**Teste Funcional**

**Panorama Geral das Ferramentas de Apoio**

Considerando que a atividade teste de software ela pode ser automatizada, existem ferramentas de automatização relacionadas com a técnica chamada teste funcional.

**O teste funcional ele considera a especificação do software em si, o foco não é no código**.

Os softwares eles estão cada vez mais complexos, e o número de interações que o usuário pode fazer com a interface gráfica, por exemplo, com o software em si é muito grande. Tentar gerar o mínimo de casos de testes possíveis para serem executados, bons casos de testes, mas que sejam capazes de cobrir as partes mais importantes do software que o usuário interage.

Então nesse momento ferramentas podem ser utilizadas, ferramentas que tentam selecionar e projetar os casos de teste considerando critérios já conhecidos da técnica de teste funcional, por exemplo, particionamento caso de equivalência, análise do valor limite e também ferramentas que tentam executar esses casos de teste, gerenciar os casos de teste.

Então é possível encontrar, por exemplo, ferramenta onde o usuário interage com o software, essa ferramenta de automatização grava essa interação, depois gera scripts que podem ser reexecutados quantas vezes forem necessários. Ou então, por exemplo, ferramentas que simulam a interação do usuário com o software que está sendo testado, geram os scripts que também podem ser armazenados, gerenciados e repetidos. E outra questão, são ferramentas que executam e gerenciam os casos de teste si.

É possível encontrar ferramentas onde são exercitadas partes do código mas que a chamada ela

exige uma pequena implementação, para que entradas sejam enviadas, saídas sejam retornadas, e aí possa se verificar se a saída devolvida estava de acordo com a saída que era esperada.

**Teste Funcional e Junit**

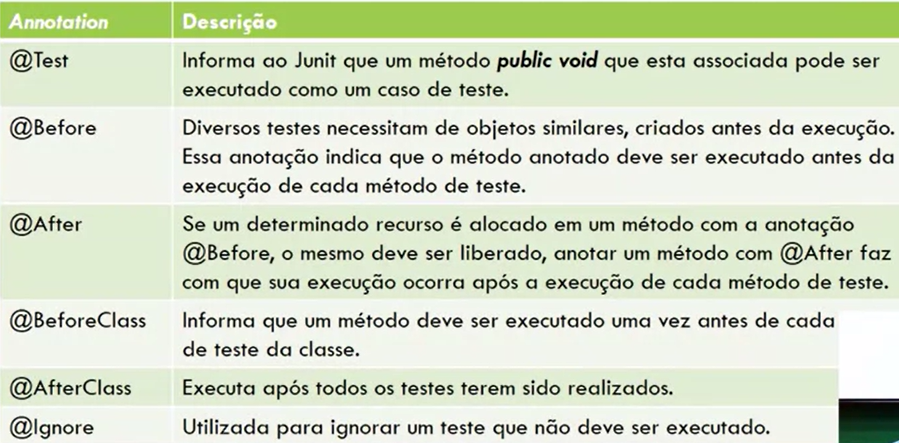
**Definição:**

“JUnite é um framework de teste de unidade para a linguagem de programação Java, proveniente de uma família de frameworks de Teste de Unidade denominada de xUnit”.

**Características:**

* Framework Open Source; (acesso livre ao código da ferramenta)
* Utiliza “Annotations” para identificar métodos de teste;
* Fornece “Assertions” para avaliar valores esperados;
* Possibilita a execução de testes automatizados, possibilitando feedback imediato ao testador.

**“Anotations são meta tags que podem ser adicionadas ao código”.**



**API JUNIT**

Classes fundamentais contidas no pacote “junit.framework”

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

**PASSOS PARA A APLICAÇÃO DE UM CRITÉRIO FUNCIONAL**

* Analisar a especificação de requisitos;
* Entradas válidas são escolhidas para determinar se o produto em teste se comporta corretamente;
* Entradas inválidas são escolhidas para verificar se estas sõ detectadas e manipuladas adequadamente;
* Os casos de testes são construídos (saídas são determinadas para cada entrada).
* O conjunto de teste é executado e as saídas obtidas sõ comparadas co as saídas esperadas;
* Um relatório é gerado para avaliar o resultado dos testes.

