Testes de Software

Metodologias e Desenvolvimento de Software

Pedro Salgueiro
pds@uevora.pt
CLV-256

Testes de software

Outline

- Testes de desenvolvimento
- Test-driven development
- Testes de release
- Testes de utilizador

Testar programas

- Testes servem para mostrar que um programa faz o que deve fazer, e para descobrir problemas e erros antes de ser colocado em utilização.

- Como se testar o software
  - Executa-se o programa com dados artificiais
  - Verifica-se os resultados, procura-se erros, anomalias ou informação sobre os atributos não funcionais do sistema

- Testes podem revelar a presença de erros mas **não** a sua ausência

- Testes fazem parte do processo de verificação e validação mais genérico, que também inclui técnicas de validação estáticas

Objetivos

- Demonstrar ao programador e ao cliente que o software está de acordo com os requisitos
  - Software específico (feito à medida)
    - Pelo menos um teste para requisito no documento de requisitos.
  - Software genérico
    - Testes para todas as funcionalidades do sistema
- Encontrar situações em que o comportamento anómalos do software, que não estão de acordo com os requisitos
  - Testes de defeitos (*defect testing*) tem como objetivo remover comportamentos inesperados do sistema, "*crashes*" do sistema, interações indesejadas, corrupção de dados, etc...

