Fundamentos de Teste de Software

Instrutor: Alexandre L. Martins

Tema 1

Instrutor: Alexandre L. Martins

Origem dos erros.

A maioria dos erros tem causa HUMANA pois depende da

- capacidade,
- habilidade,
- interpretação e
- execução

feita por pessoas.

Validação e Verificação

- Validação: determinar o grau em que um sistema atende seus requisitos, no sentido de atender as necessidades do usuário.
  - sistema útil
  - sistema confiável
- Verificação: determinar a consistência de uma implementação a sua especificação.

Teste de Software

Atividade dinâmica que visa determinar se o comportamento do software está de acordo com seu comportamento esperado.

Termos mais usados.

- Bug: problema em um sistema como falha ou defeito.
- Falha: ocorre quando um sistema não opera na forma esperada.
- Erro: é a manifestação de uma falha.
- Defeito: comportamento em um sistema causado por um mais erros.[2]

Termos mais usados.

- Erro: algo que foi feito ou ocorreu de maneira incorreta.
- Falha: manifestação de um ou mais erros.
- Defeito: comportamento indesejável de um sistema gerado por uma ou mais falhas[1].

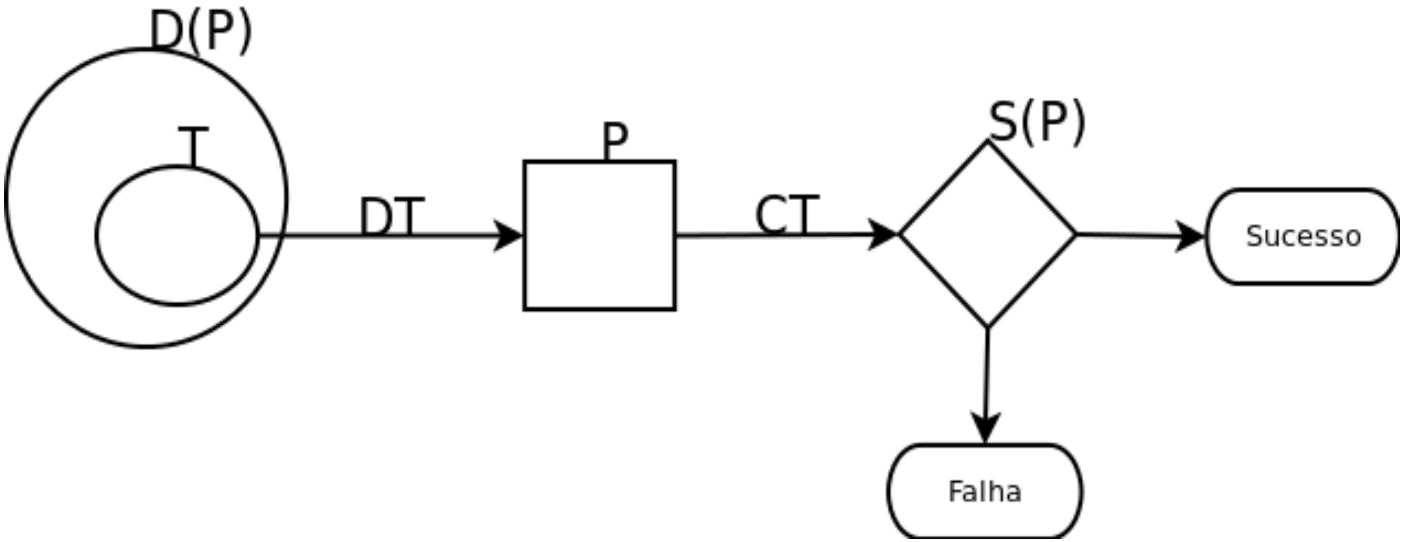
Termos mais usados.

- Defeito(*fault*): passo, processo ou definição de dados incorretos.
- Engano(*mistake*): ação humana que produz um defeito.
- Erro(*error*): é gerado por um defeito em tempo de execução.
- Falha (*failure*): quando um erro leva a uma saída inesperada.

Termos mais usados.

- Domínio de Entrada: conjunto de todos os possíveis valores que executam um programa P, D(P).
- Dado de Teste: um elemento de D(P).
- Caso de Teste: par formado por um elemento de D(P) e sua respectiva saída.
- Conjunto de Casos de Teste: T é subconjunto de D(P).
- Especificação de P: S(P).

Arquitetura Geral de Um Teste



Exemplo

O programa P recebe dois inteiros positivos e deve determinar qual se são iguais ou qual é o maior.

- Domínio, todos os pares inteiros de (x,y).
- Dado de Teste, (1,2); (2,3);(-1,2)...
- Caso de Teste. <(1,2), 2>; <(2,3), 3>; <(-1,2), erro>...

Entradas Inválidas

- Importante avaliar entradas invalidas (idade, cpf, valores esperados etc.)
- Teste de valores inválidos pode revelar erros.
- O teste deve contemplar o Domínio de entrada válido e inválido.

Técnica e Critério de Teste

- Problema: como escolher os elementos de T?
  - deve ter alta PROBABILIDADE de revelar defeitos.
  - deve ser o menor possível.

- Solução: determinar subdomínios de D(P).

Exemplo

O programa P recebe dois inteiros positivos e deve determinar qual é o maior ou igual.

- Subdomínio:
  - S' = {<(1,2), 2>; <(2,3), 3>;...}; = {<(1,2), 2>}
  - S'' = {<(2,1), 2>; <(3,2), 3>;...}; = {<(3,2), 3>}
  - S''' = {<(1,1), 1>; <(2,2), 2>;...}; = {<(1,1), 1>}
  - S'''' = {<(-1,1),erro>; <(-2,-2), erro>;...}; = {<(-1,1), erro>}
- quanto menor a cardinalidade melhor.
- alta representatividade.
- teste de regressão.

Requisitos de teste

- Regra que define o subdomínio
  - especificação
  - exercitar uma dada estrutura
    - percorrer caminhos
    - instruções
    - pontos de decisão
- Ideal: para cada requisito apenas um caso de teste.

Fases de Teste

- Teste de Unidade
  - aplicado pelo desenvolvedor
  - erro de estrutura de dados, algoritmos incorretos ou simples erros de programação.
- Teste de Integração
  - equipe de desenvolvedores
  - erro de interface.
- Teste de Sistema
  - uma equipe de teste
  - feito ao termino do sistema
  - requisitos não funcionais (segurança, robustez, etc.)
  - Corretude, completude e coerrência.

Qualidade de Software

Qualidade de Software: **grau de adequação** das características de um dado software aos requisitos definidos antes do seu desenvolvimento.

Qualidade de Software

Medidas de qualidade, dois grupos:

- Atributos dinâmicos: as características do sistema quando executado e

- Atributos Estáticos: código e documentação

Atributos de Qualidade Estática

- Estrutura do código,
- Manutenção,
- Viabilidade de teste e
- Qualidade da Documentação.

