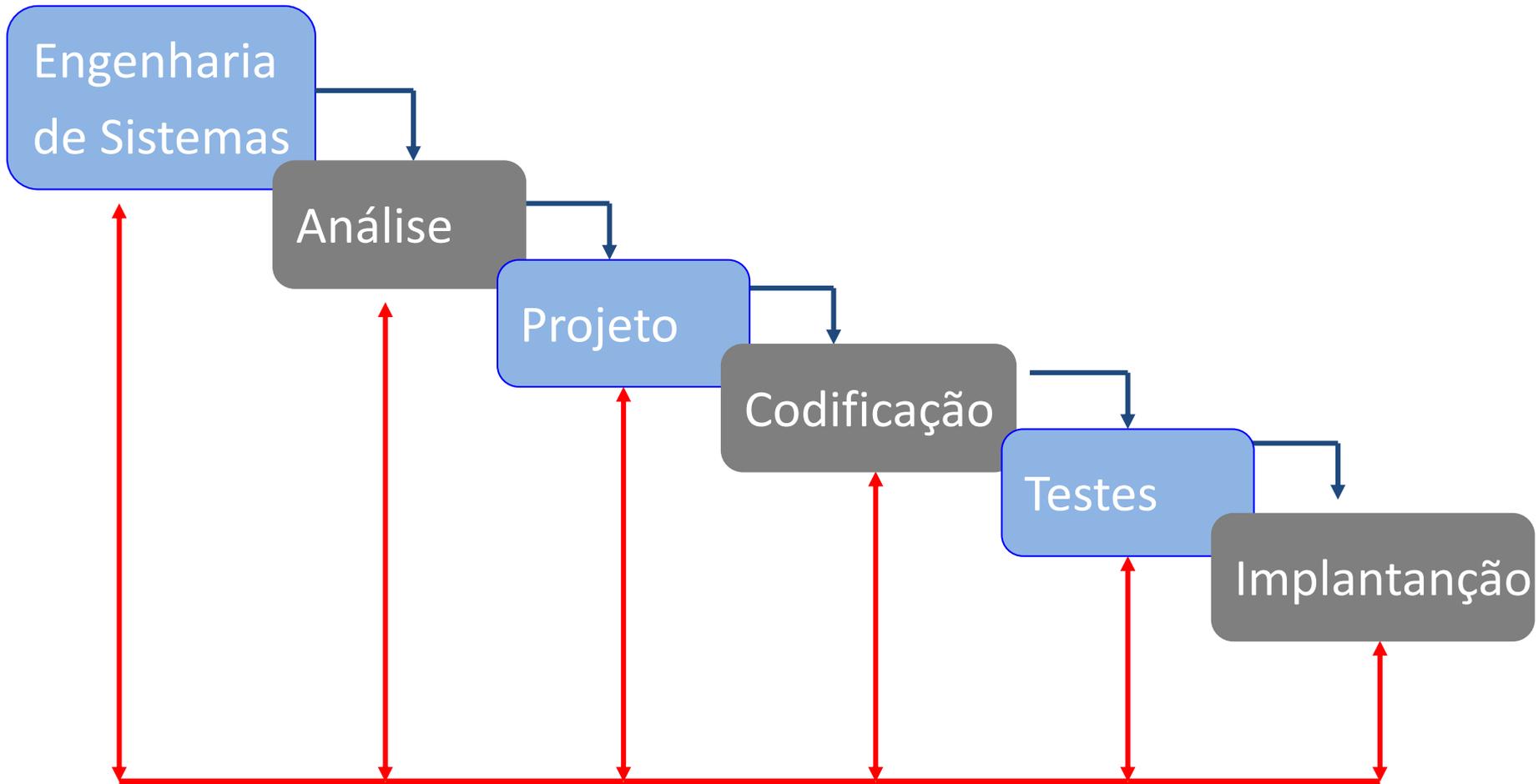


- **Um processo contém diversas atividades divididas em fases.**
- o encadeamento específico dessas fases dá-se o nome de Modelo Prescritivos ou não Prescritivos de Processo

# Modelos Prescritivos

- seguem uma forma mais definida e sistemática de se construir software.
- surgiram para estruturar atividades de um processo
- prescrevem um conjunto de elementos e como esses elementos se inter-relacionam
- Tipos de Modelos Prescritivos:
  - Cascata
  - Modelo Incremental
  - Modelo Evolucionário