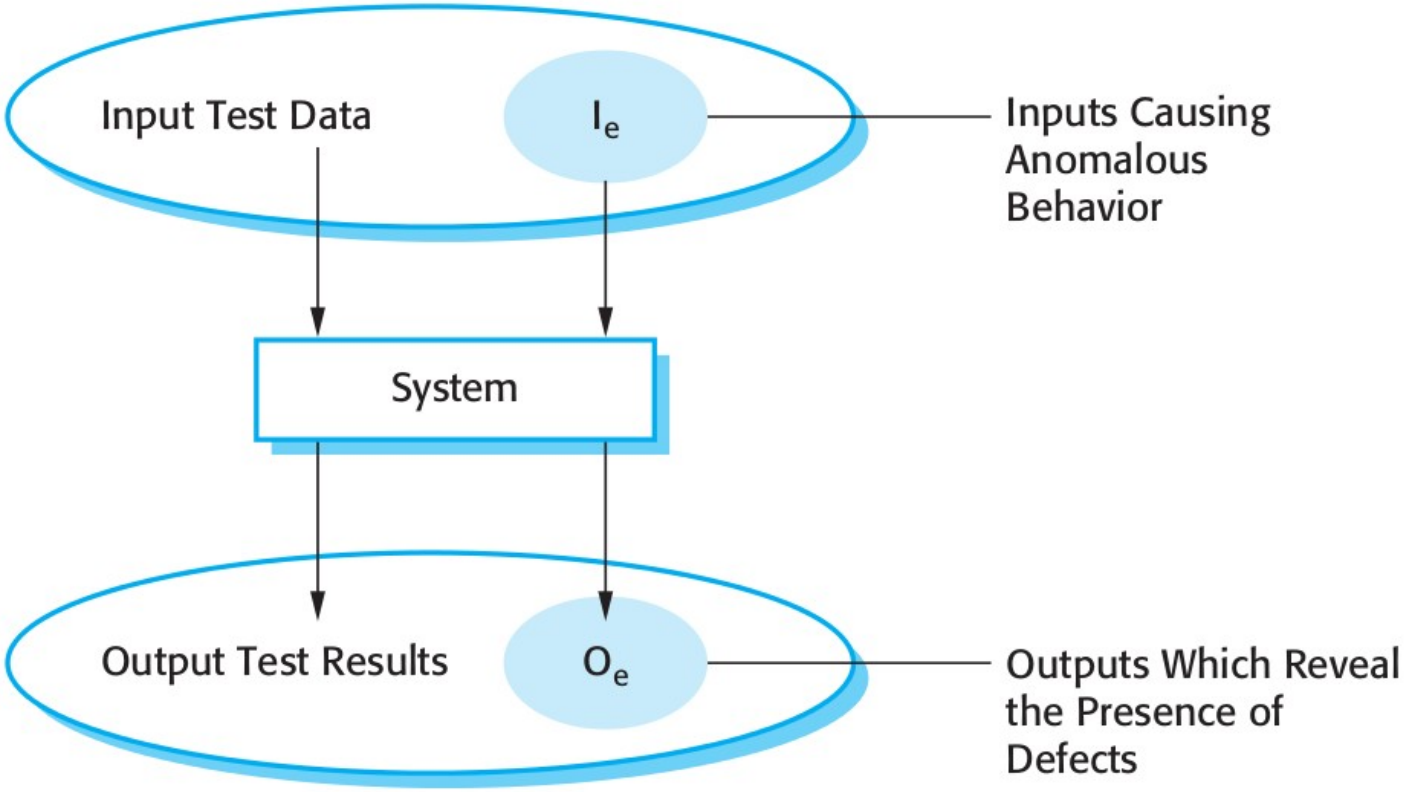
Testes de validação e de defeitos

- 1º objetivo
  - Dá origem aos testes de validação
    - Onde se espera que o sistema tenha o comportamento esperado, usando um conjunto de testes que refletem a utilização normal do sistema
- 2º objetivo
  - Dá origem aos testes de defeito
    - Testes desenhados para revelar defeitos e errosawarte
    - Testes deliberadamente “obscuros” que não precisam de refletir o comportamento normal do sistema

Processo de testes - objetivo

- Testes de validação
  - Demonstrar ao programador e ao cliente que o software está de acordo com os requisitos
  - Um bom teste mostra que o sistema funciona como deve funcionar
- Testes de defeito
  - Encontrar erros e problemas que dão origem a comportamentos anormais e que não estão de acordo com as especificações
  - Um bom teste faz com que o sistema funcione de forma incorreta, expondo os seus erros e defeitos.

Teste de programas - Modelo de input-output



Verificação vs Validação

- Verificação
  - “Estamos a criar o produto de forma correcta?”
  - O software deve estar de acordo com as especificações

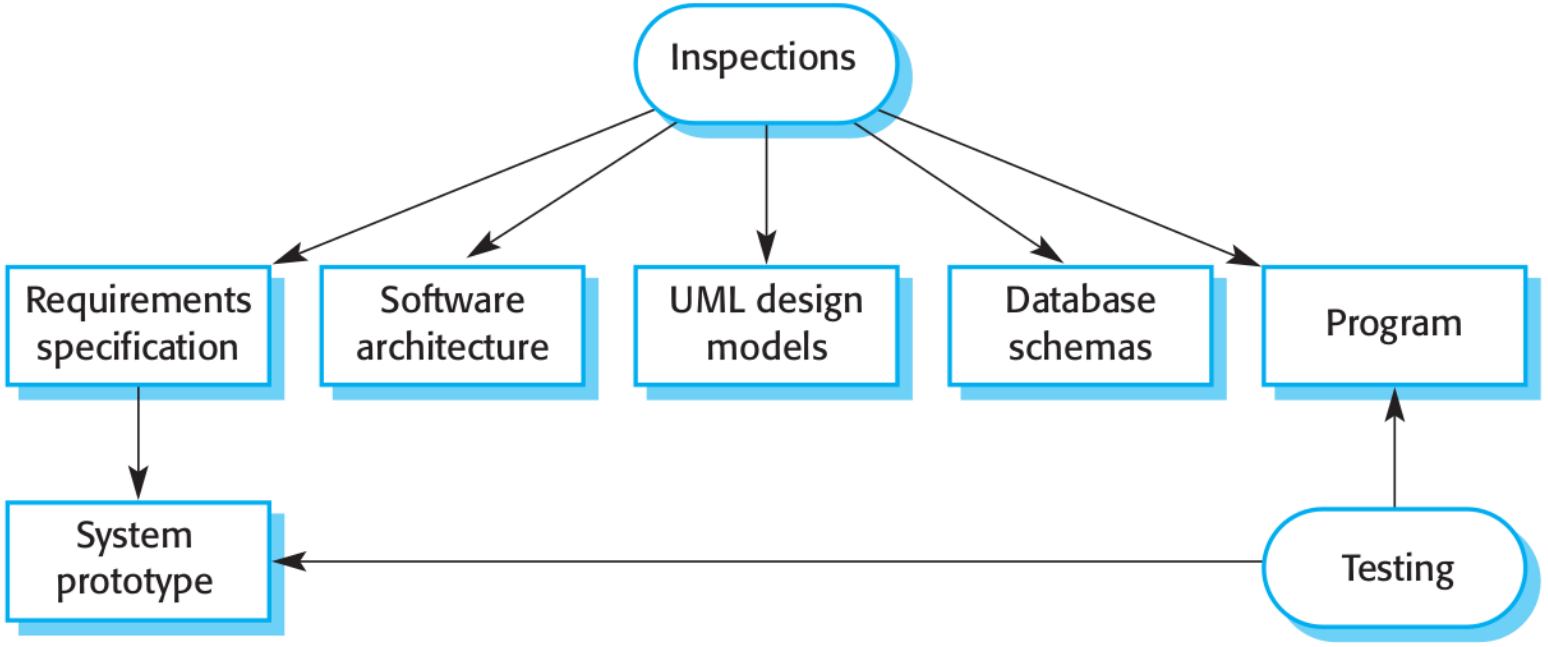
- Validação
  - “Estamos a construir o produto certo?”
  - O software deve fazer aquilo que o utilizador realmente necessita