Atributos de Qualidade Dinâmica

- Confiabilidade - sem falha,
- Corretude - requisito observados,
- Completude - requisitos implementados/incorporados,
- Consistência - convenções e padrões,
- Usabilidade - simplicidade de uso,
- Performance - tempo de execução/uso de memória

Confiabilidade, duas definições

- É a propriedade de um sistema operar sem falha em um dado intervalo de tempo e para um dado uso.

- É a propriedade do sistema de operar sem falha em um dado ambiente.

Confiabilidade, uma certeza e um limite

Nenhum sistema complexo é **100%** livre de falhas.

Corretude X Confiança

- Testes não garantem Corretude de um software.
- Corretude deve ser matematicamente demonstrada.
- Não é possível testar todo Domínio de Entrada.
- Prova de corretude garante *error-free*.
- Teste tem por objetivo diminuir a probabilidade de falhas. Aumentar o grau de Confiança da aplicação.

Corretude X Confiança

- Corretude: o programa está correto ou não está.

- Confiança: a probabilidade de um sistema operar sem falhas (0% até 100%). Quanto mais erros você acha, no momento certo, maior a confiança.

Teste X Debugar

Debugar: localizar e corrigir erros.

Teste: processo sistemático de localização de falhas.

Teste X Debugar

Teste

- Execução sistemática do programa para encontrar erros,
- Medir qualidade do programa,
- Analise da documentação e do código e
- Permite tomar decisões sobre a duração, intensidade e alvo do teste.

Exercício 1 - Fundamentos

- Suponha um programa P que recebe como entrada um arranjo de inteiros de tamanho 'n' e fornece como saída o arranjo ordenado e o maior elemento no arranjo. Determine,

1. D(P).
2. Um exemplo de um Dado de Teste de D(P).
3. Determinar S(P).
4. Determine os casos de teste de T.
5. Quais subdomínios podem ser definidos.

- Suponha n = 10 e o valor que pode ser atribuído a cada posição do arranjo variando entre 0 - 9.
  a. Qual o tamanho de D(P)?
  b. É viável realizar testes exaustivos em P? Justifique.

- A especificação de P pode ser melhorada? Justifique.

Exercício 2 - Fundamentos

- Dada a especificação:
  - O programa deve receber como entrada três valores inteiros. Deve retornar se estes três valores correspondem aos lados de um triângulo isósceles, equilátero ou escaleno. Caso os valores não constituam um triângulo o programa deve retornar uma exceção do tipo *ArithmeticException*.
    - D(P).
    - Um exemplo de um Dado de Teste de D(P).
    - Determinar S(P).
    - Determine os casos de teste de T.
    - Quais subdomínios podem ser definidos?
    - Quais tipos de falha este programa pode manifestar? Descreva duas em detalhes.

Exercício 3 - Fundamentos

- Dada a especificação:
  - Uma interface de cadastro de fornecedores possui as seguintes entradas:
    - Nome do fornecedor (String alfa-numérica de tamanho máximo de 100 caracteres alfa-numéricos).
    - CPF do fornecedor.
    - Nome do produto fornecido.
    - Quantidade do produto fornecido (Valor inteiro maior que zero)
    - Data do cadastro do fornecedor (dd/mm/aaaa)
  - Todos os campos são obrigatórios e devem ser preenchidos. Não preeenchimento de um campo gera nova tela na qual os campos não preenchidos ficam assinalados em vermelho.
  - Todos os campos devem ser preenchidos de forma correta. Não preeenchimento correto de um campo gera nova tela na qual os campos preenchidos de maneira incorreta ficam assinalados em verde.

Exercício 3 - Fundamentos

- Para esta especificação você recebe o seguinte relatório de teste:
  - Campo nome: Machado de Assis, ok
  - Campo cpf: 1241338383, ok
  - Produto: livros, ok
  - Quantidade: 10, ok
  - Data: 12/12/2013, ok
- Com base no relatório apresentado, você aceita que o software foi plenamente testado? Justifique sua resposta.

Tema 2

Instrutor: Alexandre L. Martins

Teste Funcional

Instrutor: Alexandre L. Martins

Introdução

A derivação de casos de teste se dá com base apenas nas especificações do programa.

Especificação Funcional

Uma descrição do comportamento esperado do programa.

Denominação

- Teste Funcional
- Teste Caixa-Preta
- Teste Baseado em Especificações

Particionamento

- Casos de teste são alocados em classes geradas com base nas especificações.
- Todos os elementos da classe possuem a mesma capacidade par identificar um defeito.
- Reduz o tamanho do D(P).
- Permite identificar singularidades (valores especiais e valores limites).

Particionamento X Geração Aleatória

- Particionamento
  - custo alto para identificar as partições
  - pode ser automatizado
  - critério de seleção de elementos mais eficiente utilizando técnicas de amostragem
  - não sobreposição de elementos
- Geração Aleatória
  - baixo custo
  - gera muitos casos de teste
  - sensível a conjunto de casos de teste não homogênios

Exercício - Particionamento

Suponha P um programa que identifica se um triângulo é isósceles, equilátero ou escaleno.

1. Quais as possíveis particionamentos de D (P)? De um exemplo de um elemento pertencente a cada classe.
2. A geração aleatória de casos de teste é eficiente para testar este programa? Justifique.

Exercício - Particionamento

História: O analista de teste responsável pelo teste do programa TRIÂNGULO reportou que o aplicativo satisfaz as especificações mas gera uma exceção quando são introduzidos valores que não correspondem aos lados de um um triângulo (<(12,0,0), exception>). O analista reprovou o produto.

A decisão do analista está correta? Justifique sua resposta.

Abordagem Sistemática

- Identificar as especificações funcionais
  a. documentação
  b. informais
- Identificar funcionabilidades de teste independentes.
  c. cada funcionabilidade possui uma classe que a testa.
  d. menos custosa que tentar identificar um elemento que teste várias funcionabilidades.
- Identificar Classe de Valores representativos ou derivar um modelo.
  e. Identificar a classe de valores consiste em enumerar os possíveis valores de entrada que pertencem a classe(valores limites, singularidades etc.)
     i. baixo custo
     ii. técnica simples baseada na leitura direta da especificação
  f. derivar um modelo consistem em modelar a especificação e com base na modelagem derivar os casos de teste
     i. alto custo devido a construção do modelo
     ii. especificação descrita na forma de uma máquina de estados finitos.