# Ciclo de Vida Clássico ou Cascata



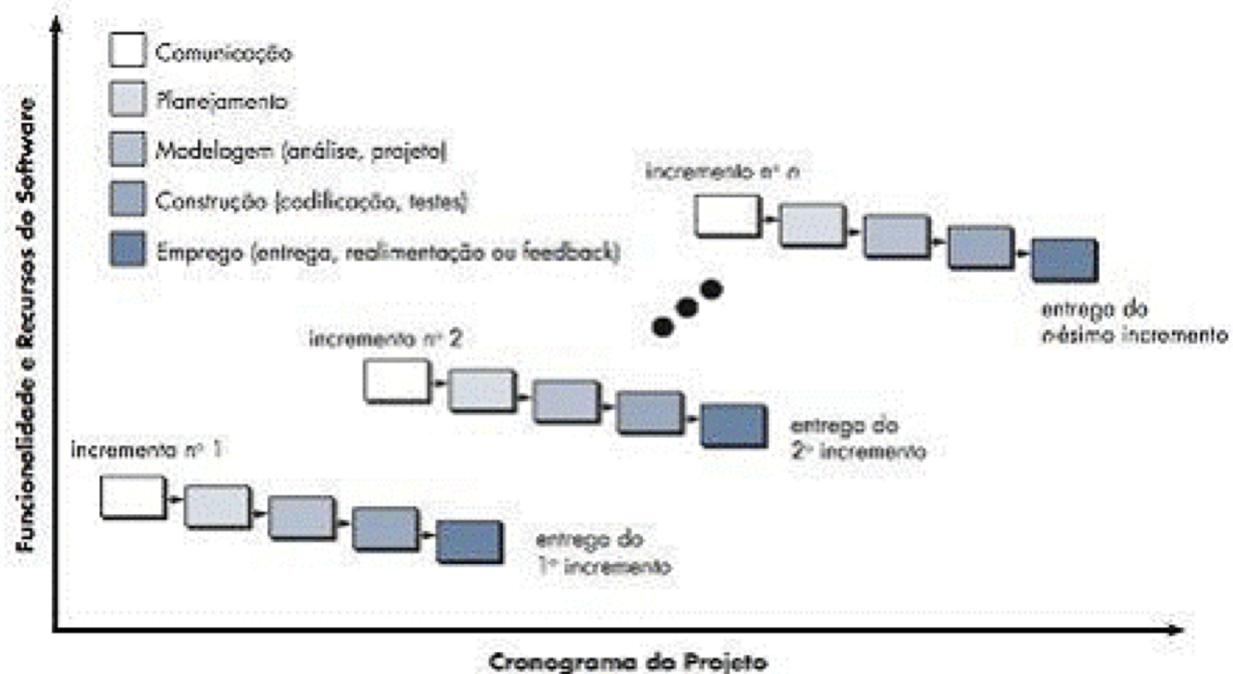
# Problemas

- Projetos reais raramente seguem o fluxo sequencial – ocorre alguma iteração
- O cliente tem dificuldade de declarar todas as exigências – há dificuldade em acomodar incertezas de início de projeto no modelo em cascata
- Uma versão de programa só estará disponível em fase adiantada do cronograma.

# Modelo Incremental

- baseia-se na entrega rápida de um conjunto funcional para o usuário, de forma que esse conjunto possa ser efetivamente utilizado pelo usuário – entrega parcial
  - precisa-se conhecer os requisitos do produto completo, ou seja os requisitos são razoavelmente bem definidos e compreendidos.
  - entregas sucessivas são feitas, culminando no produto completo que atende os requisitos identificados e acordados – o processo se desenvolve de forma incremental – cada incremento: nova funcionalidade é adicionada
  - processo incremental combina elementos de fluxos lineares e paralelos

# Modelo Incremental

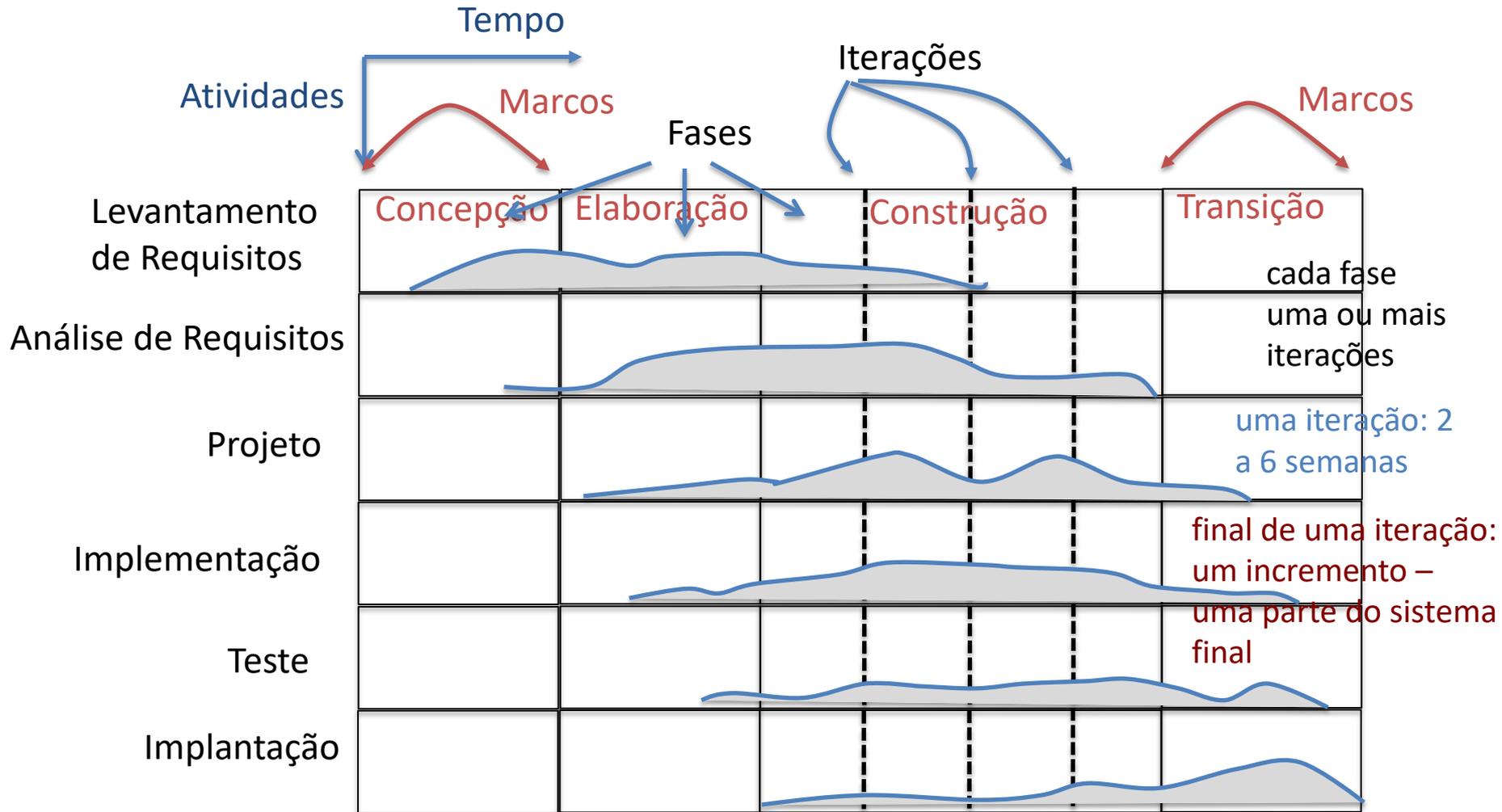


1º Incremento – parte essencial do protótipo, ou seja requisitos básicos que devem ser atendidos para que o software entre em operação

1º incremento entregue: cliente usa e fornece feedback que são analisados para o planejamento do 2º incremento e assim sucessivamente.