Inspeções e testes

- Inspeções de software
  - Análise e verificação da representação estática do sistema por forma a encontrar problemas
    - Verificação estática

- Testes de software
  - Observação do comportamento do sistema
    - Verificação dinâmica
    - Sistema é executado com dados de testes, verificando-se o seu comportamento

Inspeções e testes



Inspeções de software

- Análise da representação do código por humanos, com o objetivo de descobrir anomalias e defeitos
- Inspeções não necessitam de executar o sistema
  - Podem ser usadas antes da sua implementação
- Pode ser feita a qualquer representação do sistema
  - Requisitos, desenho, dados de configuração, dados de testes, etc…
- Técnica eficaz para encontrar erros

Inspeções de software

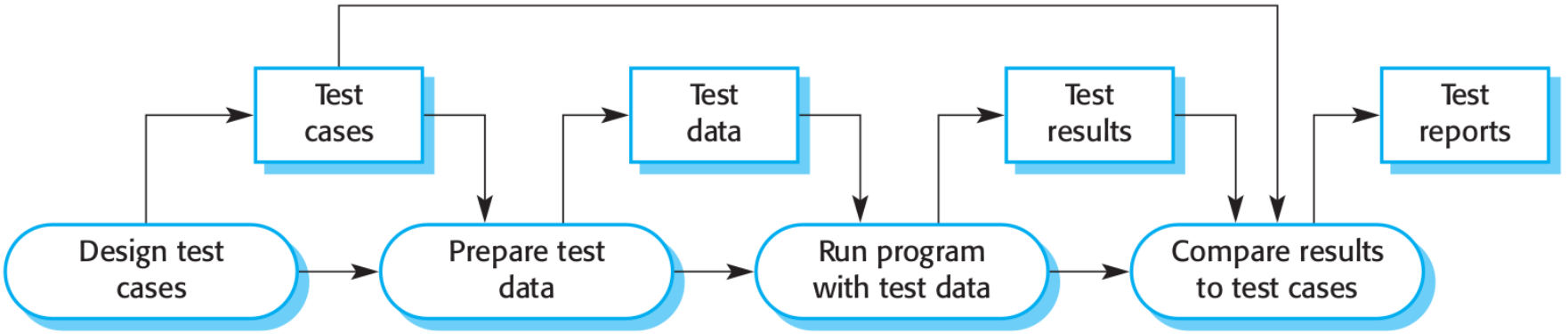
Vantagens

- Durante os testes, os erros podem esconder outros erros.
  - Como as inspeções são um processo estático, não existe a preocupação com interações entre erros.
- Versões incompletas do sistema podem ser inspecionadas sem custos adicionais
  - Se o programa estiver incompleto, é necessário desenvolver testes específicos para testar os componentes disponíveis.
- Para além de se procurarem defeitos
  - Inspeções podem ser úteis para encontrar atributos de qualidade de um programa: satisfação de standards, portabilidade e manutenção

Inspeções e testes

- Inspeções e testes são atividades complementares
- Ambas devem ser usadas no processo de validação e verificação
- Inspeções podem verificar se o sistema está de acordo com as especificações
  - Não podem ser usadas para verificar se o sistema está de acordo com as reais necessidades do cliente
- Inspeções não podem ser usadas para verificar características não funcionais
  - Desempenho, usabilidade, etc...