Abordagem Sistemática

- Gerar especificações de casos de teste
  - combinar os valores de entrada de uma dada especificação funcional.
  - identificar combinações singulares, invalidas e repetidas
- Gerar casos de teste e instanciar os testes
  - um ou mais elementos de D(P) são selecionados com base na especificação dos casos de teste válidas

Diretrizes da Abordagem Sistemática

Valores numéricos

a. selecione
   i. valores discretos: testar todos os valores.
   ii. intervalos de valores: testar extremos e um valor dentro do intervalo.
b. Valores Especiais e Singularidades
   i. valores especiais como espaços em branco devem ser explorados na entrada e na saída.
   ii. valores singulares (ex. zero) devem ser sempre selecionados
c. Valores ilegais
d. Números reais

- Intervalos variáveis (x < y) selecione
  i. x = y = 0
  ii. x = 0 < y e y = 0 < x
  iii. 0 < x = y e 0 < y = x
  iv. 0 < x < y e 0 < y < x
  v. x < 0 e y < 0
- Arranjo
  a. os elementos devem ser testados como descrito anteriormente
  b. testar tamanhos limites do arranjo
- String
  a. explorar comprimento
  b. validade de caracteres

Vantagens do Teste Funcional

- Pode e deve ser usado em qualquer etapa de desenvolvimento de software
- Indiferente a detalhes de implementação
  - linguagem (Java, Prolog, SML etc.)
  - habilidades do programador
  - estrutura de código
  - ambiente de execução
- Pode ser automatizado (especificação formal ex. Z)
- Auxilia na compreensão das próprias especificações aumentando o conhecimento do sistema

Desvantagens

- Qualidade do teste depende da qualidade da ESPECIFICAÇÃO.
  - especificações são sujeitas a erros
  - especificações são feitas, geralmente, em linguagem natural
- Particionamento é uma tarefa difícil e exige
  - alta capacitação
  - muita experiência
- Linguagens formais para especificação são complexas e exigem capacitação

Exercício

Dada a especificação:

Uma software realiza operações de +, - , * e / sobre inteiros e reais. Os operandos devem ser do mesmo tipo. Operações sobre inteiros resultam em inteiros sobre reais reais. A aplicação possui uma interface gráfica com 16 dígitos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, C(clear) e =. As operações são sempre do tipo (operando1 operador operando2). A aplicação retorna "Error" quando ocorre um erro de operação.

1. Identifique as unidades testáveis independentemente.
2. Gere um conjunto de caso de testes baseado na especificação.
3. Identifique valores limites e singularidades.

Baseado em Young e Pezzè

Exercício

Dada a especificação: P recebe como entrada dois valores inteiros e retorna a soma destes valores.

História: O analista de teste usou os seguinte caso de teste <(2,2), 4> e constatou que P satisfaz a especificação. Um segundo analista de teste atribuiu a P o caso de teste <(2,5),10> e reprovou P. Os ambos os testes são independentes e os analistas só possuem informações do teste que realizaram.

Qual analista tomou a decisão correta? Justifique.

Teste Estrutural

Instrutor: Alexandre L. Martins

Introdução

A definição dos critérios de teste se dá com base na estrutura da implementação.

Denominação

- Teste Estrutural
- Teste Caixa-Branca(White-Box)

Grafo

O código de uma aplicação é descrito na forma de um grafo no qual:

- N: conjunto de nós
- E: conjunto de arestas
- s: nó inical
- o: nó de saída
- G=(N,E,s,o)

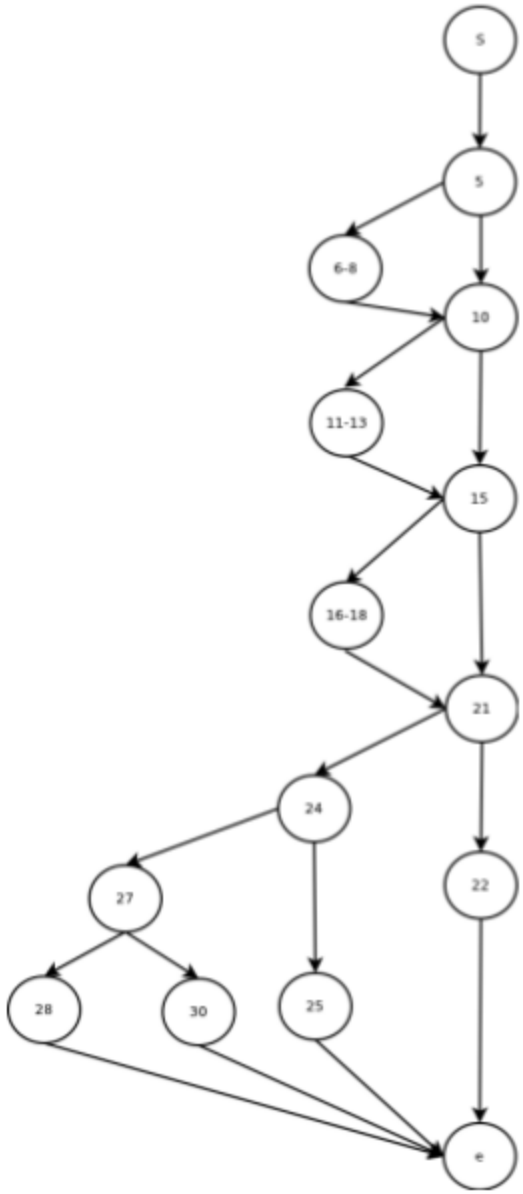
```java
public static void triType(int a, int b, int c){

        int t;

        if (a > b){
            t = a;
            a = b;
            b = t;
        }
        if(a > c){
            t = a;
            a = c;
            c = t;
        }
        if(b > c){
            t = b;
            b = c;
            c = t;
        }

        if(a+b <= c){
            System.out.println("Not a Triangle");
        }else{
            if(a==b && b==c){
               System.out.println("Equilateral");
            }else{
                if(a==b || b==c){
                    System.out.println("Isoceles");
                }else{
                    System.out.println("Scalene");
                }
        }
    }
}
```

codigo/listagem1.java


5
5
6-8
10
11-13
15
16-18
21
24
22
27
28
30
25
e

Caminho

Sequência de execuções possíveis no Grafo.

- $c = (n_1, n_2, n_3,...,n_k)$ para $k >= 2$.
- entre n_i e n_{i+1} existe uma única aresta.
- c é sempre finito.

Tipos de Caminhos

- Viáveis
  - c = <s, 5, 10, 15, 21, 22, e>
- Inviáveis
  - c = <s, 5, 6-8, 10, 15, 21, 24, 25, e>
- Inexistente
  - não implementado.

Considerações sobre os caminhos

- Viáveis: representam execuções validas do sistema.

- Inviáveis: representam comportamentos que não podem ser exercitados. Caso sejam, indica a presença de um erro.

Exercício

Construa o Grafo de Fluxo de Controle do programa bolha.

- Identifique um caminhos viáveis.
- Há caminhos inviáveis?

```java
public class Bolha {
    public static void main(String[] args) {
        // TODO code application logic here
        int a[] = {4,2,3,1,7};
        int i   = 0;
        int aux = 0;
        boolean fimOrd = false;

        while(!fimOrd){
            fimOrd = true;
            i = 0;
            while(i < a.length - 1){
                if(a[i] > a[i+1]){
                    aux = a[i+1];
                    a[i+1] = a[i];
                    a[i] = aux;
                    i++;
                    fimOrd = false;
                }else{
                    i++;
                }
            }
        }
    }
}
```

Elementos do Fluxo alvo de Cobertura

- Teste de comando
- Teste de Decisão
  - DC
- Teste de Condição
  - CC
  - C-DC
  - MC/DC
  - RC/DC
  - M-CC
- Teste de Caminho

Teste de Comandos

- Todos os comandos do código devem ser exercitados.

