

---

# Engenharia de Software

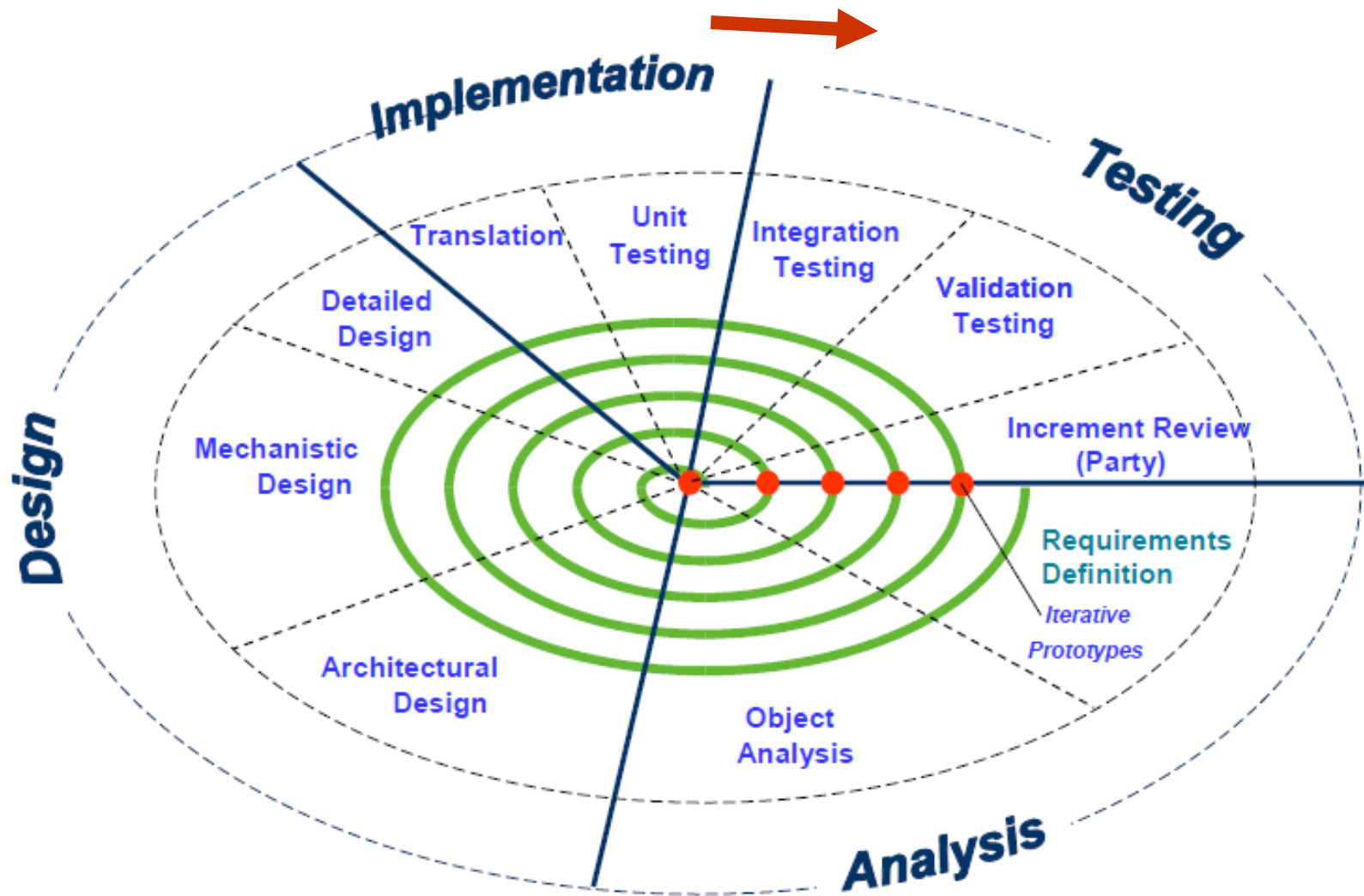
Verificação e Teste

**Luís Morgado**

Instituto Superior de Engenharia de Lisboa  
Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