Processo de teste de software – Um modelo



Testes - etapas

- Testes durante a fase de desenvolvimento
  - *Development testing*
  - Sistema é testado durante a etapa de desenvolvimento
  - Procurar erros e defeitos
- Testes *de Release*
  - Testes a uma versão completa do sistema (*release*)
  - Equipa de testes diferente
  - Antes de ser entregue aos utilizadores
- Testes de utilizador
  - Testes feitos ao sistema por utilizadores reais, no seu ambiente de trabalho

Testes durante a fase de desenvolvimento

- Testes de desenvolvimento incluem todas as atividades de testes realizadas durante o desenvolvimento do sistema

- Testes unitários
  - “Unidades” ou objetos individuais do sistema são testados
  - Testar funcionalidades de objetos ou métodos

- Testes de componentes
  - Várias unidades são integradas por forma a criar componentes
  - Testar o interface entre os vários componentes

- Testes ao sistema
  - Todos os componentes são integrados
  - Teste do sistema como um todo
  - Testar interações entre componentes

Testes unitários

- Processo de testar componentes individuais de forma isolada
- Processo de testes de defeitos
- “Unidades” podem ser
  - Funções ou métodos individuais de um objeto
  - Objetos com vários atributos e métodos
  - Componentes com interfaces bem definidos para aceder às suas funcionalidades

Testar objetos de classes

- Cobertura completa dos testes a uma classe envolve
  - Testar todas as operações associadas com um objeto
  - Especificar e ler todos os atributos de um objeto
    - *Setters* e *getters*
  - Testar o objeto em todos os possíveis estados
- Herança pode tornar difícil o desenho de testes a objetos
  - Informação não está localizada

Testes autónomos

- Sempre que possível, os testes unitários devem ser automatizados
  - Executar e verificar os testes sem intervenção manual

- *Frameworks* de testes
  - Criar e executar os testes
  - e.g.: JUnit
  - Fornecem classes de testes genéricas
    - Estendidas para criar testes específicos
  - Executar todos os testes implementados
    - Gerar relatórios

Testes autónomos – componentes

- Setup
  - Inicialização do sistema e do teste
    - Inputs
    - Outputs esperados
- Chamada/Execução
  - Método, função ou objeto é testado
- Verificação/Assertion
  - Comparação do output da chamada com a output esperado
    - Teste passou com sucesso: True
    - Teste não passou: False

Testes unitários – Quais implmentar?

- Testes unitários devem mostrar
  - Componentes testados fazem o esperado
  - Se houver defeitos ou problemas, devem ser revelados pelos testes

- Origem a dois tipos de testes
  - Testes que refletem o uso normal do sistema e mostrar que o componente funciona de acordo com o esperado
  - Testes que usam inputs pouco comuns ou anormais
    - Verificar que o sistema é capaz de os processar e não fazem o sistema “crashar”

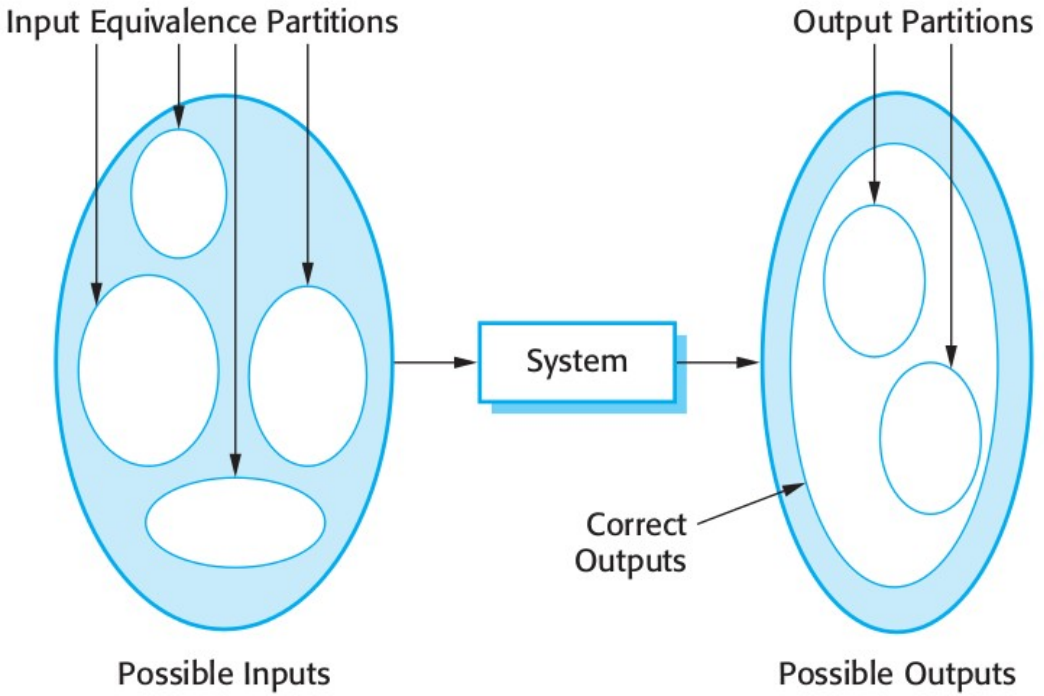
Estratégias de testes

- Particionar os testes
  - Identificar grupos de inputs que têm características comuns e devem ser processados da mesma forma
    - Escolher testes das categorias identificadas
- Baseado em regras/guias
  - Testes escolhidos de acordo com algumas regras
  - Regras/guias baseadas na experiência dos programadores
    - Nos erros comuns de programação