- Todos os comandos podem ser executados sem que todos os caminhos sejam exercitados. Suponha a listagem "triType" suprimindo a linha 22. teste não identifica o erro.

- Métrica
  - C = (#de desvios executados/# de desvios)*100.
  - todo comando pertence a um desvio.

- Um caso de teste por comando é necessário.

Teste de Decisão

- DC - Decision Coverage
- Toda decisão no código de ve ser exercitada para true e false
- Decisão: ponto de desvio no fluxo formado por uma expressão booleana ou que pode ser valorada como true ou false.
  - if(a || b);
  - while(a);
  - if(a < 4);
  - a = b && c;

Teste de Decisão(caso de teste)

- Suponha if(a || b)... são casos de teste para garantir cobertura
  - <(a=true, b=false), true>
  - <(a=false, b=false), false>

Teste de Condição - CC

- CC - Condicion Coverage

- Toda condição em uma expressão booleana deve ser exercitada para valores true e false.

- Condição: elemento de uma expressão booleana irredutível a expressões menores
  - if(a)
  - if(a && b) a e b são condições

Teste de Condição - C-DC

- C-CC - Condition-Decision Coverage
- Toda condição em uma expressão booleana deve ser exercitada para valores true e false.
- O efeito da valoração da condição deve afetar a decisão para true e false

Teste de Condições - C-DC (caso de teste)

- Suponha if(a && b)... são casos de teste para garantir cobertura de C-DC
  - <(a=true, b=true), true>
  - <(a=false, b=true), false>
  - <(a=true, b=false), false>

Teste de Condição - MC/DC

- MC/DC - Modified condition/decision coverage
- cada decisão assume true ou false
- cada decisão assume true ou false
- cada condição deve afetar a decisão independente das demais
- Casos de teste são compostos por pares

Teste de Condições - MC/DC (caso de teste)

- Suponha if((a || b) && c)... são casos de teste para garantir cobertura de MC/DC
  - c = (<(a=true, b=false, c = true), true>;<(a=true, b=false, c = false), false>)
  - a = (<(a=true, b=false, c = true), true>;<(a=false, b=false, c = true), false>)
  - b = (<(a=false, b=true, c = true), true>;<(a=false, b=false, c = true), false>)

Teste de Condição - RC/DC

- RC/DC - Reinforced condition/decision coverage
- cada decisão assume true ou false
- cada decisão assume true ou false
- cada condição deve afetar a decisão independente das demais
- Casos de teste são compostos por pares
- Casos de teste devem ser validos dentro de um contexto

Teste de Condições - RC/DC (caso de teste)

- Suponha if((a || b) && c)... são casos de teste para garantir cobertura de RC/DC
  - c = (<(a=true, b=false, c = true), true>;<(a=true, b=false, c = false), false>)
  - a = (<(a=true, b=false, c = true), true>;<(a=false, b=false, c = true), false>)
  - b = (<(a=false, b=true, c = true), true>;<(a=false, b=false, c = true), false>)
- Porém (a=false, b = true, c= true) é um caso de teste inválido.

Teste de Condição - M-CC

- M-CC: Toda tabela verdade deve ser explorada

Teste de Condições - M-CC (caso de teste)

- Suponha if(a || b)... são casos de teste para garantir cobertura de M-CC
  - <(a = true, b = true), true>;
  - <(a = true, b = false), true>
  - <(a = false, b = true), true>
  - <(a = false, b = false), false>

Todos os caminhos

- Todos os caminhos devem ser exercitados

Considerações sobre os Critérios de Cobertura

- Todos os caminhos: complexo e na prática inviável a não ser para programas triviais com loops simples ou sem loops.
- M-CC: simples mas complexo em termos de custo de geração de casos de teste o que o torna inviável para programas complexos.
- RC/DC: complexo mas eficiente do ponto de vista da geração de casos de teste
- MC/DC: complexo mas eficiente do ponto de vista da geração de casos de teste
- C/DC: simples eficiente na geração de casos de teste
- CC: simples eficiente na geração de casos de teste
- DC: simples eficiente na geração de casos de teste

Vantagens do Teste Estrutural

- Auxília na produção de casos de teste.
- Permite identificar códigos frios.
- Permite identificar códigos maliciosos.
- Faz uma análise mais fina do código.
- Fornece critério de parada.
- Fornece métrica de avaliação da cobertura.

Desvantagens do Teste Estrutural

- Seu uso não se aplica a todas as fazes de desenvolvimento de uma aplicação
- Deve ser usado como suporte ao teste funcional
- Por si só não identifica defeitos em uma aplicação
- Altos valores de cobertura não necessariamente significam maior localização de erros

Exercício

1. Casos de teste que satisfazem o CC satisfazem o DC? Justifique.
2. Casos de teste que satisfazem o MC/DC satisfazem o RC/DC?
3. Suponha a decisão (a && b && c && d) quantos casos de teste são necessários para satisfazer o M-CC?

Exercício

1. Gere os casos de teste para o DC, CC e C-DC da expressão
   a. ((a && b) || c)
   b. Suponha
      i. if ((a && b) || c) x = (1/(f + g -2));
      ii. qual erro pode ser gerado?
      iii. os critérios de cobertura podem identificar este erro? Justifique.
      iv. Como solucionar este problema?

Tema 3

Instrutor: Alexandre L. Martins

Teste de Desempenho

Introdução

Encontrar gargalos no sistema. Muito usado em aplicações para web.

Por que fazer teste de performance?

1. Identificar problemas com expansão, identificar aumentos de custo e/ou perda de reputação corporativa.
2. Avaliação de desempenho
    a. verificar se aplicação opera nos limites especificados
    b. verificar tempo de resposta
3. Avaliação de Infraestrutura
    a. Avalia a capacidade atual
    b. Avalia o efeito de alterações
4. Avaliação de aumento de performance(tuning)

Tipos de Teste de Performance

- Teste de Performance: valida os limites especificados
- Teste de Carga:
  - picos
  - Endurence Test (limites ao longo do tempo)
- Teste de Stress
  - abaixo e acima do normal
  - Spike Testing: abaixo ou pico em curtos intervalos de tempo
- Teste de Capacidade
  - tratar alta carga sem perda de performance

Relação

- É rápido o sufuciente? TP
- Aguenta quanto? TC
- Se algo der errado? TS

Benefícios e Limites

- T de Performance
  - determina velocidade, escalabilidade e estabilidade.
  - determina satisfação do cliente.
  - identificar erros entre performance e especificação.
  - suporte para tuning, planejamento e otimização.

- Limite
  - não identifica erros durante situações extremas.
  - deve ser feito no ambiente de execução real.
  - pode ser realizado apenas com aplicação pronta.