- o modelo de processo incremental entrega um produto operacional a cada incremento, ou seja um produto pronto para ser utilizado pelo cliente. Este é um produto que deve ter sido testado e ter a qualidade.
- o número de pessoas (profissionais ou stakeholders envolvidos) podem aumentar nas entregas posteriores, ou seja, conforme a complexidade do produto aumente. Espera-se que primeiras entregas sejam produtos mais simples –compreende a parte mais básica. Entregas posteriores compõem funcionalidade mais complexas, mas que dependem das primeiras para funcionarem.
- vantagem: melhor administração de risco – pode-se depender da entrega de um hardware específico que só estará disponível numa data posterior. Então, pode-se planejar que a parte do software que dependerá dos dados vindos desse hardware para um incremento futuro quando esse dispositivo já estiver disponível para uso.

# Estrutura geral de Processo Incremental e Iterativo



Processo Unificado

cada fase está associada a um artefato  
cada fase é concluída com um marco, que é o  
ponto onde decisões são tomadas e revistas.

# Fases do Processo Unificado

- concepção:
  - idéia geral e escopo do são definidos. Planejamento inicial do desenvolvimento e marcos entre as fases são estabelecidos.
- elaboração:
  - requisitos do sistema são definidos e priorizados sua ordenação. Iterações para a fase de Construção são definidas.
- construção:
  - fase de maior iterações incrementais. Nesta fase partes do produto podem ser entregues, o que envolve elaboração de manual do usuário.
- Transição:
  - usuários são treinados para interagir com o sistema. Envolve a aceitação do usuário. Avaliação de gastos. Inicia Manutenção.

# Modelo Evolucionário

- existem categorias de negócio em que a mudança de requisitos é constante, ou seja pode e ocorre durante o desenvolvimento de um produto de software.
- Modelo Evolucionário se caracteriza por um processo não linear – o planejamento do produto deve ser reavaliado durante o desenvolvimento do produto e significa que o próprio produto pode ser alterado em relação a sua concepção inicial.
  - ideal para produtos que evoluem ao longo do tempo.
- Exemplo:
  - Modelo Espiral

# Modelos não Prescritivos

- modelos ágeis
  - scrum
  - EXtreme Programming (XP)

# Scrum

- processo iterativo e incremental
- divide um projeto em pequenas partes (SPRINTS) executado por equipes pequenas
  - **dividir para conquistar**
  - complexidade:
    - produto a ser construído dividido em subprodutos independentes que depois serão integrados.
    - cada subproduto é um entregável que quando finalizado é entregue ao cliente, que o utiliza e fornece feedbacks.
    - cada subproduto pode ser construído paralelamente a outro, de acordo com o tamanho da equipe disponível.
  - equipe
    - dividida em pequenos grupos, cada um responsável por um subproduto
    - mais fácil a comunicação e controle das atividades de equipes pequenas (3 a 9 pessoas)

- Backlog do produto (Product backlog):
  - identificado o que precisa ser construído é elaborada uma lista de tudo que precisa ser feito no ciclo de vida do projeto.
- O produto backlog é dividido em partes ou itens que podem ser feitos e entregues de forma independente.
- Regra: em geral 80% do valor do produto para o usuário final está em 20% das funcionalidades do produto!
- Os itens (backlogs menores) ordenados de acordo com seu valor para o negócio – assegura que as principais funcionalidades são feitas e entregues primeiro
  - vantagem: maior parte das funcionalidades (80%) não são muito necessárias - se atrasos ou outros problemas acontecerem, o impacto é pequeno!