Particionar os testes

- Dados de input e output enquadram-se em diferentes classes
  - Todos os membros de uma classe estão relacionados
- Cada classe
  - Partição equivalente
  - Sistema comporta-se de forma igual para cada membro da classe
- Casos de teste devem ser escolhidos a partir de cada partição

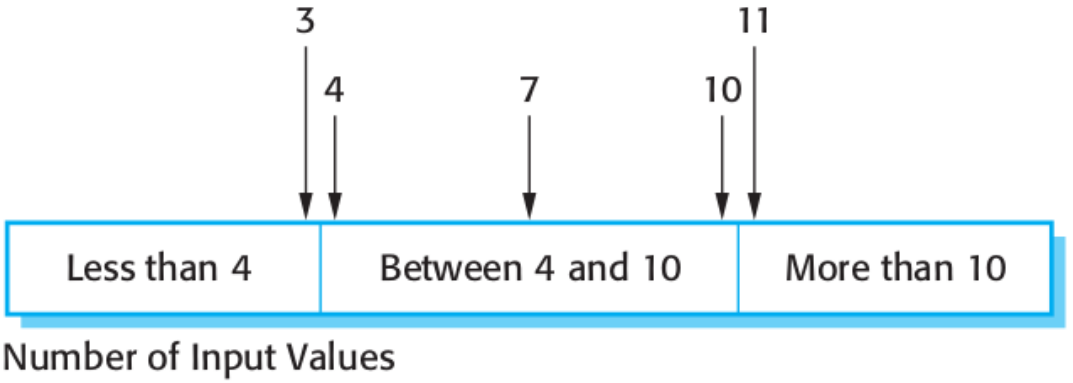
Partições equivalentes

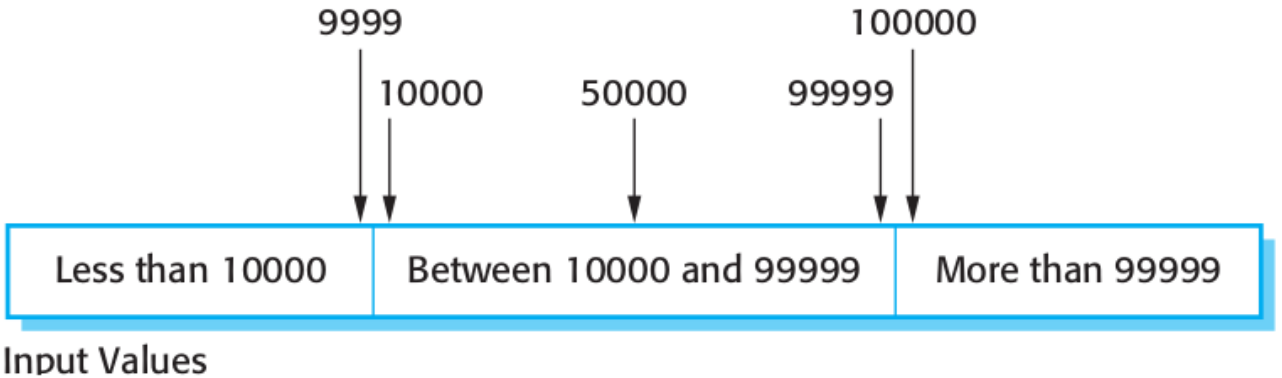


Partições equivalentes – exemplo

- Um programa aceita entre 4 e 10 inputs
- Cada input
  - 5 dígitos
  - > 10,000

Partições equivalentes - exemplo





Guião - testar sequências

- Sequências
  - listas, arrays, etc…

- Testar usando sequências apenas com um valor
- Testar o primeiro, ultimo e elemento do meio
- Testar sequências com tamanho zero

Guião genérico para escrever testes

- Escolher inputs que forcem o sistema a gerar todas as mensagens de erros possíveis
- Escolher inputs que possam causar “buffer overflows”
- Repetir os mesmos inputs várias vezes seguidas
- Forçar a geração de outputs inválidos
- Forçar processos que produzam resultados muito pequenos ou muito grandes