Benefícios e Limites

- T de Carga
  - identifica o throughput necessário para superar picos
  - adequa hardware e software
  - identifica situações de concorrência
  - identifica falhas em situação de alta carga
  - identifica número de usuários com os quais a aplicação pode tratar
- Limites
  - resultados só podem ser avaliados em relação a outros teste de carga

Benefícios e Limites

- T de Stress
  - identifica possíveis corrupções de dados
  - identifica até onde uma aplicação pode ir antes de falhar
  - identifica efeitos colaterais em hardware quando da ocorrência de uma falha
  - segurança contra ataques via sobrecarga
- Limites
  - difícil saber quanto stress deve ser aplicado
  - o teste pode afetar outros sistemas

Benefícios e Limites

- T de Capacidade
  - determina quanta carga o sistema suporta de forma "confortável"
  - permite fazer planejamentos
  - determina se o desempenho está de acordo com a aplicação
- Limites
  - casos de teste são difíceis de determinar(caso médio)

Risco Velocidade

- aplicação é rápida o suficiente?
- aplicação é rápida o suficiente para processar, armazenar e acessar dados?
- com qual velocidade a aplicação se recupera de um erro?
- aplicação tem capacidade de apresentar a informação mais atual?

Risco de Escalabilidade

- aplicação pode armazenar de forma confiável todos os dados ao longo de sua vida?
- aplicação é segura quando em alta carga?
- funções se mantém quando em alta carga?

Risco de Estabilidade

- aplicação opera por longos períodos?
- quando a aplicação volta ao "ar" ela mantem suas funções?
- pode ocorrer um system crash? (concorrência)
- alguma transação pode gerar um efeito colateral?

Tema 4

Instrutor: Alexandre L. Martins

Teste em Programas Orientados a Objeto

Instrutor: Alexandre L. Martins

Introdução

- Programas Procedurais
  - Resolvem um problema via construção de funções ou procedimentos que tratam dos dados.
  - Os dados não estão isolados das estruturas de manipulação.
- Orientado a Objeto
  - Dados e estruturas de manipulação estão isolados (encapsulados).
  - Herança.
  - Polimorfismo.
  - Acoplamento dinâmico.

OO e teste de Software

- OO não eliminou erros de software.
- amenizou problemas relativos a acessos indevidos de dados (encapsulamento).
- criou uma nova modalidade de erros (erros de interface).

Elementos

- Classe
- Objeto
  - Atributo
  - Método

Conceitos Básicos de OO

- Programa é uma coleção de objetos
- Objetos trocam mensagens
- Objeto é uma instância de uma classe
- Classe é um elemento de uma estrutura hierárquica de classes

Componete

- Criados em qualquer linguagem
- OO potencializa a idéia de DBC (Desenvolvimento Baseado em Componente)
- Componente, Unidade coesa utilizada em um dado contexto, ativada via interface e pode ser reutilizada

Falhas em POO

- Definição de métodos e atributos
- Erros de hierarquia (acesso a métodos e atributos indevidos)
- Reuso de componente
- Interface
- POO não se comente menos erros apenas se comete outros erros

Problemas de teste em POO

- encapsulamento(ex. método “private”) - Todos os atributos são “private”. Solução: implementar “get” e “set” para todos os métodos ou uso de reflexão quando a linguagem permitir.

Problemas de teste em POO

- herança: mal uso dos atributos herdados
  - mudança de uma classe na hierarquia exige reteste
- herança múltipla: qualquer alteração em uma das classes pai exige novos testes na classe filho.
- herança repetida: classe A é pai da classe B e C. Classe D é filha de B e C.

Problemas de teste em POO

- classe abstrata: só pode ser testada quando uma classe concreta a especializa.
- polimorfismo
  - indecibilidade no acoplamento dinâmico (método A.x é sobrescrito na subclasse B, versão polimórfica deve ser testada)
  - extensibilidade de hierarquia (não é possível gerar um único T para todos os polimorfismos)

Fases de Teste

- POO:
  - unidade: teste de cada método individualmente,
  - de classe: teste de integração de métodos de uma classe
  - de integração: teste entre classes do sistema
  - de sistema
- POO maior ênfase no teste de integração pois métodos devem ser simples, em teoria.

Fluxo de Dados

- permite visualizar com mais clareza as interações entre os elementos do programa.
- permite derivar caso de teste com base no critério de definição e uso.

```java
package DefUse;

import java.io.BufferedReader;

public class DefUse {

    public static void main(String args[]){

        String str = null;
        File arq = new File("teste.txt");
        try {
            FileReader     fr = new FileReader(arq);
            BufferedReader br = new BufferedReader(fr);

            while(br.ready()){

                str = br.readLine();
                System.out.println(str);

            }

        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();

        }
    }

}

```

```java
package DefUse;

import java.util.Stack;

public class ExDefUse {

    public static void main(String args[]){

        Stack<String> st = new Stack<String>();

        if(!st.empty()){
            if(st.contains("oi")){
                System.out.println("OK");
            }
        }

    }
}

```

```java
package DefUse;

import java.util.Stack;

public class ExDefUse {

    public static void main(String args[]){

        Stack<String> st = new Stack<String>();

        if(!st.empty()){
            st.add("2");
        }else{
            st.add("1");
        }

        try{
            st.remove(0);
        }catch (Exception e) {
            // TODO: handle exception
            e.printStackTrace();
        }

    }
}
```

Vantagens do critério de def/use

- geração de casos de teste
- critério de cobertura
- critério de parada

Limites

- polimorfismo e acoplamento dinâmico

Estrtégia de teste em POO

1. Partindo da superclasse
    i. testar cada método individualmente
    ii. testar sequência de chamadas de métodos(intra e inter)
    iii. gerar log de teste
        1. casos de teste
        2. falhas localizadas
2. Testar subclasse
    a. critério de teste baseado no log da superclasse
    b. reaproveitar casos de teste quando possível

Tema 5

Instrutor: Alexandre L. Martins

Geração de Malha de Teste

Instrutor: Alexandre L. Martins

Introdução

- Testar todos os elementos de D(P) é desejável mas inviável.
- Geração de Dados de Teste
  - Após o particionamento, qual elemento de cada partição selecionar?
- Não há algoritmo que permita definir se um caso de teste é adequado a um requisito. Indecidível.

Problema

- Correção Coincidente: um caso de teste exercita um caminho errado mas a saída é correta.
  - Ex. P é um programa que calcula a potência entre x e y. P possui uma linha, x*y. Caso de teste <(2,2), 4>. Programa errado saída correta.
  - Situação muito rara.

Problema

- Caminho Ausente: uma funcionabilidade não implementada representa um caminho ausente.
  - teste estrutural não identifica
- Caminho não-executável: não há caso de teste que o execute
  - ex. if(a > 10){ if(a < 5){....}}
- Mutante equivalente: impossível gerar casos de teste que os identifique

Técnicas de Geração

- Aleatória
- Simbólica
- Teste Evolutivo

Aleatória

- Mais simples
- Mais barata
- Extremamente sensível a singularidades ou a espaços amostrais não homogênios.
- Não garante satisfação de critério
- Não localiza elementos não executáveis
- Não identifica mutantes equivalentes

Simbólica

- Técnica deriva expressões algébricas que representam a execução de um dado caminho.
- Resultado da execução simbólica é uma expressão na qual as variáveis de saída são representadas em termos das variáveis de entrada.