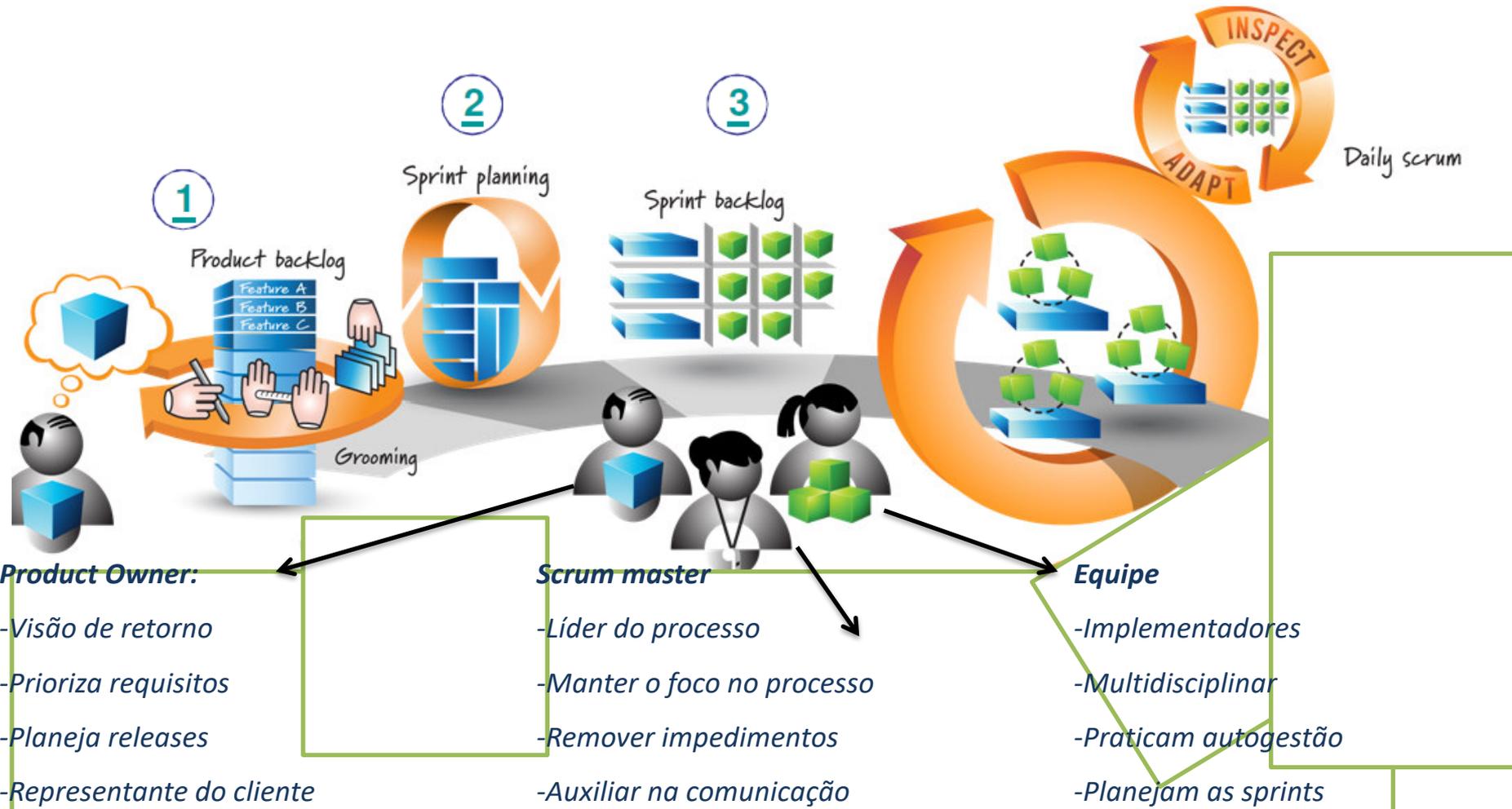
# Estrutura do SCRUM

reunião de 2 horas no máximo que define o que será feito na spring. Se repete a cada semana de duração da sprint

cerca de 15 minutos – verifica o que foi feito no dia anterior, o que deve ser feito no dia e problemas que impactam a finalização da sprint



# Scrum - Papeis



# Teste de Software

# Fundamentos de Teste de Software

- Objetivo:
  - encontrar problemas (erros)
  - os requisitos estabelecem o que é um erro – falha que faz com que o software não se comporte como esperado.

# Definições

- erro (*error*):
  - diferença entre o resultado de uma computação e o resultado correto ou esperado.
- defeito (*fault*):
  - linha de código, bloco ou conjunto de dados incorretos que provocam um erro observado.
- falha (*failure*):
  - não funcionamento do software, possivelmente provocado por um defeito, mas com outras possíveis causas (ex. falha de leitura, comunicação, etc.).
- engano (*mistake*):
  - erro humano, é a ação que produz ou produziu um defeito no software

# Depuração

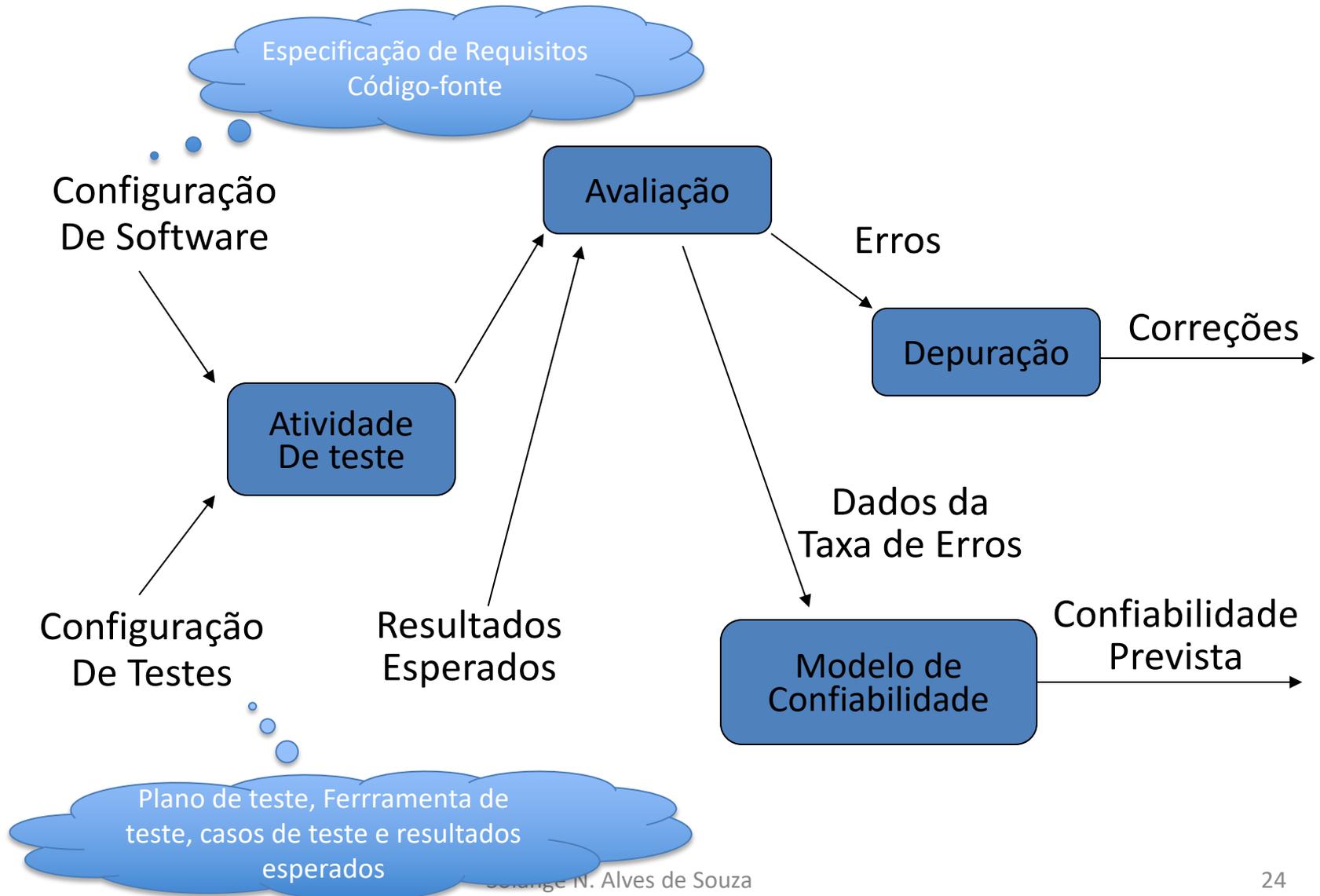
- busca a causa do erro, ou seja, o defeito oculto que está causando o erro.
- o fato de saber que não funciona, não significa saber em que linha de código está o erro,
- O processo de depuração é a parte mais imprevisível de todo o processo de desenvolvimento.
  - Pode demorar 1 hora, 1 semana, 1 mês para ser diagnosticado e corrigido.