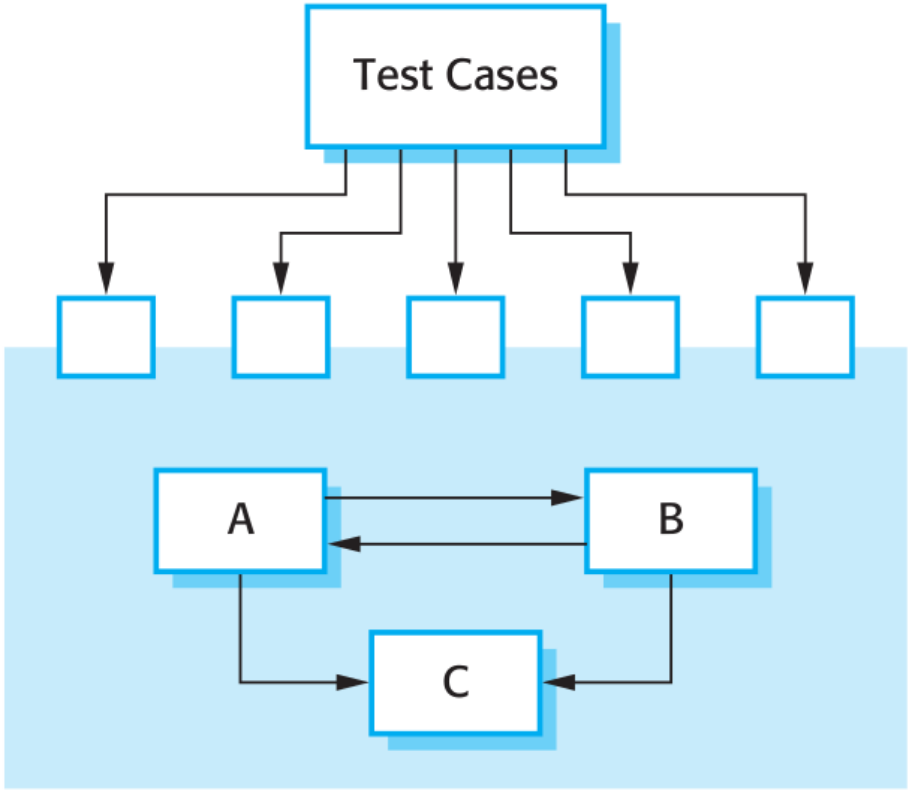
Testes de componentes

- Componentes de software
  - Componentes compostos
  - Resultam da interação entre objetos
- Acede-se à funcionalidade do objeto através do seu interface
- Testes de componentes compostos
  - Devem focar-se em mostrar que o interface tem um comportamento de acordo com a sua especificação
  - Deve-se assumir que os testes unitários estão terminados (e completos)

Testar interfaces

- Objetivo
  - Detetar falhas
  - Erros de interface
  - Suposições erradas

Testes de componentes



Erros de interfaces

- Má utilização
  - Chamada de componente usando o interface de forma errada, e.g.: ordem errada dos parâmetros
- Interface mal percebido
  - Utilização de componente de forma errada, devido a suposições erradas
- Problemas de sincronização
  - Acesso a dados desatualizados

Guião - testar interfaces

- Desenhar testes de forma a que parâmetros tomem valores extremos do seu domínio
- Testar parâmetros do tipo apontadores com valores nulos
- Desenhar testes que façam os componentes falhar
- Colocar o sistema sob *stress*
- Variar a forma com os componentes são chamados/ativados

Testes de sistema

- Teste de sistema durante a fase desenvolvimento
  - Integrar componentes
  - Criar uma versão do sistema
  - Testar o sistema integrado
- Foco
  - Testar interações entre os componentes
- Objetivo
  - Testar se os componentes são compatíveis
  - Se os dados corretos são transferidos através dos interfaces
  - Testar o comportamento global do sistema

Testes de sistema e de componentes

- Durante a etapa de testes de sistema
  - Integração de componentes externos, implementados de forma isolada, ou reutilizáveis
    - Deve ser feita a integração destes componentes com os componentes do sistema
    - Deve ser testado do o sistema
  - Integração de componentes desenvolvidos por outras equipas
    - Deve ser testado do o sistema
    - Processo coletivo

Testes baseados em Use Cases

- Use Cases
  - Identificam interações do sistema
  - Podem ser usados como base dos testes do sistema
- Cada Use Case
  - Envolve vários componentes
  - Testar um Use Case força a interação entre os vários componentes