Limites da Execução Simbólica

- Laços: número de interações finitos e determinável.
- Variáveis Compostas: não é possível identificar a qual posição do vetor a variável simbólica se refere
  - ex. a[i] < a[j]
- Chamada de Função ou Procedimento: não é possível representar em uma expressão o resultado de uma chamada.

Geração com algoritmo genético

- Teste Evolucionário
- Algoritmo Genético
  - Método de busca e otimização
  - Teoria da Evolução de Darwin
- Aplicar a geração de caso de teste os mesmos princípios da evolução das espécies de Darwin (seleção natural)

Construção da representação

- Representar o espaço de busca do problema e as possíveis soluções de forma computacional (estrutura, tipo etc.)
- Definir uma função de avaliação(fitness)

Aplicação do Algoritmo Genético

- Possíveis soluções devem ser agrupadas em um conjunto denominado população
- Cada solução na população é um indivíduo
- As soluções são aplicadas ao problema. A função fitness avalia as respostas dadas por cada solução. Quanto melhor a resposta melhor a avaliação.
- Soluções com melhor respostas são combinadas para gerar novos indivíduos
- Novos indivíduos são agregados a população
- Indivíduos com avaliações ruins são eliminados

Tema 6

Instrutor: Alexandre L. Martins

Automação de Teste

Instrutor: Alexandre L. Martins

Introdução

- processo de automatizar o teste e a análise dos resultados
- depende da técnica a ser usada (ex. teste de cobertura exige o uso de uma ferramenta)
- a construção de uma ferramenta: custo benefício
- preparo para o uso da ferramenta
- ferramenta de uso geral
- ferramenta específica

Idéia Geral

- tarefa repetitiva tem grande potencial para ser automatizada
- tarefa de cunho intelectual deve ser suportada por ferramentas mas não automatizada

Métricas Estáticas de Complexidade e Qualidade

- LOC., linhas de código
- eLoc., linhas de código executáveis
- DIT, profundidade da árvore de herança
- NOC, quantidade de filhos
- CBO, acoplamento entre classes
- LCOM, falta de coesão entre os métodos
- Complexidade ciclomática (mais usada): não depende do tamanho do código mas do número de ramos na estrutura de controle

Ferramentas de Suporte para Análise

Ferramentas que detectam erros em especificações formais.

Tema 7

Instrutor: Alexandre L. Martins

TDD - Test-Driven Development

Introdução

O teste é escrito antes do código.

Idéias

- Kiss: Keep it simple, stuped (KISS)
- "You ain`t gonna need it" (YAGNI)

Ciclos

1. Definição dos casos de teste
   a. antes de escrever definir o caso de teste
   b. entender muito bem a especificação(caso de uso ou user stories)
2. Execução
   a. execute todos os teste
   b. o teste novo gera erro. Por que?
3. Codificação
   a. se todos os testes passarem escreva o código
   b. senão retorne a versão anterior
4. Refatore
5. Repita o processo

Vantagem

- programadores entendem melhor o programa.
- aplicação é mais simples(passar no teste)
- sem depuração
- singularidades e limites são testados em separado
- alta modularidade

Limite

- Necessita ser bem gerenciado
- Cobertura de falha é fraca
- erros na formulação do caso de teste serão repassados para o código
- foco no teste de unidade pode levar a um menosprezo do teste de integração e sistema.

Geração de Malha de Teste

Instrutor: Alexandre L. Martins

Introdução

- Testar todos os valores da entrada de P é impossível na prática.

- Não há algoritmo de propósito geral que determine se um caso de teste é adequado ou não.

Problema na Geração de Caso de teste

- Correção coincidente(raro): caminho errado resultado correto(x*y e x^y)
- Caminho ausente: funcionabilidade não implementada
- mutantes equivalentes

Técnicas de Geração

- Aleatória
- Simbólica
- Algoritmo genético

Simbólica

- Técnica deriva expressões simbólicas
- Resultado da expressão é uma expressão na qual as variáveis de saída são representadas em termos das variáveis da entrada

Limetes da Exceução simbólica

- Laços
- Variáveis compostas
- Chamadas de Função

Tema 8

Instrutor: Alexandre L. Martins

Gerenciamento de Teste

Processo de Qualidade

- Qualidade é uma meta a ser atingida
- Questão,

Alta Confiança X Alto Custo

Processo de Qualidade Ideal

- falhas são identificadas muito cedo
- real, falhas são identificadas conforme a abordagem do teste

Relação custo benefício da detecção

- O momento em que uma falha é identifica define o seu custo no sistema
  - ex.falha de implementação teste unitário, baixo custo
  - falha de implementação na fase de teste de sistema, auto custo na correção

Planejamento e Monitoração

- Grau de visibilidade
  - processo está de acordo com as metas
  - cronograma
- Certificações
  - ISO9000
  - SEI CMM
  - DO-178B
  - Definem como processo deve ser estruturado
    - ferramentas
    - sequência de teste
    - avaliação de teste
    - processos caros

Plano de Análise

Descreve o processo de qualidade e inclui

1. objetivo e escopo
2. documentação
3. identificar os itens a serem testados
4. selecionar a atividade de teste
5. identificar equipes

Metas de qualidade

- 100% de qualidade é o mesmo que não definir meta
- Sempre se deve explicitar quais metas se deseja atingir
  - ex. é implícito que se deseja um sistema com baixo número de falhas mas qual é o valor exato de falhas aceitáveis?

Métricas para medir confiança

- disponibilidade, tempo em que o sistema opera sem falha
- tempo entre falhas, o tempo decorrido entre duas falhas

Segurança

- critério de classificação de falhas
- leva em conta o efeito da falha e não somente seu caráter funcional
- Sistemas críticos
- Sistema pode ser seguro e mesmo assim não ser confiável

Monitoramento

- Número de falhas localizadas e removidas em função do tempo
  - número de falhas diminui com tempo
  - sistema não possui mais falhas ou sistema é ineficiente
- Quantidades de falhas leves ou moderadas pode ser um bom indicador de qualidade

ODC(classificação ortoganal de defeitos)

1. classifica a falha
2. analisa a causa
3. taxonomia de falhas

Documentação

- Aumenta o reuso de casos de teste
- armazena o conhecimento acumulado
- monitora e avalia testes
- ajudam a melhorar o processo de teste

Tipos de Documentos

- Planejamento(organização do teste)
- Especificação(conjuntos de teste e casos de teste)
- Relatório(análise de resultados)

FIM

Bibliográfia

Foundation of Software Testing, A.P. Mathur

Introdução ao Teste de Software. M. E. Delamaro

Teste e Análise de Software, Mauro Pezzè

Teste Orientado a Falhas

Instrutor: Alexandre L. Martins

Introdução

Baseado na criação de hipóteses sobre possíveis falhas que podem ocorrer em um sistema. A fim de avaliar a eficácia de um conjunto de casos de teste para identificar estas falhas.

