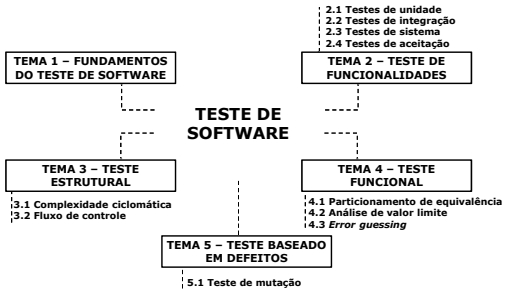
Aula 5

Engenharia de Software

Prof. Alex Mateus Porn

Conversa Inicial



Fundamentos do teste de software

- **"O teste é um processo individualista e o número de tipos diferentes de testes varia tanto quanto as diferentes abordagens de desenvolvimento de software"** (Pressman, 2011, p. 401)

- **Segundo Wazlawick (2013):**
  - **Erro (*error*): diferença detectada entre o resultado obtido e o resultado esperado**
  - **Defeito (*fault*): linha de código ou conjunto de dados incorretos que provocam um erro**
  - **Falha (*failure*): não funcionamento do software, geralmente ocorre por um defeito**
  - **Engano (*mistake*): ação que produz um defeito no software**

- **Conforme Delamaro, Maldonado e Jino (2007):**
  - **O engano caracteriza-se como ação equivocada de um conceito específico do domínio, geralmente causado por humanos**
  - **A existência de um defeito em um programa leva à ocorrência de um erro, desde que o defeito seja executado**

- **Ainda segundo Delamaro, Maldonado e Jino (2007):**
  - **O erro caracteriza-se por um estado inconsistente ou inesperado, devido à execução de um defeito**
  - **Esse estado inconsistente ou inesperado pode ocasionar uma falha**

Exemplificando

1. **Solicitar dois números inteiros ao usuário do programa**
2. **Os números informados devem ser maiores do que 1**
3. **Se os números informados forem menores ou iguais a 1, deverá ser avisado que estão incorretos**
4. **Se os números digitados forem maiores do que 1, o programa deve computar x^y**

5. **Após computar x^y, o programa deve perguntar ao usuário se ele deseja realizar um novo cálculo**
6. **Se a resposta do usuário for "Sim", o programa deverá solicitar mais dois valores**
7. **Se a resposta do usuário for "Não", o programa deverá ser encerrado**

Exemplificando

```
1  programa
2  {
3      funcao inicio() {
4          inteiro x, y, result
5          escreva("Digite o valor da base e do expoente na sequência")
6          leia(x, y)
7          enquanto((x <= 1) ou (y <= 1)) {
8              escreva("Valores incorretos, digite novamente")
9              escreva("Digite o valor da base e do expoente na sequência")
10             leia(x, y)
11         }
12         result = x;
13         enquanto(y > 1) {
14             result = result * x
15             y = y - 1
16         }
17         escreva("Resultado = ", result)
18     }
19 }
```

- **Engano**
  - **Linha 7: trocar os sinais de "<=" por "<"**

```
7           enquanto((x <= 1) ou (y <= 1)) {
```

  - **Linha 15: trocar o sinal na operação "y – 1" por "y + 1"**

```
15                  y = y - 1
```

- **Defeito**
  - **Se for digitado 1 para a base e qualquer número positivo para o expoente, ao processar a linha 7 o programa computará o cálculo e identificará um defeito incompatível com o requisito 3**

- **Erro**
  - **O defeito executado na linha 7 produzirá o erro de não gerar a mensagem conforme o requisito número 3 e computar o cálculo incompatível com o requisito 4**

- **Falha**
  - **Ao executar a linha 15, será identificado o defeito de uso de um operador matemático incorreto, que produzirá um erro na geração do cálculo, ocasionando uma falha conhecida como *loop infinito*, na qual o programa não será finalizado e não apresentará o resultado esperado pelo usuário**

Verificação, validação e teste

- **Conforme Wazlawick (2013):**
  - **Verificação: analisa o software para ver se foi construído de acordo com a especificação**
  - **Validação: analisa o software construído para ver se atende às necessidades dos interessados**
  - **Teste: atividade que permite realizar a verificação e a validação do software**

Casos e dados de teste

- **Segundo Delamaro, Maldonado e Jino (2007):**
  - **Dado de teste: é um elemento do domínio de entrada de um programa**
    - **Programa que computa x^y, um dado de teste pode ser (2, 5)**
  - **Caso de teste: é um par formado por um dado de teste mais o resultado esperado**
    - **Programa que computa x^y, o caso de teste do dado de teste (2, 5) seria ((2, 5), 32)**