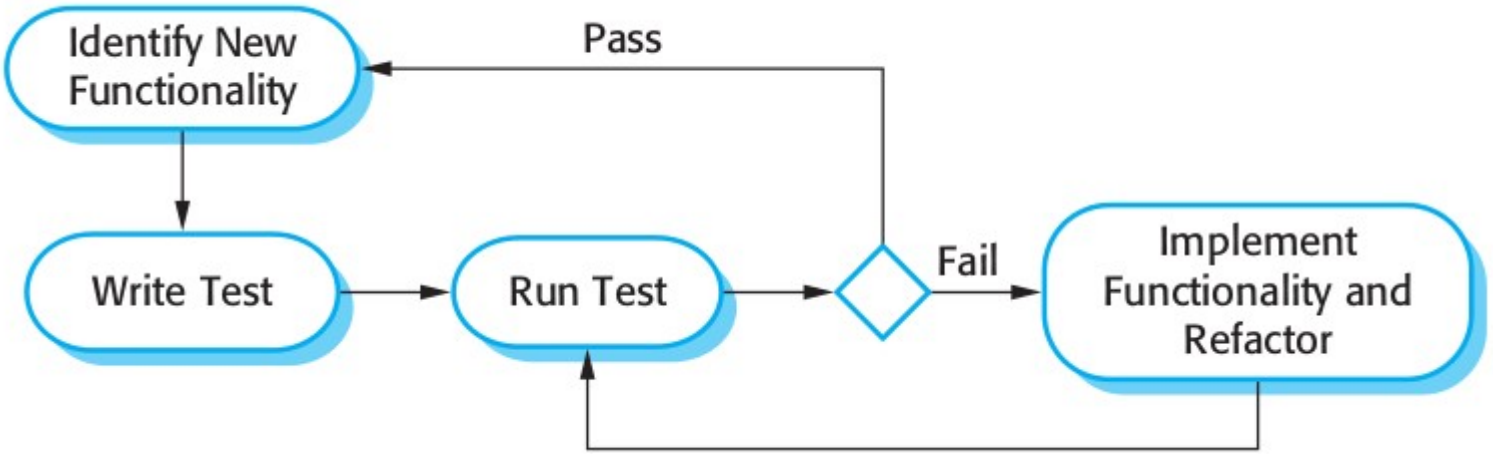
“Políticas”/Regras de testes

- Testes exaustivos
  - É impossível
  - Necessário criar regras que estabeleçam uma cobertura mínima
- Exemplos
  - Todas as funções acessíveis via menus devem ser testadas
  - Combinações de funções que são acedidas através do mesmo menu devem ser testadas
  - Todas funções que trabalho com dados introduzidos pelos utilizadores devem ser testadas

Test-driven development (TDD)

- Método de programação
  - Intercalar testes com desenvolvimento de código
- Testes escritos antes de implementar código
  - Fator critico no desenvolvimento: testes têm de “passar” com sucesso
- Código desenvolvido de forma incremental
  - Sempre com testes para esse incremento
  - Não se avança para outro incremento sem que todos os testes passem com sucesso
- Base dos métodos ágeis
  - Pode ser usado em metodologias baseadas em planos

Test-driven development



Test-driven development – Atividades

- Identificar o incremento da funcionalidade a implementar
  - Incremento pequeno (poucas linhas de código)
- Implementar o teste para o novo incremento
  - Teste autónomo
- Executar o novo teste
  - E todos os outros testes
  - Inicialmente o teste vai falar (a funcionalidade ainda não foi implementada)
- Implementar a funcionalidade e voltar a executar os testes
  - Continuar até todos os testes passarem
- Quando todos os testes passarem
  - Escolher novo incremento para implementar

Test-driven development – Benefícios

- Cobertura do código com testes
  - Todos os segmentos de código estão associados pelo menos com um teste
- Testes de regressão
  - Criados de forma incremental à medida que o sistema é desenvolvido
- Debug simplificado
  - Quando um teste falha, torna-se óbvio qual o problema
- Documentação do sistema
  - Testes servem de documentação ao sistema

Testes de regressão

- Testar o sistema por forma a verificar se alterações não introduziram erros no código previamente funcional

- Considerando um processo manual
  - Difícil e dispendioso

- Considerando um processo automático
  - Simples
  - Todos os testes são executados sempre que existe uma alteração no sistema

- Todos os testes devem passar com sucesso antes do código ser “commited”