# Categorias de testes

- Verificação:
  - software construído de acordo com o que foi especificado – atende aos requisitos
  - “Estamos fazendo a coisa do jeito certo?”
- Validação:
  - software construído atende às verdadeiras necessidades dos interessados – os requisitos foram corretamente entendidos – atendem realmente às necessidades?
  - “Estamos fazendo a coisa certa?)
- Teste:
  - atividade que permite realizar a verificação e validação do software.

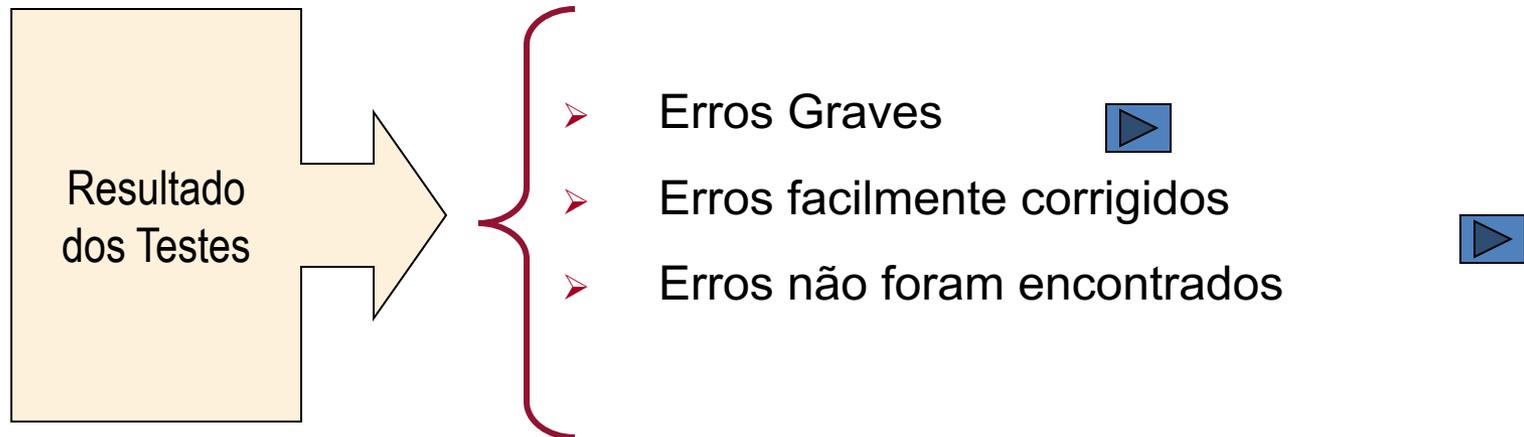
- Área de Teste:
  - definir conjuntos finitos e exequíveis de testes que, mesmo não garantindo que o software esteja livre de defeitos, consigam localizar os mais prováveis, permitindo sua eliminação.
- Lei de Murphy
  - Se alguma coisa pode sair errado, sairá (no pior momento possível)
  - Se tudo parece estar indo bem é porque você não olhou direito
  - A natureza sempre está a favor da falha oculta.

# Fluxo de Informação de Teste



- Os testes são realizados e todos os resultados obtidos são avaliados: os resultados obtidos são comparados com os resultados esperados.
- Dados errôneos encontrados: inicia-se a depuração.

Os resultados de teste reunidos e avaliados, dão uma indicação qualitativa da qualidade e confiabilidade do software.



## ■ Erros Graves:

- ☞ Se exigirem modificação de projeto e forem encontrados com regularidade indicam: **qualidade e confiabilidade do software suspeitos.**
- ☞ Testes adicionais são necessários.

- Erros facilmente corrigíveis:



- Qualidade e confiabilidade aceitáveis ou,
- Os testes são inadequados para detectar erros graves.

- Não foram encontrados erros:

- Atividade de teste não foi bem elaborada.
- Erros serão descobertos pelo usuário.
- Custo da correção: 60 a 100 vezes maior que fase de desenvolvimento.

# Projeto de Casos de Teste

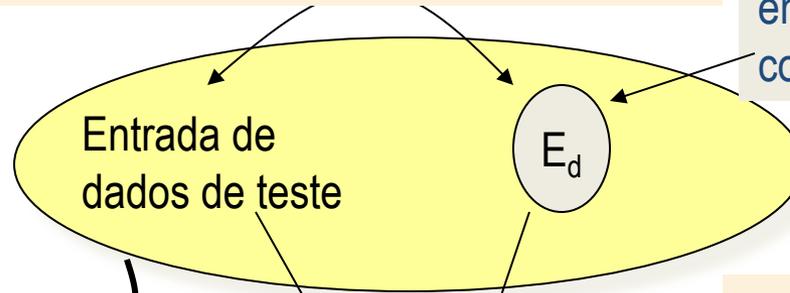
- Teste de Caixa Preta
  - Conhecendo-se a função específica que um produto deve executar, testes podem ser realizados para demonstrar que cada função é totalmente operacional.
- Teste de Caixa Branca
  - Conhecendo-se o funcionamento interno de um produto, testes podem ser realizados para garantir que “todas as engrenagens se encaixam”.