Por que estudar falhas?

- Definir causas
- Prevenir ocorrência
- Criar normas
- Melhorar linguagens de programação
- Melhorar técnicas de programação

Exemplo

- Garbage Collector do Java
  - criado para evitar falhas de manipulação de memória
- Tratamento de exceções
  - criado para permitir que um software se recupere de uma falha
  - alguns conjuntos de falhas já estão pré definidos e são verificadas pelo compilador.

Conceitos Centrais

- Casos de Teste
  - diferenciam o programa original (PO) dos programas mutantes (PM)
- Programa Mutante
  - programa que possui pequenas diferenças em relação ao progrma original
- Geradores de Falhas
  - medem a eficácia da malha de teste

Hipóteses

- Programador Competente: apenas poucos e pequenos erros são necessários.

- Efeito Acoplamento: grandes falhas são compostas de pequenas falhas. Logo, a inserção de pequenas falhas pode revelar quandes falhas.

Na prática...

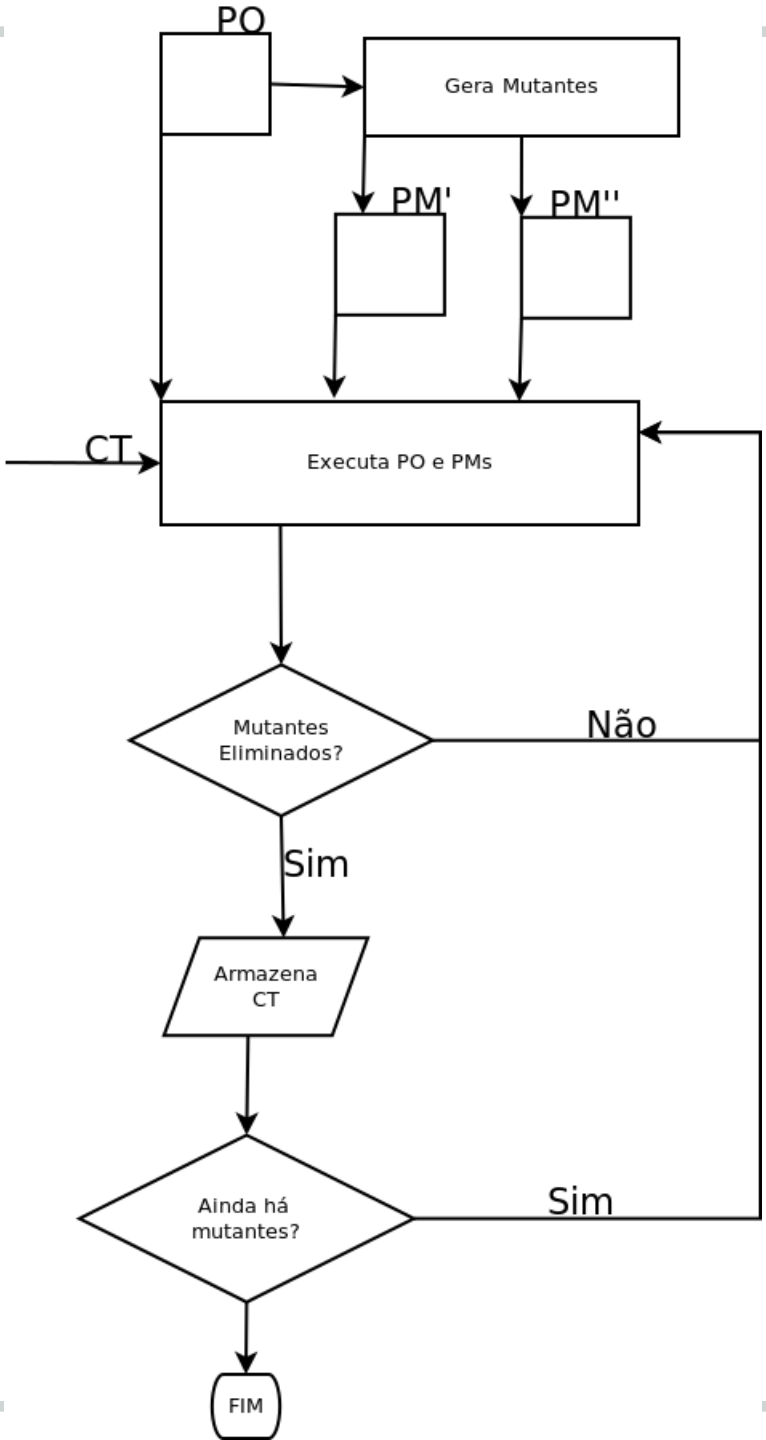
- pequenas variações sintáticas são criadas
- operadores de mutação definem quais alterações
  - trocar constantes C_1 por C_2 para C_1 diferente de C_2
  - não inicializar variáveis
  - inserir variáveis absolutas abs(a) para a < 0
  - Substituir operadores aritméticos
  - Substituir conectores lógicos
  - Substituir operadores de relacionamento
  - Remover comandos

Terminologia

- Programa Original: programa a ser testado
- Programa Mutante: que possui pequena variação sintática em relação ao programa original
- Mutante Distinguível: difere do programa original por pelo menos uma execução de caso de teste
- Mutante equivalente: mutação não afeta funcionabilidade
- Mutante Valido: sintaticamente correto
- Mutante Util: é valido e difere do programa original

Processo de Análise Mutante

1. Seleção dos operadores
   a. devem ser relacionados ao tipo de falha que se deseja estudar
2. Geração Mutante
   a. Aplicar os Operadores ao código
3. Matar Mutantes
   a. Para cada caso de teste executar o PO e o PM. Se ocorrer diferença de execução matar o mutante e armazenar o caso de teste
   b. Mutantes que não morrem
      i. malha de teste deve ser melhorada
      ii. PO = PM
      iii. PM é equivalente a PO


PO
Gera Mutantes
PM'
PM''
CT
Executa PO e PMs
Mutantes Eliminados?
Não
Sim
Armazena CT
Ainda há mutantes?
Sim
FIM

Adequação da Malha de Teste

Quanto mais mutantes mortos melhor a malha de teste.