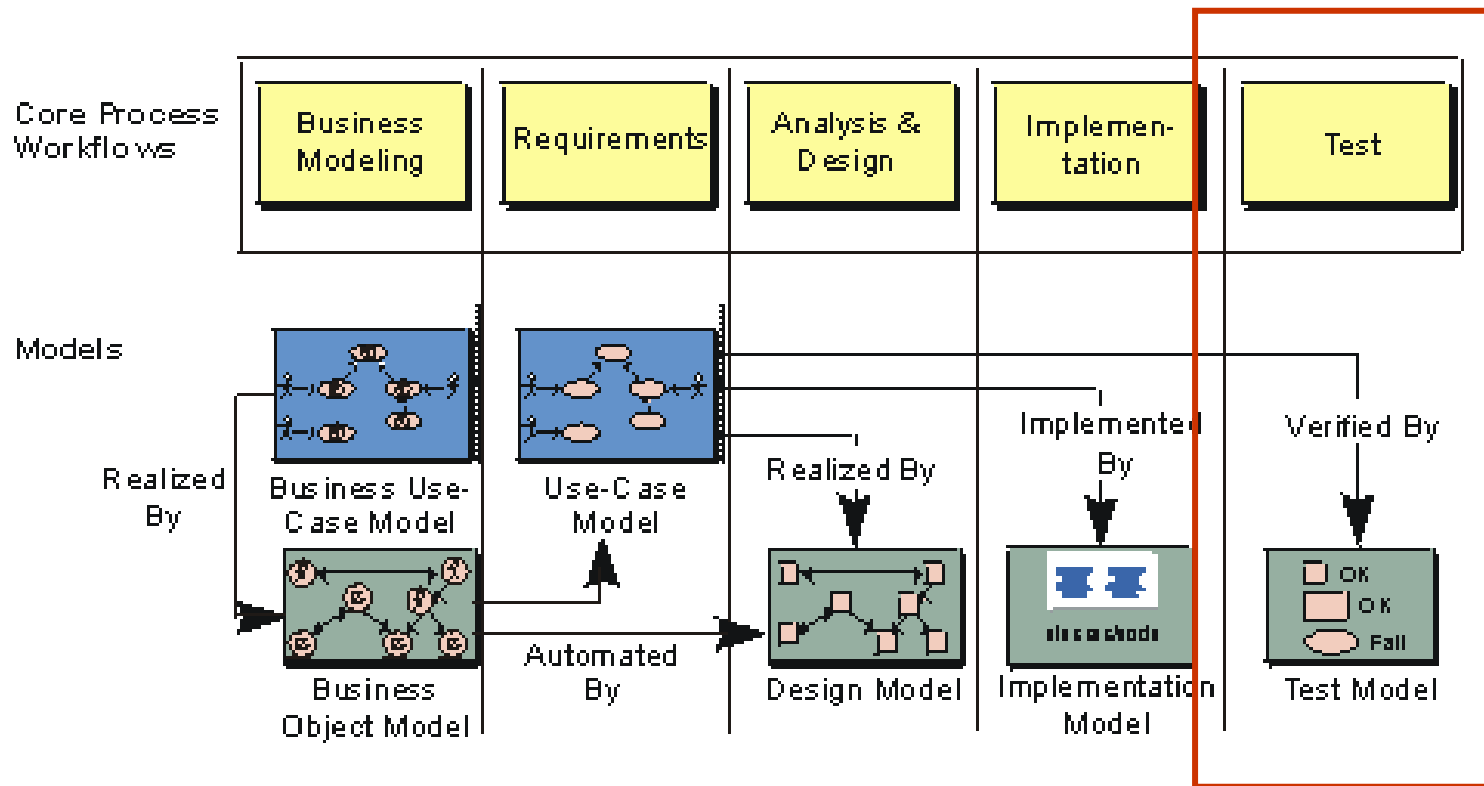
---

# Processo de desenvolvimento



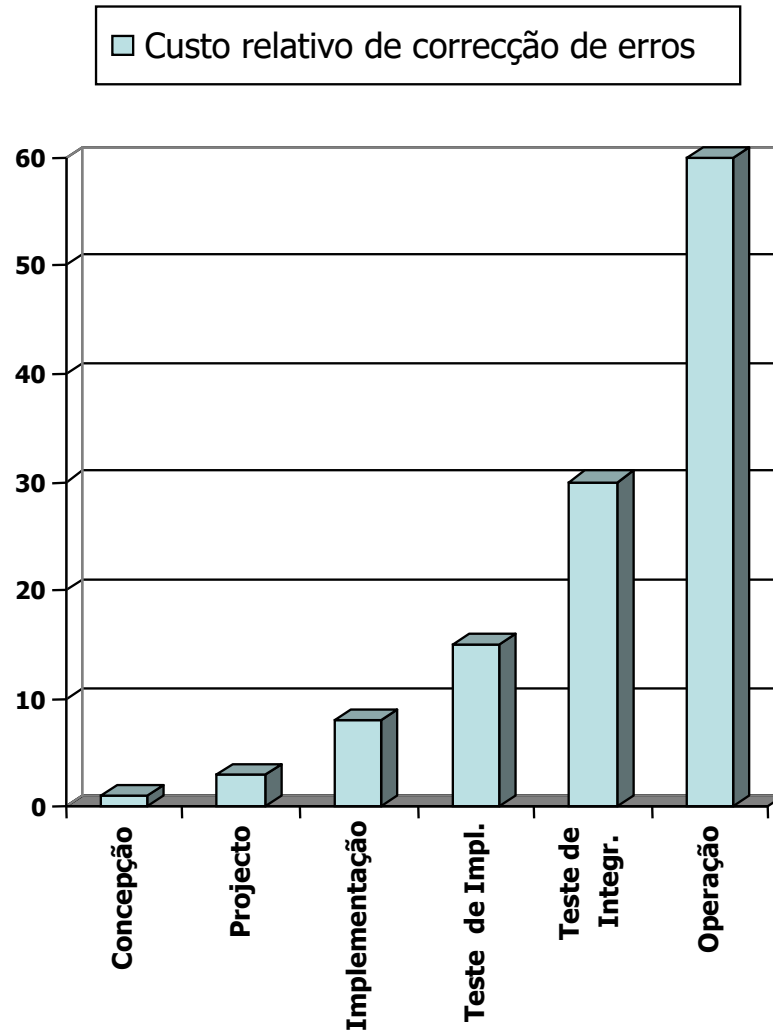
[Douglass, 2006]