Testes de Release

- Testar uma *release* específica do sistema
  - Que será colocada em utilização fora do ambiente de desenvolvimento
- Objetivo
  - Mostrar que o sistema está pronto para ser usado
  - Mostrar que o sistema implementa o que especificado
- Testes de caixa-preta (black-box)
  - Ignora-se a implementação do sistema
  - Testes criados a partir das especificações
  - Verificar as saídas de acordo com as entradas

Testes de release e de sistema

- Testes de release
  - São um tipo de testes de sistema

- Diferenças importantes
  - Testes de release
    - Não devem ser feitos pela equipa de desenvolvimento
    - Verificar se o sistema cumpre os requisitos
    - Verificar se o sistema está suficiente bom para utilização
    - Testes de validação

  - Testes de sistema
    - Feitos pela equipa de desenvolvimento
    - Encontrar problemas
    - *Defect testing*

Testes de desempenho

- Podem fazer parte dos testes de release
  - Desempenho e fiabilidade
- Devem
  - Refletir o perfil de utilização do sistema
  - Variar a carga do sistema de forma até que o sistema se torne inutilizável
- Testes de stress
  - Tipo de teste de desempenho
  - Sistema é sobrecarregado por forma a avaliar o seu comportamento quando falha

Testes de utilizador

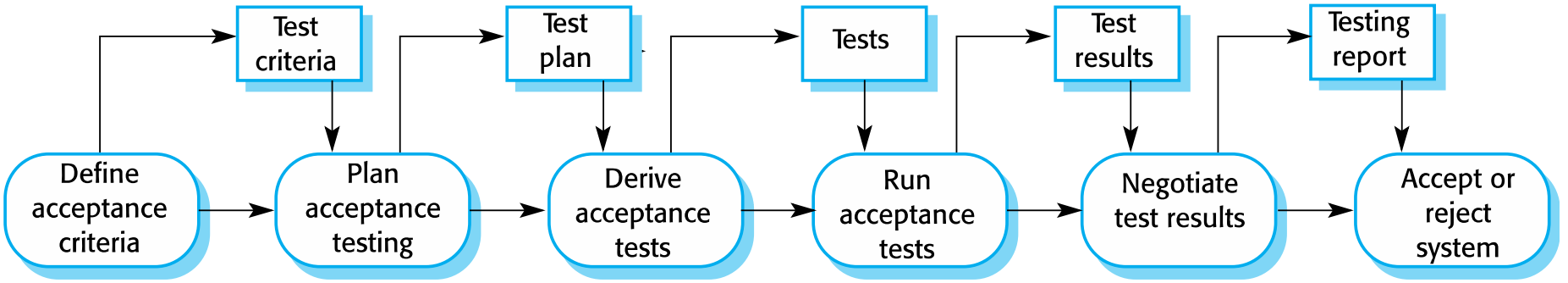
- Utilizadores ou clientes indicam como testar o sistema
  - Ajudam na fase de testes

- Fase de testes essencial
  - Mesmo quando são feitos os testes de sistema e de release
  - Porquê?
    - Influencias dos utilizadores podem ter grande impacto na fiabilidade, desempenho e usabilidade do sistema.
    - Estas influências não podem ser replicadas nos testes de sistema ou de release.

Testes de utilizador - Tipos

- Testes alfa
  - Utilizadores trabalham com a equipa de desenvolvimento para testar o software no ambiente de desenvolvimento
- Testes beta
  - Utilizadores testam uma release do sistema para podem experimentar o sistema e encontrar problemas, juntamente com a equipa de desenvolvimento
- Testes de aceitação
  - Decidir se o sistema está pronto para começar a ser usado

Testes de aceitação - processo



Testes de aceitação - Etapas

- Definir o critério de aceitação
- Planear os testes de aceitação
- Criar os testes de aceitação
- Executar os testes de aceitação
- Negociar os resultados dos testes
- Aceitar ou rejeitar o sistema

Métodos ágeis e testes de aceitação

- Métodos ágeis
  - Utilizador/Cliente faz parte da equipa de desenvolvimento
    - Responsáveis por decisões relacionadas com a aceitação do sistema
- Testes são definidos pelo utilizador/cliente e integrados com outros testes
  - Corridos de forma automática
- Não existe etapa (separada) de testes de aceitação
- Problema
  - Garantir que o cliente/utilizador representa os interesses de todos os interessados do sistema

Bibliografia

- Software Engineering. Ian Sommerville. 10th Edition. Addison-Wesley. 2016. Capítulo 8.