**CLASSES DE EQUIVALENCIA**

Programa Indentifier

Tabela

Descrição gerada automaticamente

**Identifier.java**

public class Identifier {

public boolean validateIdentifier(String s) {

char achar;

boolean valid\_id = false;

if (s.length() > 0) {

achar = s.charAt(0);

valid\_id = valid\_s(achar);

if (s.length() > 1) {

achar = s.charAt(1);

int i = 1;

while (i < s.length()) {

achar = s.charAt(i);

if (!valid\_f(achar))

valid\_id = false;

i++;

}

}

}

if (valid\_id && (s.length() >= 1) && (s.length() <= 6))

return true;

else

return false;

}

public boolean valid\_s(char ch) {

if (((ch >='A') && (ch <= 'Z')) || ((ch >= 'a') && (ch <+ 'z')))

return true;

else

return false;

}

public boolean valid\_f(char ch) {

if(((ch >= 'A') && (ch <= 'Z')) || ((ch >= 'a') && (ch <= 'z'))

|| ((ch >= '0') && (ch <= '9')))

return true;

else

return false;

}

}

**IdentifierMain.java**

public class IdentifierMain {

public static void main(String[] args) {

if(args.length == 0) {

System.***out***.println("Uso: IndentifierMain <string>");

}else {

Identifier id = new Identifier();

if (id.validateIdentifier(args[0])) {

System.***out***.println("Valido");

}else {

System.***out***.println("Invalido");

}

}

}

}

**IdentifierMainTest.java**

import static org.junit.jupiter.api.Assertions.\*;

import java.io.ByteArrayOutputStream;

import java.io.PrintStream;

import org.junit.AfterClass;

import org.junit.Before;

import org.junit.BeforeClass;

import org.junit.jupiter.api.Test;

public class IdentifierMainTest {

static private ByteArrayOutputStream *baOut*;

static private PrintStream *psOut*;

*@BeforeClass*

public static void beforeClassInit() {

*baOut* = new ByteArrayOutputStream();

*psOut* = new PrintStream(*baOut*);

System.*setOut*(*psOut*);

}

*@AfterClass*

public static void afterClassFinalize() {

*psOut*.close();

}

*@Before*

public void setUp() {

*baOut*.reset();

}

*@Test*

public void testeValido01() {

IdentifierMain.*main*(new String [] {"a1"});

String output = *baOut*.toString();

*assertEquals*("Valido", output);

}

*@Test*

public void testeInvalido02() {

IdentifierMain.*main*(new String [] {"2b3"});

String output = *baOut*.toString();

*assertEquals*("Invalido", output);

}

*@Test*

public void testeInvalido03() {

IdentifierMain.*main*(new String [] {"z-12"});

String output = *baOut*.toString();

*assertEquals*("Invalido", output);

}

*@Test*

public void testeInvalido04() {

IdentifierMain.*main*(new String [] {"a1b2c3d"});

String output = *baOut*.toString();

*assertEquals*("Invalido", output);

}

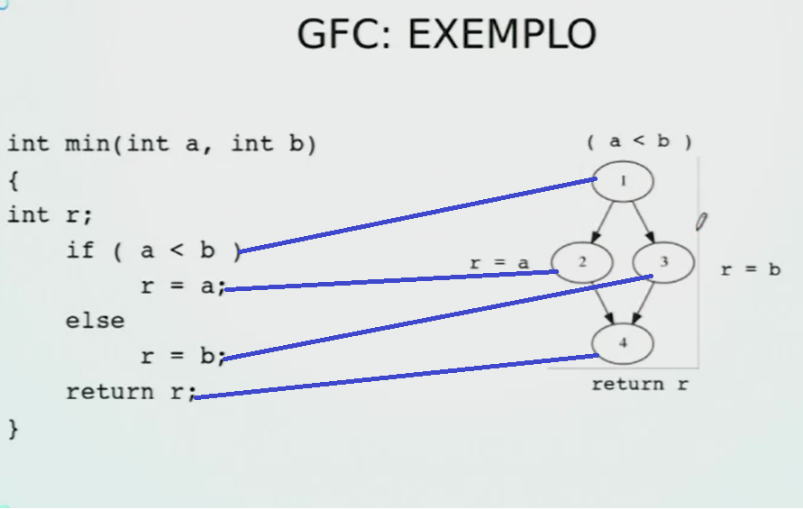
}

**Teste estrutural**

**Definição:**

“O teste estrutural usa o código do programa para definir os requisitos de teste e deles derivar os casos de teste. O objetivo é observar as “estruturas” que compõem o programa e garantir que todas elas tenham sido executadas durante o teste.”