# Teste de Software



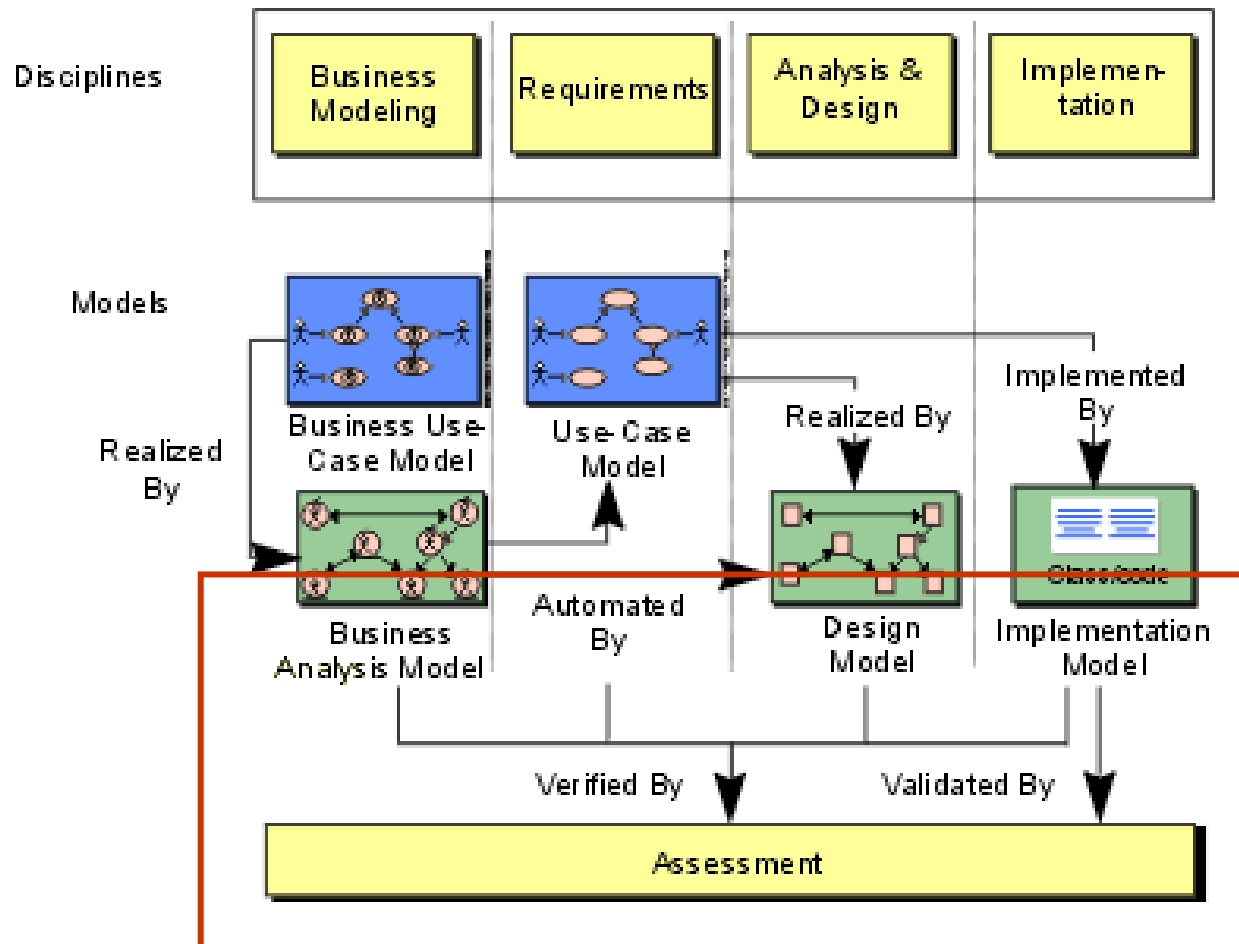
[Rational , 2003]

# Custos de Correção de Erros



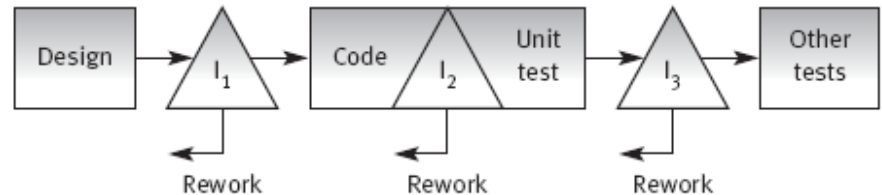
[Boehm]

# Verificação Contínua e Progressiva



# Revisão e Inspeção de Software

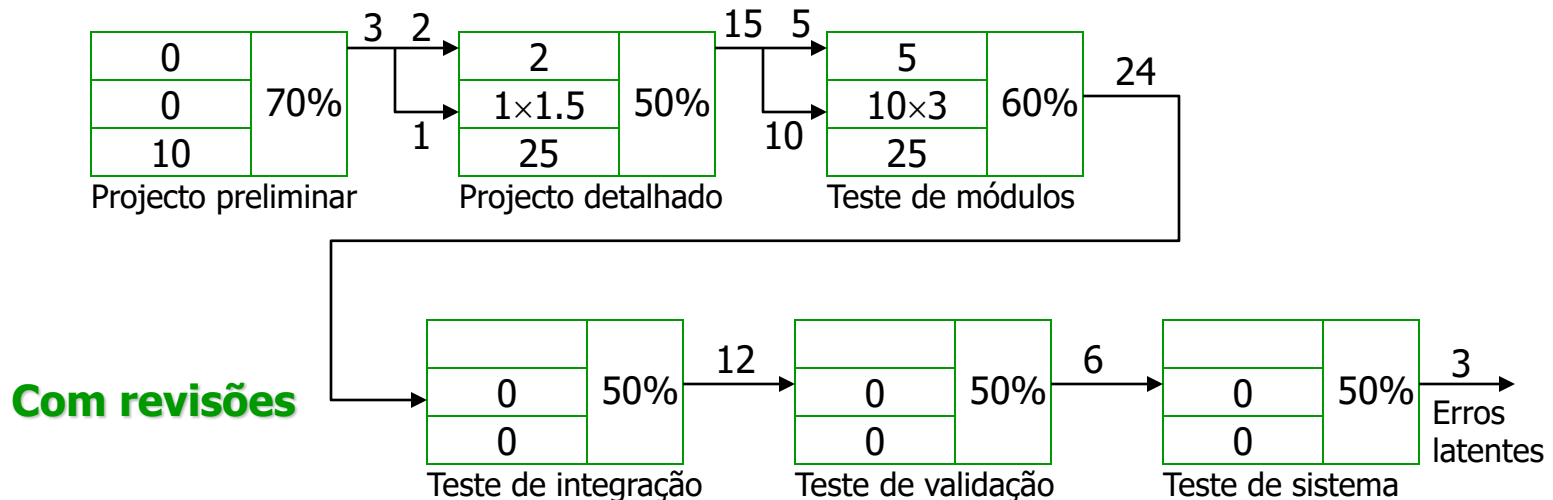
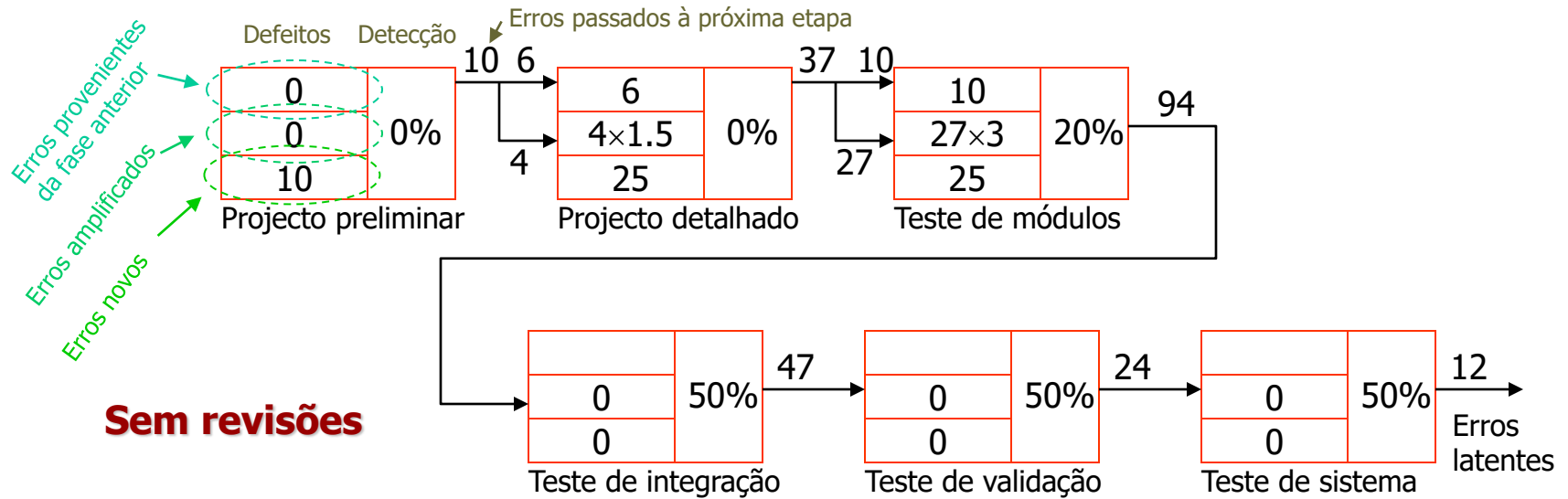
- **Funcionam como um filtro de todo o processo de desenvolvimento**