Teste de funcionalidades

- Para Wazlawick (2013):
  - Tem como objetivo verificar e validar se as funções implementadas no software estão corretas
  - Nesse método de teste, são encontrados os testes de unidade, integração, sistema e aceitação

Testes de unidade

- De acordo com Wazlawick (2013):
  - São os mais básicos e consistem em verificar se um componente individual do software (unidade) foi implementado corretamente
  - Consideraremos neste exemplo como uma unidade do programa o laço de repetição implementado entre as linhas 13 e 16 do programa que computa x^y

Teste de integração

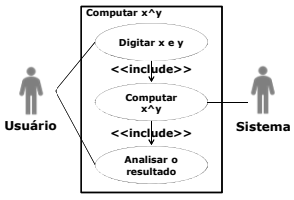
- Segundo Wazlawick (2013):
  - É feito quando unidades estão prontas, são testadas isoladamente e precisam ser integradas para gerar uma nova versão do sistema
  - Para exemplificar, consideremos como duas unidades do programa que computa x^y:
    - Unidade do exemplo do teste de unidade
    - Unidade do laço de repetição que verifica se os valores de x e y são maiores que 1

Teste de sistema

- Conforme Wazlawick (2013):
  - Verifica se a atual versão do sistema permite executar processos ou casos de uso completos do ponto de vista do usuário, sendo capaz de obter os resultados esperados
  - Se cada uma das operações do sistema já estiver testada e integrada corretamente, então, deve-se verificar se o fluxo principal do caso de uso pode ser executado corretamente, bem como os fluxos alternativos

Teste de sistema

- Diagrama de casos de uso do Sistema x^y



- Caso de uso: digitar x e y
  - Entrada: usuário informa os valores de x e y
  - Alternativo: usuário informa um valor incompatível, x ou y > 1
    - O sistema solicita novos valores ao usuário, e o caso de uso é reiniciado
- Caso de uso: computar x^y
  - O programa calcula automaticamente x^y com base nos valores informados pelo usuário
- Caso de uso: analisar o resultado
  - Saída: usuário analisa o resultado obtido

Teste de aceitação

- Para Wazlawick (2013):
  - Teste realizado pelo cliente ou usuários do sistema, que consiste na aceitação da aplicação desenvolvida
  - Costuma ser realizado utilizando-se a interface final do sistema

Teste estrutural

- Conhecido como *teste de caixa branca*, pois todos os testes são executados com conhecimento do código-fonte
- É capaz de detectar uma quantidade substancial de defeitos pela garantia de ter executado pelo menos uma vez todos os comandos e condições do programa (Wazlawick, 2013)
- Destacam-se entre os possíveis critérios de teste os critérios baseados na complexidade e no fluxo de controle

Critérios baseados na complexidade

- Conforme Delamaro, Maldonado e Jino (2007):
  - Utilizam informações sobre a complexidade do programa para derivar requisitos de software
  - Dois critérios bastante conhecidos são a complexidade ciclomática e os caminhos linearmente independentes

Complexidade ciclomática

- Métrica que proporciona uma medida quantitativa da complexidade lógica de um programa (Delamaro; Maldonado; Jino, 2007)

- IF-THEN: 1 ponto
- IF-THEN-ELSE: 1 ponto
- CASE: 1 ponto para cada opção, exceto OTHERWISE
- FOR: 1 ponto
- REPEAT: 1 ponto
- OR ou AND acrescenta-se 1 ponto para cada um
- NOT: não conta
- Chamada de sub-rotina ou recursiva: não conta
- Estruturas de seleção e repetição em sub-rotinas ou programas chamados: não conta

- Conforme Wazlawick (2013):
  - Simples e fáceis de testar: ≤ a 10
  - Médio risco em relação ao teste: > 10 e ≤ 20
  - Alto risco: > 20 e ≤ 50
  - Não testáveis: > 50

Exemplificando

- Programa que computa x^y
  - Dois laços de repetição "enquanto" = 2 pontos
  - Um operador "ou" = 1 ponto
  - Complexidade ciclomática = 3 pontos

Caminhos linearmente independentes

- Refere-se a qualquer caminho do programa que introduza pelo menos um novo conjunto de instruções de processamento ou uma nova condição (Delamaro; Maldonado; Jino, 2007)
- Para analisar os caminhos de um programa, sugere-se usar o Grafo de Fluxo de Controle (GFC) (Wazlawick, 2013)
- Todos os comandos são representados em nós
  - Fluxos de controle em arestas