# Teste de Caixa Preta

# TESTE DE CAIXA PRETA

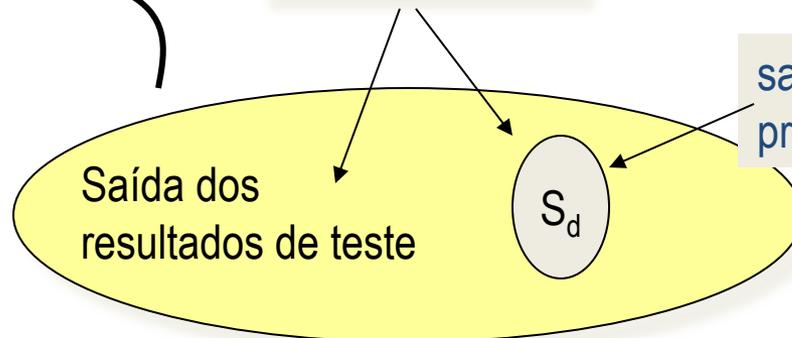
difícil checar erros devido a funcionalidade e erros devido a utilização de dados errados

entradas que provocam comportamento anômalo



preciso usar experiência e conhecimento do domínio da aplicação

testam a funcionalidade a partir do estudo das entradas e saídas



saídas que revelam a presença de defeitos

- Trata-se de uma abordagem complementar que tem a probabilidade de descobrir uma classe de erros diferente daquela dos métodos de caixa branca.
- O teste de caixa preta tende a ser aplicado para integração de módulos ou componentes.
- **aplicado a funções ou a objetos**

# Testes de Caixa Preta tentam descobrir erros nas seguintes categorias:

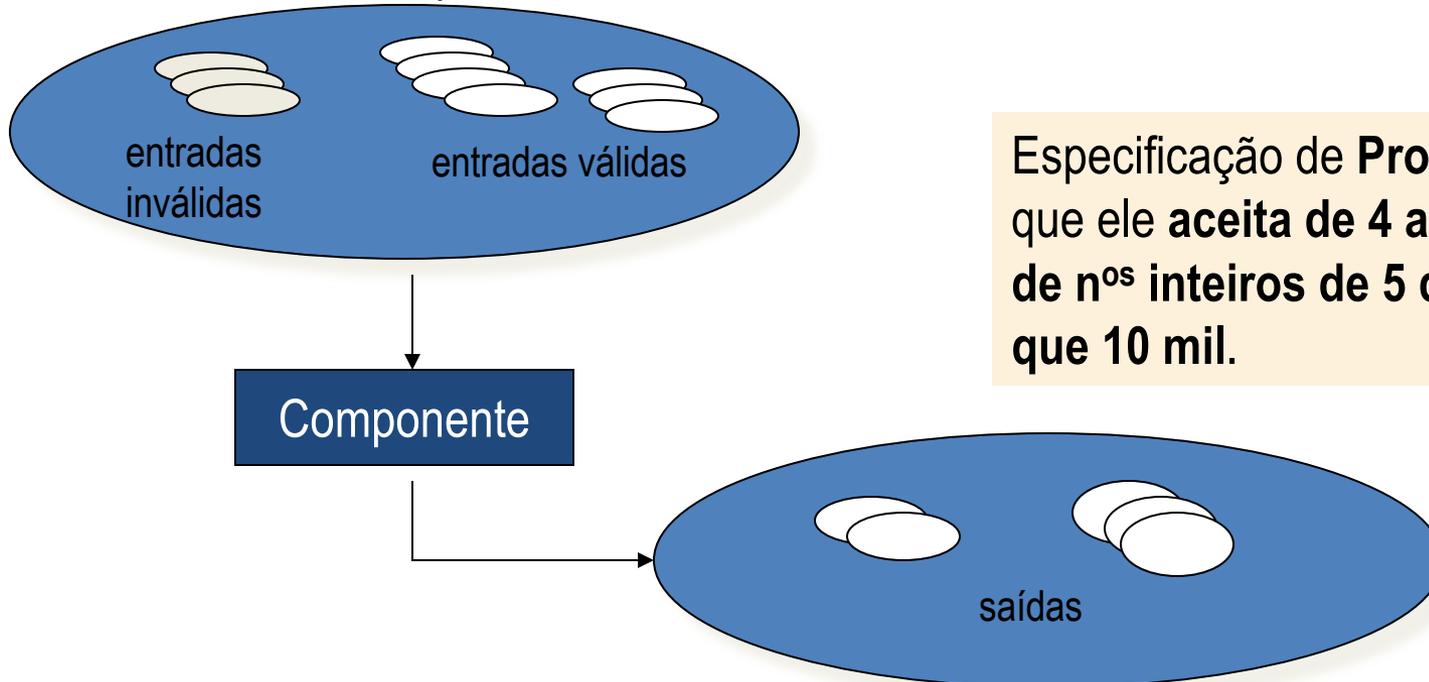
- funções incorretas ou ausentes;
- erros de interface;
- erros nas estruturas de dados ou no acesso a bancos de dados externos;
- erros de desempenho;
- erros de iniciação e término de partes ou de todo o sistema.

# Testes de caixa Preta

- Partição de Equivalência

# Partição de Equivalência

- conjuntos de dados onde todos os membros do conjunto devem ser processados de maneira equivalente.
- divide a entrada de um programa em classes de dados a partir das quais os casos de teste podem ser derivados.



Especificação de **Programa** declara que ele **aceita de 4 a 10 entradas de n<sup>os</sup> inteiros de 5 dígitos maiores que 10 mil.**

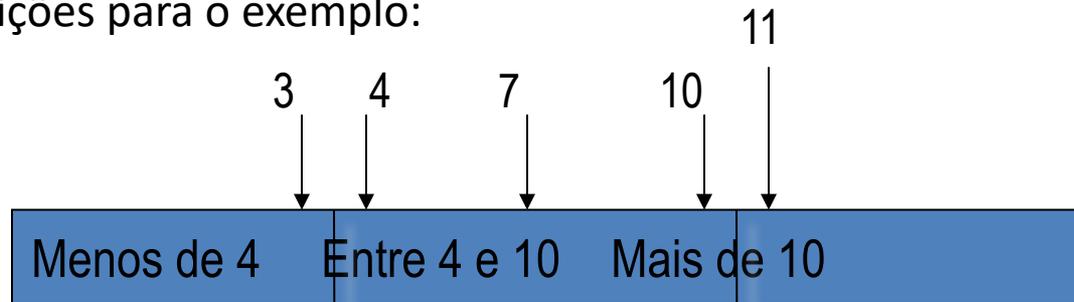


Especificação de Programa declara que ele aceita de 4 a 10 entradas de n<sup>os</sup> inteiros de 5 dígitos maiores que 10 mil.