[Endres & Rombach, 2003]

- **Revisões individuais**
  - Frequentes
  - No papel
  - Atitude de abertura a potenciais problemas ou soluções alternativas
- **Inspeções**
  - Apesar das pessoas conseguirem detectar alguns dos erros cometidos, grandes grupos de **erros escapam mais facilmente a quem os produz**, do que a outras pessoas
  - São uma forma de **utilizar a diversidade de um grupo de pessoas** para:
    - **Identificar necessidades de introdução de melhoramentos** nos produtos de uma pessoa ou de uma equipa
    - **Conseguir um trabalho de qualidade mais uniforme**, ou pelo menos **mais previsível**, do que seria possível sem revisões

# Revisões e Detecção e Remoção de Defeitos



# Inspeções

---

- Declaração de dados:
  - Todas as variáveis foram declaradas?
  - *Arrays* e *Strings* têm valores iniciais correctos?
  - Dimensões e tipos de dados correctos nas variáveis de armazenamento de dados?
  - Existem variáveis com nomes idênticos?



# Inspeções

---

- Referências de dados:
  - São utilizadas variáveis antes de serem afectadas?
  - Indexações ocorrem dentro dos limites ?
  - Definições de estruturas de dados são consistentes entre módulos ?

# Inspeções

---

- Cálculo:
  - Cálculos envolvem variáveis não numéricas ?
  - Cálculos envolvem variáveis de diferentes tipos ou dimensões ?
  - Variável destino de dimensão inferior à dimensão do valor de afectação ?
  - Precedência de operadores é clara ?
  - Divisão por zero ?

# Inspeções

---

- Comparações:
  - Comparação entre variáveis inconsistentes ?
  - Relações de comparação correctas ?
  - Expressões lógicas correctas ?
  - Precedência de operadores é clara ?
  - A avaliação de expressões lógicas é clara ?

# Inspeções

---

- Fluxo de controlo:
  - Todos os ciclos terminam ?
  - Existem condições que possam impedir a execução de ciclos ?
  - Início e fim de instruções correctamente definido ?
  - Existem decisões não exaustivas ?

# Inspeções

---

- Verificação geral:
  - O programa é fácil de perceber ?
  - A arquitectura geral é adequada e clara ?
  - A arquitectura detalhada é adequada e clara ?
  - Ser-nos-á fácil alterar o programa ?
  - Estamos satisfeitos com o programa ?

# Inspeções

---

- **Forma muito eficaz de detectar erros**
  - Inspeções individuais (*desk checking*)
  - Inspeções por outros elementos da equipa de desenvolvimento
  - Inspeções guiadas por listas de verificação
  - Inspeções em grupo

# Inspeções:

## Aspectos a ter em conta

---

- **Rever o produto**, não quem o produz.
- **Definir uma agenda** e manter essa agenda.
- **Limitar o debate**.
- **Definir as áreas a analisar**, mas não tentar resolver todos os problemas referidos.
- Tomar **notas escritas**.
- **Limitar o número de participantes** e insistir na preparação prévia da reunião.
- **Definir uma lista de tópicos a contemplar para cada produto** a rever.
- Garantir que no planeamento do projecto são contemplados os **recursos e o escalonamento necessário às revisões**.
- Garantir **formação adequada** a todos os elementos participantes nas revisões.
- **Rever as revisões** preliminares.