Exemplificando

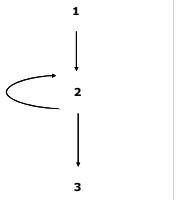
Código Fonte

```
escreva("Digite o valor da base...")
leia(x, y)

enquanto((x <= 1) ou (y <= 1)) {
                    .
                    .

enquanto(y > 1) {
                    .
                    .
}
```

Grafo de Fluxo de Controle

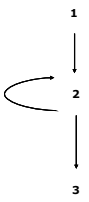


Critérios baseados no fluxo de controle

- De acordo com Delamaro, Maldonado e Jino (2007):
  - Fazem uso do GFC de modo similar ao critério de caminhos linearmente independentes
  - Utilizam apenas características de controle da execução do programa, como comandos ou desvios, para determinar quais estruturas são necessárias
  - Destacam-se os critérios: Todos-Nós, Todas-Arestas e Todos-Caminhos

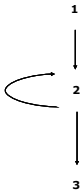
Critério Todos-Nós

- Exige que a execução do programa passe, ao menos uma vez, em cada vértice do GFC
- Conforme a figura, o critério Todos-Nós exige que sejam criados casos de teste que executem ao menos uma vez os nodos 1, 2 e 3



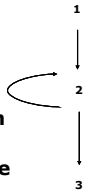
Critério Todas-Arestas

- Requer que cada aresta do grafo seja exercitada pelo menos uma vez
- Conforme a figura, o critério Todas-Arestas exige que sejam criados casos de teste que passem ao menos uma vez pelas arestas (1, 2), (2, 2) e (2, 3)



Critério Todos-Caminhos

- Requer que todos os caminhos possíveis do programa sejam executados
- Conforme a figura, o critério Todos-Caminhos exige que sejam definidos casos de teste que percorram os caminhos (1, 2, 3) e (1, 2, 2, 3) ao menos uma vez



Teste funcional

- É executado sobre as entradas e saídas do programa sem que se tenha conhecimento do seu código-fonte, sendo, portanto, identificado como *teste de caixa preta*
- Destacam-se como principais critérios do teste funcional o particionamento em classes de equivalência, a análise do valor limite e o *error guessing*

Particionamento em classes de equivalência

- Entrada especificada como um intervalo de valores
  - Um conjunto válido e dois inválidos
- Entrada especificada como uma quantidade de valores
  - Um conjunto válido e dois inválidos
- Entrada especificada como um conjunto de valores aceitáveis
  - Um conjunto válido para cada uma das formas de tratamento
  - Um conjunto inválido para outros valores quaisquer
- Entrada especificada como uma condição
  - Um conjunto válido e um inválido

Exemplificando

- Para o programa que computa x^y, o domínio de entrada corresponde a um intervalo de valores
  - Conjunto válido
    - ((2, 3), 8), ((3, 2), 9), ((5, 6), 15625), ((4, 3), 64)

- Conjunto inválido 1
  - ((2, 1), "Erro"), ((3, 0), "Erro"), ((5, -2), "Erro"), ((4, 1), "Erro")
- Conjunto inválido 2
  - ((1, 3), "Erro"), ((-5, 2), "Erro"), ((0, 6), "Erro"), ((-4, 3), "Erro")

Análise de valor limite

- Entrada especificada com um intervalo de valores
  - Testa-se os limites desse intervalo e os imediatamente subsequentes

- Entrada especificada com uma quantidade de valores
  - Testa-se com nenhum valor de entrada, somente um valor, o limite de valores e um acima do limite
- Se entrada ou saída for um conjunto ordenado
  - Maior atenção ao primeiro e ao último elemento
- Usar a intuição para definir outras condições limites

Exemplificando

- Para o programa que computa x^y, o domínio de entrada corresponde a um intervalo de valores
  - Conjunto de casos de teste no limite
    - ((2, 2), 4)

- Conjunto de casos de teste imediatamente inferior ao limite
  - ((2, 1), "Erro"), ((1, 2), "Erro"), ((1, 1), "Erro"),
- Conjunto de casos de teste imediatamente superior ao limite
  - Impossível determinar

Error guessing

- **"Essa técnica corresponde a uma abordagem ad-hoc na qual a pessoa pratica, inconscientemente, uma técnica para projeto de casos de teste, supondo por intuição e experiência alguns tipos prováveis de erro e, a partir disso, definem-se casos de teste que poderiam detectá-los"** (Delamaro; Maldonado; Jino, 2007, p. 24)

Teste baseado em defeitos

Teste de mutação

- **Geração dos mutantes**
- **Execução do programa em teste**
- **Execução dos mutantes**
- **Análise dos mutantes vivos**

```
7           enquanto((x <= 1) ou (y <= 1)) {
```