**Grafo Fluxo de Controle**

****

**Texto preto sobre fundo branco

Descrição gerada automaticamente**

**GFC – Programa Identifier**

**Diagrama

Descrição gerada automaticamente**

* O teste estrutural, ao contrario do funcional, tem como objetivo exercitar os comando do nosso programa;
* Para abstrair a estrutura dos comandos usamos o Grafo de Fluxo de controle.
* O Grafo de Fluxo de Controle permite que se abstraia os comandos e o fluxo de execução do programa.

**CRITÉRIOS: FLUXO DE CONTROLE E DE DADOS**

**Fluxo de controle:**

* Significa que teremos que achar casos de teste que exercitem os elementos do GFC;
* Todos os NÓS; (sejam executados pelo menos uma vez)

Pode ser utilizado para verificar, cobrir, ou executar as arestas do Grafo. Também pode ser chamado de Cobertura de ramo ou Cobertura de desvios.

* Todos os ARCOS; (executar todos os desvios de execução do programa)

Também chamado de Teste de Cobertura de comando.

**Fluxo de Dados:**

* Definição – ponto em que uma variável recebe um valor;
* Uso – ponto em que um valor de variável é usado em um cálculo ou numa condição;
* Casos de teste de melhor qualidade;
* Conceitos mais alto;
* Muitos requisitos não executáveis.

**GCC – GNU COMPILER COLLECTION - Comandos para compilar:**

gcc -fprofile-arcs -ftest-coverage identifier.c -o identifier

identifier.c identifier.gcno

GCOVR - Visualizar o resultado

gcovr -r . --branches --html --html-details -o identifier.html

--> Lembrando que identifier é o nome do algoritmo testado.

**Teste de Mutação**

O teste de mutação é uma técnica de teste de software que envolve a introdução de pequenas alterações, chamadas mutações, no código-fonte do programa. Cada mutação representa um possível erro ou defeito no programa original. O objetivo do teste de mutação é verificar se os casos de teste existentes são capazes de detectar essas mutações, identificando assim possíveis áreas de fragilidade no código. É uma abordagem eficaz para melhorar a cobertura de teste e a qualidade do software.

Como Avaliar os Casos de Teste?

* Vamos “contar’ quantos mutantes conseguimos matar
* Total de mutantes: 110
* Mutantes mortos: 60
* Mutantes equivalentes: 10

(Um mutante equivalente é uma alteração no código fonte de um programa que não afeta o comportamento observável do programa original. Isso significa que o mutante equivalente possui a mesma funcionalidade do programa original, mas foi modificado de alguma forma. O objetivo do teste de mutação é identificar mutantes equivalentes e garantir que os casos de teste sejam capazes de detectar essas alterações sem afetar o comportamento do programa. Ao eliminar os mutantes equivalentes, o teste de mutação ajuda a aumentar a confiabilidade e robustez do software.)

Escore de mutação = MUTANTES MORTOS = 60 = 0,60

TOTAL - EQUIVALENTES 100

**Escore de mutação:** O escore de mutação avalia a qualidade de um conjunto de casos de teste.

O escore de mutação é uma métrica utilizada na análise de mutação, uma técnica para avaliar a eficácia dos conjuntos de casos de teste em detectar defeitos em um programa. O escore de mutação é calculado com base no número de mutantes mortos, ou seja, os mutantes que foram detectados e considerados "mortos" pelos casos de teste, em relação ao número total de mutantes gerados pelo processo de mutação.

Portanto, o escore de mutação não é influenciado pelos mutantes equivalentes, leva em consideração o número de casos de teste utilizados para matar os mutantes e é influenciado pelo número de operadores de mutação utilizados no teste. No entanto, sua principal finalidade é avaliar a qualidade do conjunto de casos de teste em detectar mutantes defeituosos, identificando a efetividade dos testes na detecção de erros no código.

**COMO SÃO GERADOS OS MUTANTES**

Operadores de mutação

Linguagem C = Comandos, Constantes, Operadores, Variáveis.

Existem pelo menos 75 operadores projetados para linguagem C.

Diagrama

Descrição gerada automaticamente

Diagrama

Descrição gerada automaticamente