# Teste de Software

- Todos os testes devem ser **relacionáveis com os requisitos** base do sistema.
- O **planeamento dos testes** deve ser realizado à medida que é feita a análise e projecto do sistema em causa.
- Há que ter em conta que, tipicamente, **80% dos erros detectados** durante os testes serão provavelmente **relacionáveis apenas com 20%** dos módulos que constituem o sistema. **O problema está em identificar esses módulos e realizar o seu teste pormenorizado.**
- Inicialmente, os testes devem ter um **âmbito local** (subsistemas) e progressivamente adquirir um **âmbito global** (sistema)
- **Testes exaustivos não são praticáveis.**
- Para terem maior eficácia, os **testes devem ser conduzidos por elementos que não tenham participado na concepção do sistema.**

**Não é possível provar que um sistema não falhará. É contudo possível reduzir drasticamente a probabilidade de falha.**



# Princípios de Teste de Software

---

- **Princípio 1:**

- Um aspecto essencial de um caso de teste é a definição dos resultados esperados

- Psicologia humana:

- Vemos o que estamos predispostos para ver
    - Desejo subconsciente de ver o resultado correcto

- **Caso de teste** deve ser composto por:

- Uma **descrição das entradas** a fornecer ao sistema
    - Uma **descrição precisa do que serão os resultados correctos** mediante essas entradas

# Princípios de Teste de Software

---

2. Um programador deve **evitar testar o seu próprio programa.**
3. **Observar pormenorizadamente os resultados** de cada teste.
4. Devem ser escritos casos de teste não só para condições de entrada inválidas ou inesperadas, mas **também para condições válidas e esperadas.**
5. Examinar um programa para **detectar se ele não faz o que é suposto fazer** é apenas metade do trabalho; a outra metade é **detectar se ele faz o que não é suposto fazer.**
6. Não planear os testes **assumindo que não serão encontrados erros.**
7. A probabilidade de se encontrarem mais erros num troço de um programa é **proporcional ao número de erros já encontrados nesse troço.**

# Níveis de Teste

---

- **Teste de Unidades** (“*Unit Testing*”) :

Cada **módulo do sistema** (classe, mecanismo, subsistema) é **testado individualmente**.

- **Teste de Integração** (“*Integration Testing*”) :

Permite verificar se as **diferentes unidades que compõem o sistema funcionam correctamente em conjunto**. Os *Casos de Teste* utilizados podem ser idênticos aos anteriores, contudo o objectivo é diferente. A utilização dos *Casos de Utilização* como guia para a elaboração destes testes é fundamental.

- **Teste do Sistema** (“*System Testing*”) :

Abrange todo o sistema, utilizando a **perspectiva dos utilizadores finais**, e realizando as acções que estes realizariam.

# Tipos de Teste

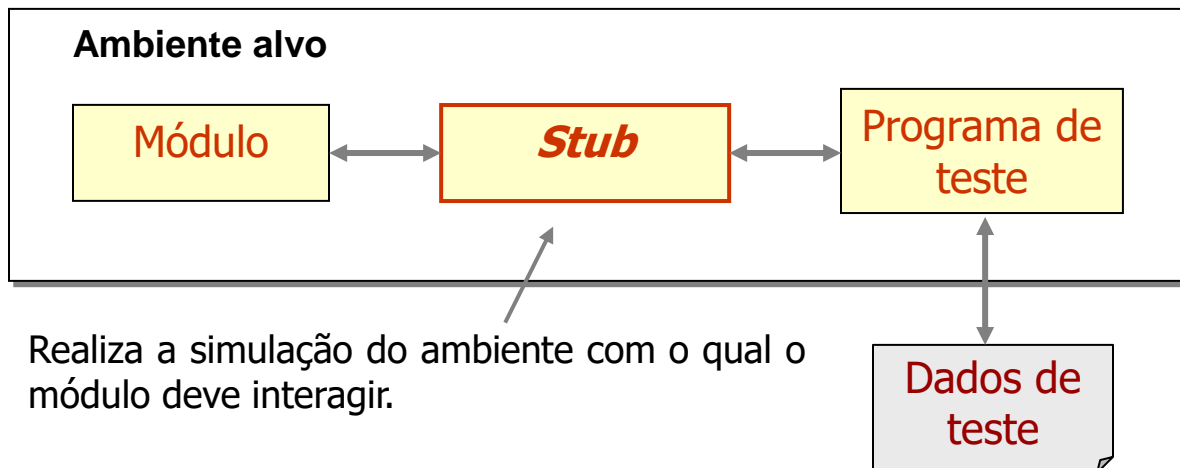
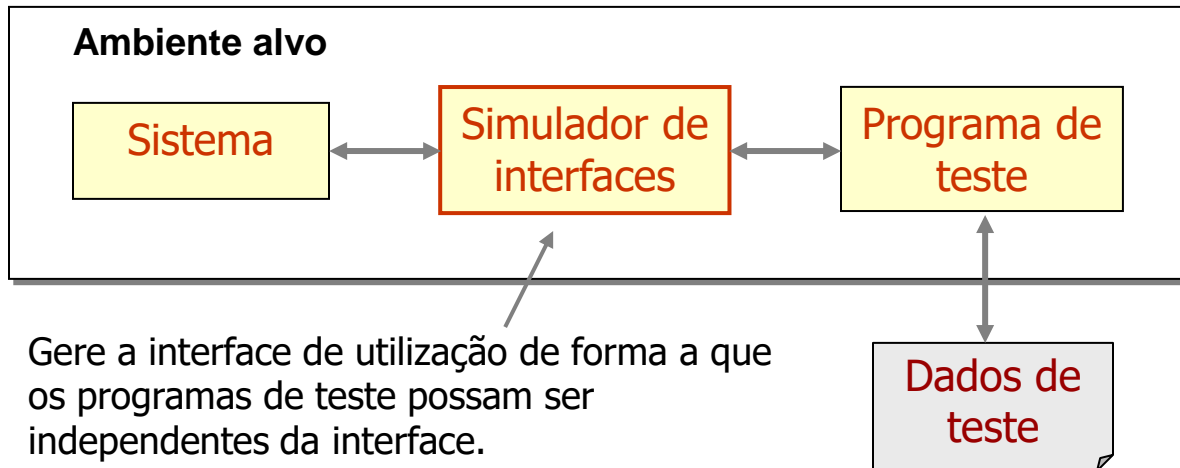
---

- **Teste de regressão:** Utilizado quando são feitas alterações para verificar se as funcionalidades anteriores se mantêm.
- **Teste operacional:** O sistema é testado em operação normal por um determinado período de tempo (utilizado também para aferir a robustez do sistema - *MTBF*).
- **Teste de desempenho ou de capacidade:** Tem por objectivo medir o nível de desempenho do sistema com diferentes *cargas* de utilização.
- **Teste de sobrecarga:** Idêntico ao anterior, mas levando o sistema aos seus limites, de forma a verificar o seu desempenho em situações extremas de utilização.
- **Teste negativo:** O sistema é testado para além dos limites para os quais foi concebido.
- **Teste baseado nos requisitos:** Tem por objectivo a verificação dos requisitos iniciais.
- **Teste ergonómico:** Teste das interfaces de utilização, em termos de consistência com os casos de utilização identificados, e da facilidade e comodidade de utilização.
- **Teste de documentação:** Um tipo de teste ergonómico onde a documentação do sistema é testada.
- **Teste de aceitação:** Corresponde à validação final do sistema, sendo normalmente realizado pela organização que encomendou o sistema.