Mutação Forte e Fraca

- Forte
  - Gera muitos mutantes
  - Combina cada mutante com cada caso de teste
  - Execução completa

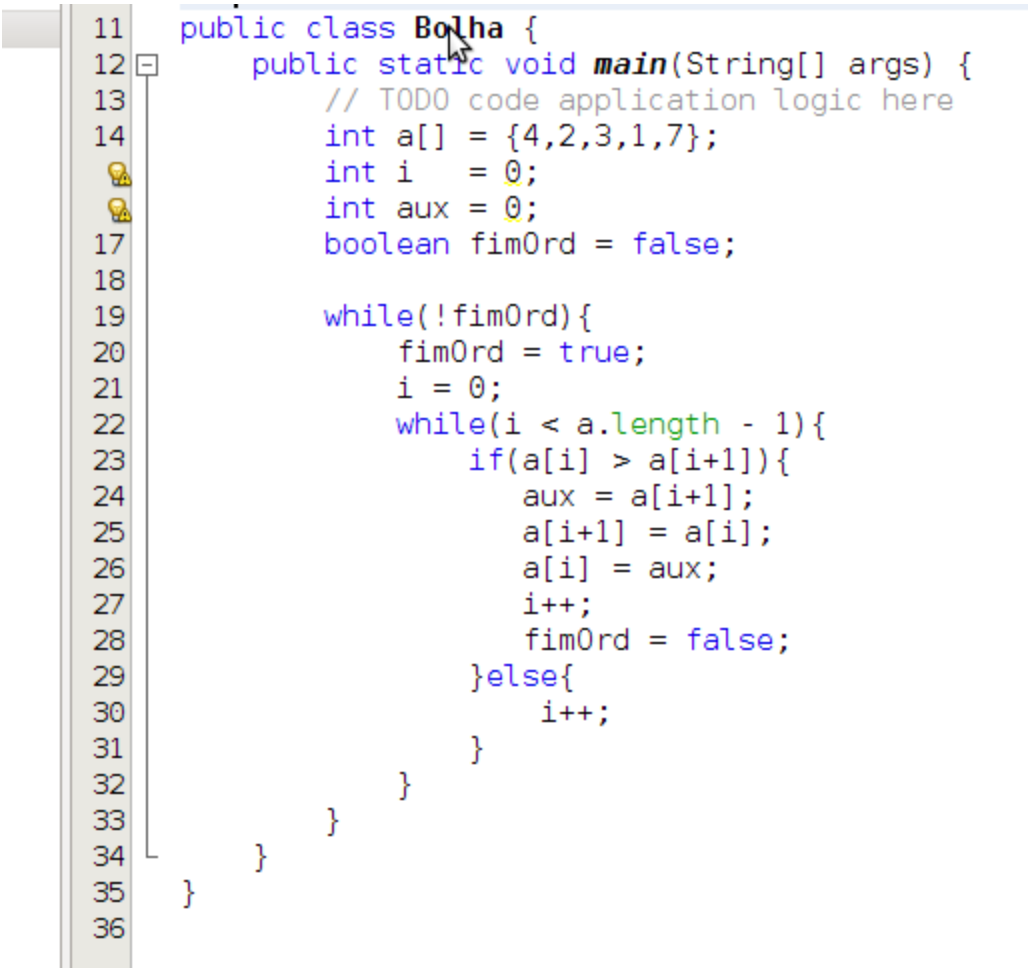
- Fraca
  - mutante é morto quando gera um estado intermediário diferente do estado gerado em PO
  - Menos custosa
  - Não reduz o número de execuções

Considerações Finais

- Teste baseado em falha tem uma idéia simples
- Permite refinar o conjunto de casos de teste
- Geração de mutantes é simples
- Porcesso de eleiminação é custoso devido ao número de execuções

Exercício

1. Gere 5 mutantes para a listagem do programa Bolha. Justifique as mutações.
2. Determine quais casos de teste matam esses mutantes.
3. Para listagem gere:
   a. um mutante inválido
   b. um mutante inútil
4. Mutação fraca mata um mutante quando este chega a um estado incompatível com o programa original. Após a morte do mutante a execução do programa original retorna ao estado inicial. Por que? O que se deve fazer para que a execução não seja reiniciada? Isso melhora o processo?



```
11  public class Bolha {
12      public static void main(String[] args) {
13          // TODO code application logic here
14          int a[] = {4,2,3,1,7};
            int i   = 0;
            int aux = 0;
17          boolean fimOrd = false;
18
19          while(!fimOrd){
20              fimOrd = true;
21              i = 0;
22              while(i < a.length - 1){
23                  if(a[i] > a[i+1]){
24                     aux = a[i+1];
25                     a[i+1] = a[i];
26                     a[i] = aux;
27                     i++;
28                     fimOrd = false;
29                  }else{
30                      i++;
31                  }
32              }
33          }
34      }
35  }
36
```

Teste Orientado a Erro

Instrutor: Alexandre L. Martins

Introdução

Uso de modelo para criar falhas hipotéticas que podem ocorrer em software. Com base nestas hipóteses são gerados casos de teste.

Por que estudar falhas?

- Definir causas,
- Prevenção,
- Criar normas,
- Melhorar linguagens e
- Classificar.

Exemplo

- Java Garbage Collector
  - Criado para evitar falhas de manipulação de memória
- Tratamento de exceções

Idéia

- Baseado no modelo de falhas. Cópias de um programa são feitas introduzindo pequenas falhas.
- Casos de teste devem diferenciar os programas com falha do original.
- Gerador de falhas

Premissas

- Qualidade do modelo
- As falhas podem realmente acontecer no sistema

Tipos de falha

- São relativas a linguagem da implementação
- São simples e de cunho sintático
- Não podem afetar a compilação do programa
- Hipóteses do Programador Competente e efeito acoplamento.

Hipótese do Programador Competente

Apenas pequenos erros são introduzidos pelo programador pois ele é competente.

Hipótese do efeito acoplamento

- Grandes falhas são compostas por pequenas falhas.
- Logo, casos de teste que revelam pequenas falhas também identificam grandes falhas.

Terminologia

- Programa Original: o que se deseja testar.
- Localização do Programa: região do código fonte onde é inserida a falha.
- Expressões substitutas
- Programa Substituto
- Comportamento distinto
- Comportamento distinguível

Análise Mutante

- Mutante: Programa Substituto com falha viável.
  - mutante não é recusado pelo compilador (Mutante Válido)
  - mutante que falha para todos os testes deve ser descartado
- Mutante Útil: seu comportamento difere do original por um pequeno conjunto de casos de teste.

Operadores de Mutação

- Critério para definir qual alteração será feita
- Definição dos operadores é não trivial
- Vinculado a linguagem da aplicação

Exemplo de OM

- Substituição de constante por constante
- Eliminação da inicialização de uma variável
- Inserir variáveis absolutas, abs(e) se e < 0
- Substituir operadores aritméticos
- Substituir conectores lógicos(&& por || ou && por &)
- Substituir operadores relacionáis (< por <=)
- Remover comando

Processo de Análise Mutante

1. Seleção dos operadores de mutação
   a. OM devem ser relacionados com o tipo de falha que se quer estudar
2. Geração dos mutantes
   a. aplicar os operadores ao código.
3. Matar Mutantes
   a. Para cada caso de teste executar o programa original e o mutante.
   b. Se execuções forem iguais eliminar caso de teste
   c. Se diferentes eliminar o mutante e armazenar o caso de teste

Se o mutante não morre?

1. Geração de casos de teste é ineficiente
2. Mutante e programa original são funcionalmente equivalentes
3. Mutante e programa são iguais

Adequação da Malha

Quanto mais mutantes mortos melhor a qualidade da malha.

Exercício

1. Análise mutante permite identificar ausência de caminhos?

2. Análise mutante pode ser classificada como um tipo de teste estrutural? Justifique.

3. Há relação entre a análise mutante e o teste funcioanal? Qual?