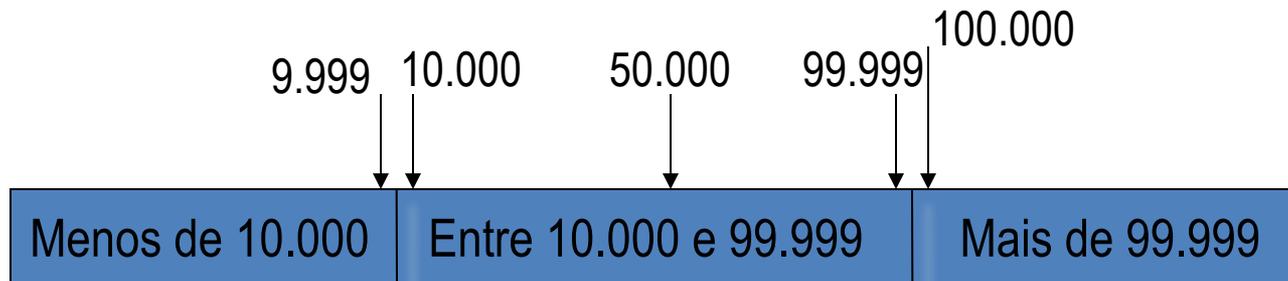
Diretriz:

☞ escolher casos de testes nos limites das partições e próximos ao ponto médio da partição

Duas partições para o exemplo:



Número de valores de entrada



Valores de entrada

Ex. Rotina de busca, que procura numa sequência de elementos um determinado valor de chave. Ela retorna a posição deste elemento na sequência

```
procedure Search (key:ELEM; T:SEQ of ELEM;  
    Found: BOOLEAN; L:ELEM_INDEX);
```

### **Pre-condition**

-- a sequência tem pelo menos um elemento  
 $T' \text{ FIRST} \leq T' \text{ LAST}$

### **Pos-condition**

-- o elemento é encontrado e é referenciado por  
(Found and  $T(L) = \text{Key}$ )

**or**

-- o elemento não está na sequência

(**not** Found **and**

**not** (**exists**  $i, T' \text{ FIRST} \geq i \leq T' \text{ LAST}, T(i) = \text{Key}$ ))

duas partições:

- 1- entradas em que a sequência contém o elemento chave
- 2 - entradas em que a sequência não contém o elemento chave

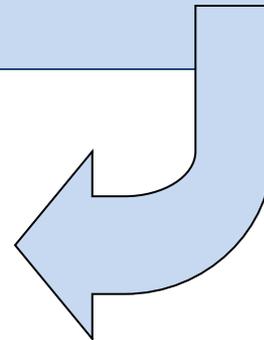
# Outras diretrizes para testes

1. Teste o software com sequências que tenham um único valor
  - o programa pode não trabalhar adequadamente com sequências que tenham um único elemento.
2. Utilize sequências de tamanhos diferentes, em diferentes testes
  - diminui chance de um defeito produzir uma saída correta.
3. Gere casos de teste de maneira que o primeiro, o médio e o último elemento da sequência sejam acessados
  - encontra problemas nos limites

mais duas partições:

3- sequência tem um único elemento

2 - sequência tem mais de um elemento



## Partições de Equivalência para a rotina de busca

Vetor		Elemento	
Um único elemento		Está na sequência	
Um único elemento		Não está na sequência	
Mais de um elemento		É o 1º da sequência	
Mais de um elemento		É o último da sequência	
Mais de um elemento		É o elemento médio na sequência	
Mais de um elemento		Não está na sequência	
Sequência de entrada		Chave (key)	Saídas (Found, L)
17		17	verdadeiro, 1
17		0	falso, ???
17, 29, 21, 23		17	verdadeiro, 1
41, 18, 9, 31, 31, 16, <b>45</b>		45	verdadeiro, 7
17, 18, 21, <b>23</b> , 29, 41, 38		23	verdadeiro, 4
21, 23, 29, 33, 38		25	falso, ??

conjunto de valores de entrada não é exaustivo

não incluídos os testes para verificar se qtde, ordem de entrada, tipo de dado dos parâmetros estava correta

# **Testes de Caixa Branca ou Testes de Estrutura**

# Testes de Caixa Branca

- É um método de projeto de casos de teste que **usa conhecimento da estrutura e da implementação para derivar casos de teste.**

# O engenheiro de software pode derivar os casos de teste que:

- garantam que todos os caminhos independentes dentro de um módulo ou método tenham sido exercitados pelo menos uma vez;
- exercitem todas as decisões lógicas para valores falsos ou verdadeiros;
- executem todos os laços em suas fronteiras e dentro de seus limites operacionais;
- exercitem as estruturas de dados internas para garantir a sua validade.

# Testes de Caminho

- ❑ testar cada caminho independente num procedimento

```
// função de busca binária que considera um vetor de objetos ordenados e uma  
// chave e retorna um objeto com 2 atributos, chamados index – o valor do índice  
// do vetor e Found – um boleano que indica se uma chave está ou não no vetor.
```

```
// A chave será -1 se o elemento não for encontrado.
```

# Implementação de uma rotina de busca binária em Java

Class BinSearch { 

```
public static void search (int key, int [ ] elemArray, Result r)
```

1 { int bottom = 0;  
int top = elemArray.length - 1;  
int mid;  
r. found = false; r.index = -1;

2 while (bottom <= top)

{ mid = (top + botttom) / 2;

3 if (elemArray [mid] == key)  
{ r.index = mid;

r. found = true;

return;

} // if

else

{

4 if (elemArray [mid] < key) bottom = mid + 1;

else top = mid + 1;

}

7 } // while

8 } // search

9 } //BinSearch

grafo de Fluxo para encontrar os caminhos independentes 

grafo de fluxo para essa rotina 

5

6

# Complexidade Ciclomática - CC

$$CC(G) = \text{Número (ramos)} - \text{Número (nós)} + 2 \quad (\text{McCabe, 1976})$$