# Técnicas de Teste

Technique	Brief description
<b>Black-Box Testing</b>	Test cases generated based on the system's functionality
<b>White-Box Testing</b>	Test cases are defined by examining the paths of logic of a system
<b>Branch Coverage Testing</b>	Verify each branch has true and false outcomes at least once
<b>CRUD Testing</b>	Build CRUD matrix and test all object creations, reads, updates, and deletions
<b>Desk Checking</b>	Developer reviews code for accuracy
<b>Exception Testing</b>	Identify error messages and exception handling processes and conditions that trigger them
<b>Inspections</b>	Formal peer review that uses checklists, entry criteria, and exit criteria
<b>Positive and Negative Testing</b>	Test the positive and negative values for all inputs
<b>Prototyping</b>	General approach to gather data from users by building and demonstrating to them some part of a potential application
<b>State Transition Testing</b>	Test cases are written to test the triggers to cause a transition from one state to another

# Ambiente de Teste



# Teste de Unidades

---

## Duas vertentes:

- **Teste estrutural** (“*white-box testing*”):

É utilizado o conhecimento que se tem da estrutura interna dos módulos na realização das actividades de teste.

- **Teste de especificações** (“*black-box testing*”):

Os teste são baseados nas especificações e no comportamento *visível* (exterior) dos módulos.

Uma vez que **são complementares**, normalmente são utilizadas as duas vertentes.

O **teste de especificações é em geral realizado antes do teste estrutural**, pois este pode levar a alterações da estrutura do sistema.

# Teste de Especificações

- **Objectivo base:**
  - Verificar se o sistema se comporta **de acordo com os requisitos especificados**.
- **Procedimento:**
  - São enviados eventos ao sistema, **de acordo com os casos de utilização definidos**, sendo observadas as respectivas respostas.
  - Uma vez que os **módulos devem comunicar através de interfaces**, há que identificar as operações suportadas em cada interface, **verificar se o comportamento observado é o esperado**, e se os **dados envolvidos estão correctos**.

**Não basta verificar o comportamento, é necessário igualmente verificar a correcção dos dados resultantes.**



# Teste Baseado em Estados

- O teste de operações isoladas não é suficiente, é necessário igualmente verificar o resultado cumulativo dessas operações:
  - Particularmente importante para objectos com comportamento caracterizado por máquinas de estado.
  - É necessário verificar como o sistema evolui em termos dos seus estados (e dos atributos respectivos).
  - Deve ser utilizada a tabela de transição de estados na validação do comportamento do objecto.

Estado Evento	Estado-1	Estado-2	Estado-3	Estado-4
Evento-1		OK	OK	Transição incorrecta
Evento-2	OK	Tempo de resposta longo	OK	OK
Evento-3	Resposta incorrecta	OK		

# Teste Estrutural

- **Objectivo: Verificar se a estrutura interna do módulo está correcta.**
- **É utilizado o conhecimento que se tem da estrutura interna dos módulos na realização das actividades de teste.**
  - **Idealmente deveriam ser analisadas todas as combinações possíveis** de parâmetros, valores e caminhos de execução. **Na prática raramente isso é possível**, devido ao número elevado de casos de teste necessário.
  - Para avaliar a eficácia do teste é utilizada a noção de **cobertura do teste**.
  - **A cobertura mínima** corresponde a **testar cada caminho de execução entre decisões (“IF - IF”) pelo menos uma vez** (assim todos os caminhos são testados, e as decisões avaliadas pelo menos uma vez).
  - Apesar de normalmente um teste de cobertura mínima ser razoável, muitos problemas surgem em **combinações de caminhos menos comuns**. Nesse sentido pode **aumentar-se o número de caminhos “IF - IF” testados**, contudo há que ter em conta que o número de casos de teste necessário cresce muito rapidamente.

# Teste Estrutural

## Exemplo:

```
boolean IntSet::Member(int t)
{
    int l = 0;
    int u = CurDim - 1;

    while(l <= u) {
        int m = (l + u)/2;
        if(t < x[m]) u = m - 1;
        else
            if(t > x[m]) l = m + 1;
            else return true;
    }


    return false;
}
```

while pode ser executado zero ou mais vezes

3 caminhos possíveis

Qual o número total de caminhos ?



Execuções do <b>while</b>	Caminhos
0	1
1	3
2	3*3
3	3*3*3
...	...

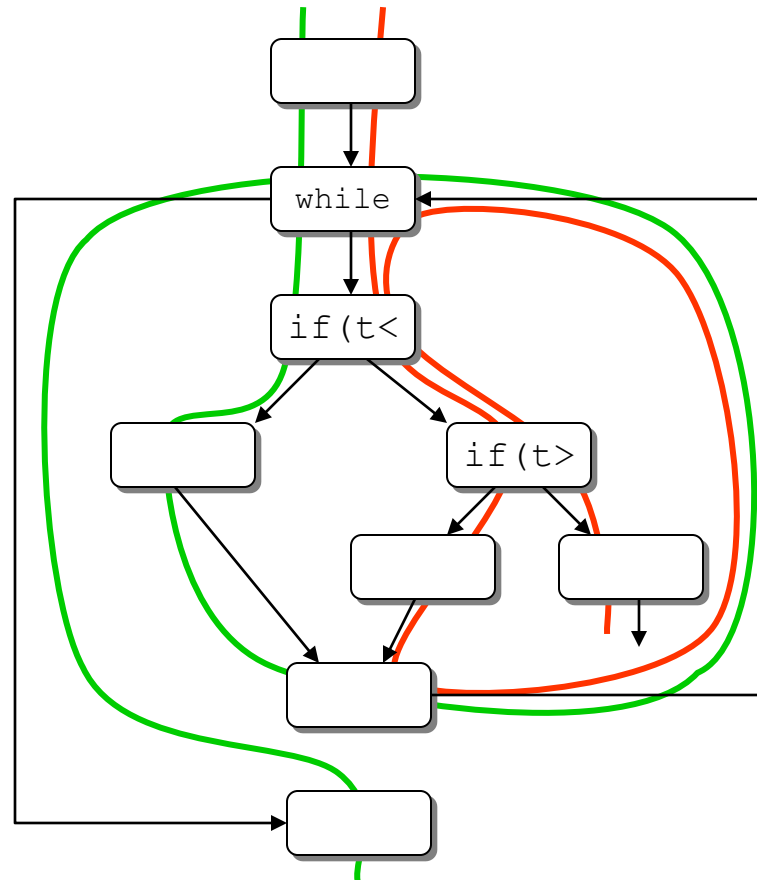

$$1 + \sum_{n>0} 3^n$$

# Teste Estrutural

É necessário escolher um conjunto de casos de teste adequados, que permita testar todos os caminhos.

Para o exemplo anterior são suficientes 2 casos de teste:

-  Caso de teste 1
-  Caso de teste 2



# Teste de Integração

---

- **Após os módulos isolados terem sido testados é necessário realizar a sua integração, sendo por isso necessário verificar se a sua operação em conjunto corresponde ao especificado.**
  - Os testes de integração **abrangem a integração a diferentes níveis** à medida que o sistema vai sendo desenvolvido.
  - Apesar de cada módulo já ter sido testado individualmente é **necessário o seu teste em conjunto, uma vez que novas falhas podem surgir da sua operação conjunta.**
  - Os **casos de utilização devem servir de base** a todo o processo de teste de integração.
  - Cada caso de utilização deve ser **testado individualmente sob duas perspectivas:**
    - **Interna** - baseada nos diagramas de interacção
    - **Externa** - baseada nas descrições do modelo de requisitos

# Teste do Sistema

---

- Após o teste de cada caso de utilização separadamente, o **sistema deve ser testado como um todo**. Os diferentes **casos de utilização devem ser testados em paralelo**, e o sistema sujeito a diferentes *cargas* de utilização.
- O teste do sistema pode ser dividido em 6 tipos de testes:
  - **Testes operacionais**
  - **Testes de desempenho**
  - **Testes de sobrecarga**
  - **Testes negativos**
  - **Testes baseados nos requisitos**
  - **Testes de documentação de utilização**
- Ao testar os casos de utilização do sistema os testes devem ser realizados em paralelo, quer sincronizados, quer não sincronizados.

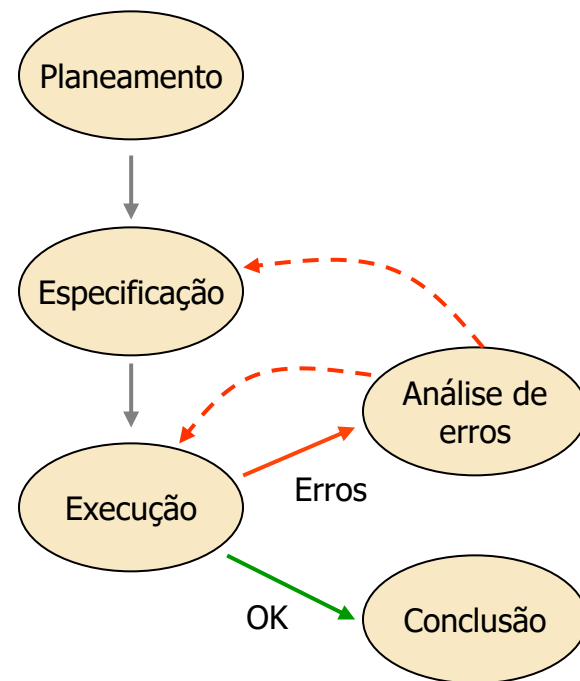
# Processo de Teste

Os testes não devem ser um processo *ad hoc*, devendo antes ser um processo estruturado e incremental, que decorre em paralelo com o restante desenvolvimento do sistema.