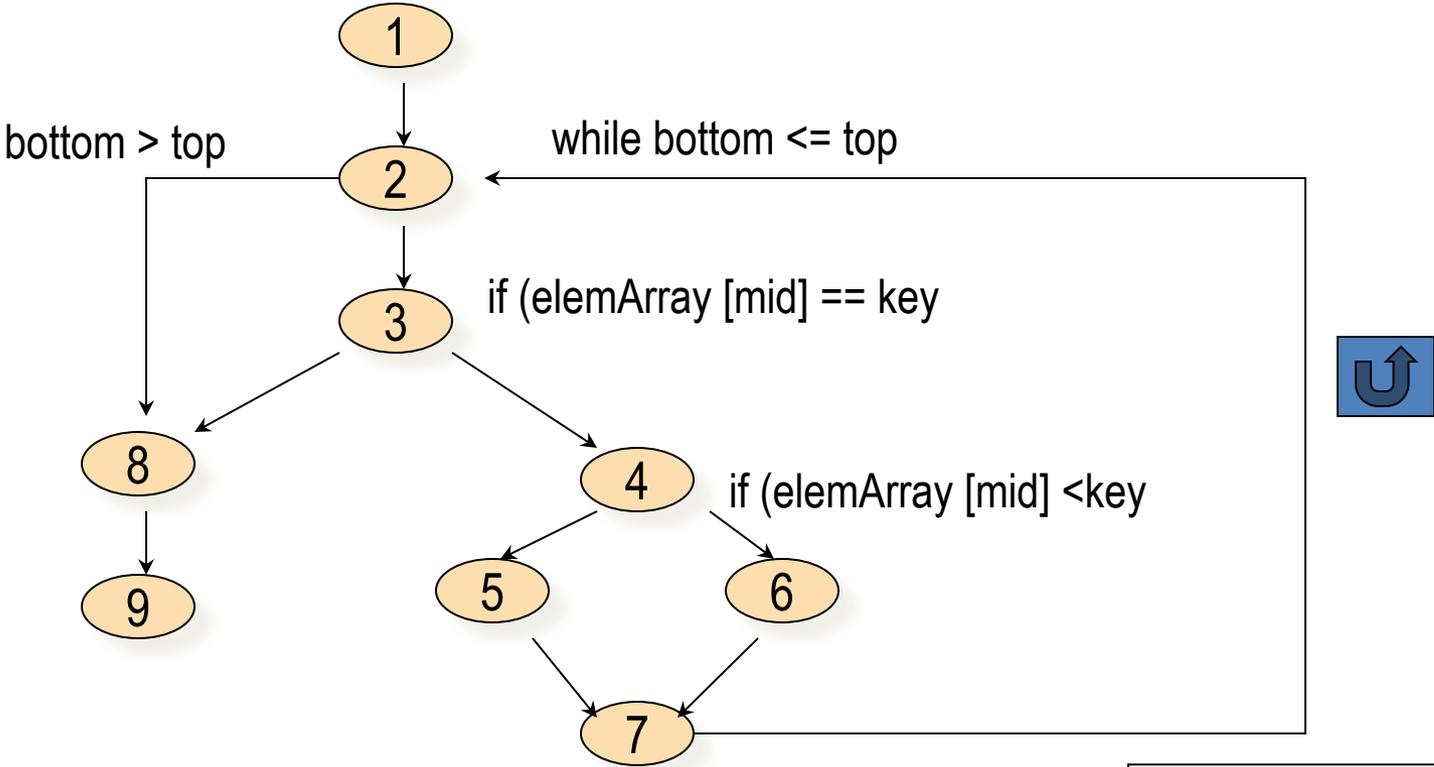
- o número de casos de teste requerido para testar todos os caminhos do programa é igual a complexidade ciclomática

# grafo de fluxo

- nós – representam decisões
- ramos – representam fluxo de controle
- declarações sequenciais (atribuições, chamadas de procedimentos e declarações de E/S) podem ser ignoradas na construção do fluxo.
- cada ramo de condição (if-then-else) é mostrado como um caminho independente.
- loops indicado por seta mostrando o retorno ao nó da condição do loop



# Grafo de Fluxo para rotina de busca binária



caminhos independentes – complexidade ciclomática

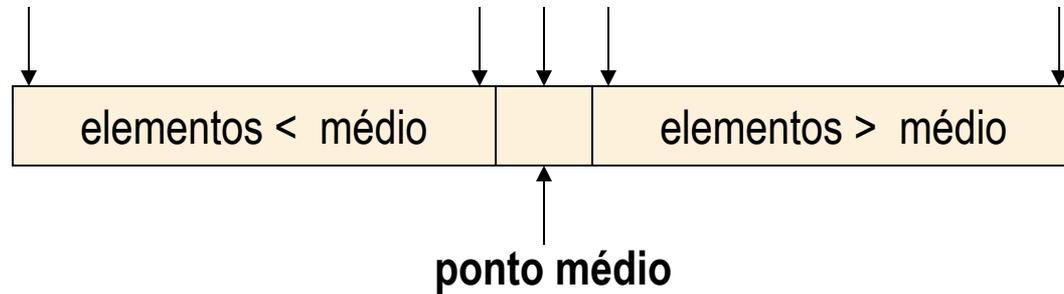
- 1, 2, 8, 9
- 1, 2, 3, 8, 9
- 1, 2, 3, 4, 5, 7, 2
- 1, 2, 3, 4, 6, 7, 2, 8, 9



- todas as declarações foram executadas pelo menos uma vez
- todos os ramos verdadeiro/falso foram exercitados

classe de equivalência de busca binária

Limites da classe de equivalência



casos de teste para a rotina de busca

Vetor de entrada	Chave (key)	Saídas (Found, L)
17	17	verdadeiro, 1
17	0	falso, ???
17, 21, 23, 29	17	verdadeiro, 1
9, 16, 18, 30, 31, 41, 45	45	verdadeiro, 7
17, 18, 21, 23, 29, 38, 41	23	verdadeiro, 4
7, 18, 21, 23, 29, 33, 38	21	verdadeiro, 3
12, 18, 21, 23, 32	23	verdadeiro, 4
21, 23, 29, 33, 38	25	falso, ??

# Testes de Estresse

- planejar testes que avaliem o comportamento do sistema em condições de sobrecarga – confiabilidade e desempenho
- Objetivo:
  - testar o comportamento da falha:
    - verificar se dados são corrompidos
    - se interrompe os serviços para o usuário