## Principais actividades envolvidas:

- Planeamento
- Especificação
- Execução
- Análise de erros
- Conclusão

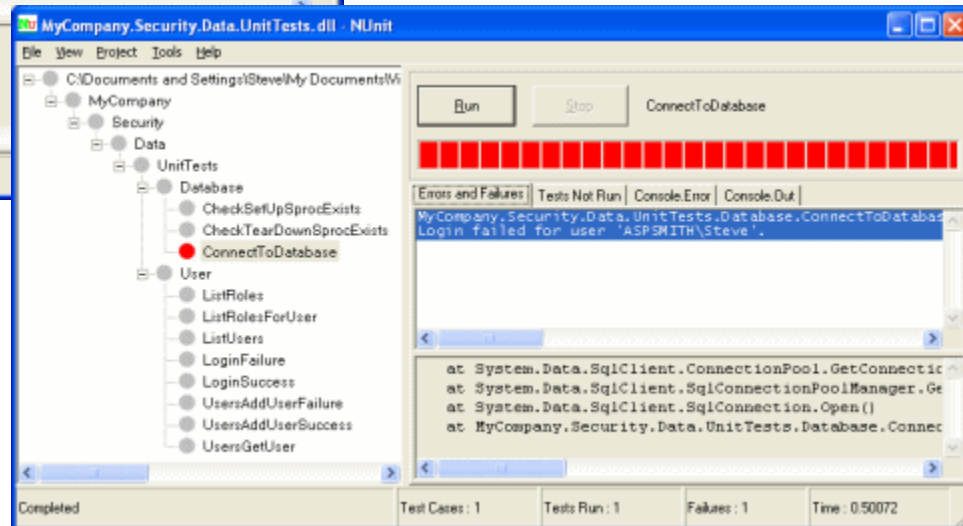
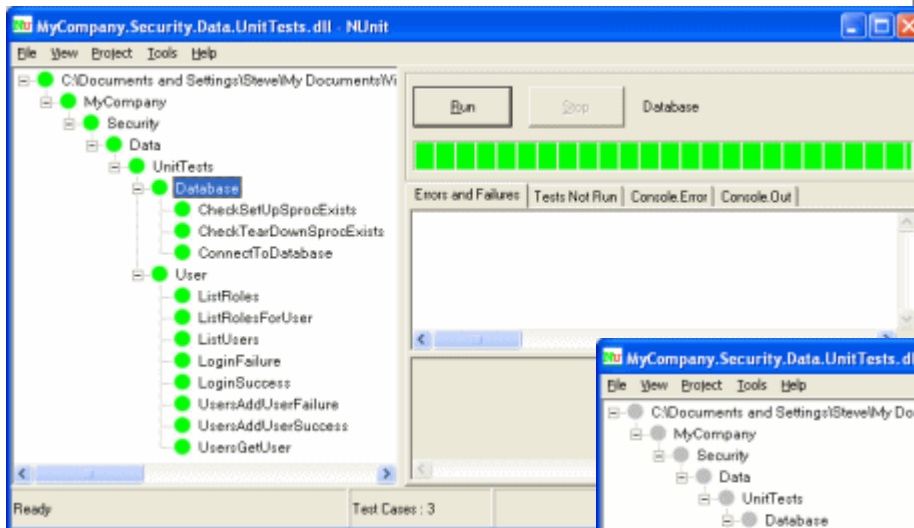
- Testar cedo
- Testar frequentemente
- Testar tudo



# Componentes de Teste

## Exemplo: *NUnit*

```
[Test]
public void TestSimpleAdd()
{
    IntegerList list = new IntegerList();
    list.Add(5);
    list.Add(10);
    Assertion.AssertEquals(2, list.Count);
    Assertion.AssertEquals(5, list[0]);
    Assertion.AssertEquals(10, list[1]);
}
```

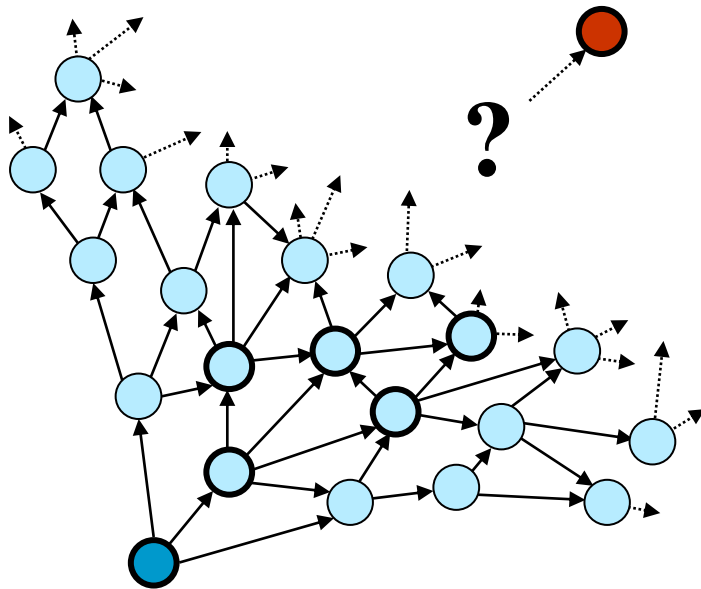




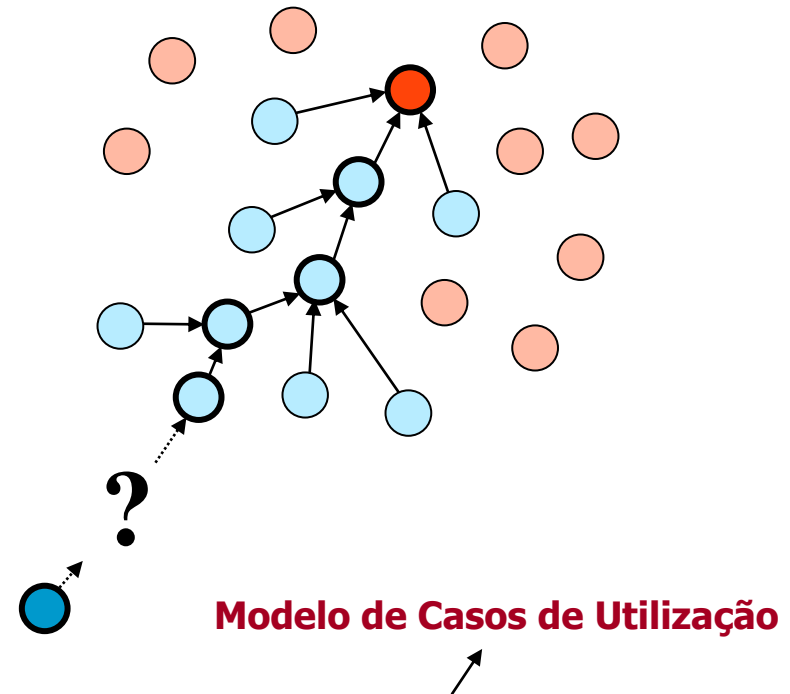
# Desenvolvimento Guiado por Testes

## *Test Driven Development*

- Conceito introduzido nos anos 90 (*SmalltalkUnit* – Kent Beck)
- Testes são escritos antes de ser escrito o código
- Testes servem para verificar se alterações não introduziram erros.



Desenvolvimento clássico:  
O objectivo pode ser difuso de início.



Desenvolvimento guiado por objectivos:  
**O objectivo é o ponto de partida.**

# Bibliografia

---

**Roger Pressman**

*Software Engineering: a Practitioner's Approach*, 2003

**McGraw-Hill**

**Glenford Myers**

*The Art of Software Testing*, 2004

**John Wiley & Sons**