Módulo 4

Engenharia de Software

Lição 6

Teste de Software

Versão 1.0 - Jul/2007

Autor
Ma. Rowena C. Solamo

Equipe
Jaqueline Antonio
Naveen Asrani
Doris Chen
Oliver de Guzman
Rommel Feria
John Paul Petines
Sang Shin
Raghavan Srinivas
Matthew Thompson
Daniel Villafuerte

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

requisitos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Aécio Júnior
Alexandre Mori
Alexis da Rocha Silva
Allan Souza Nunes
Allan Wojcik da Silva
Anderson Moreira Paiva
Anna Carolina Ferreira da Rocha
Antonio Jose R. Alves Ramos
Aurélio Soares Neto
Bruno da Silva Bonfim
Carlos Fernando Gonçalves
Daniel Noto Paiva
Denis Mitsuo Nakasaki
Fábio Bombonato
Fabrício Ribeiro Brigagão
Francisco das Chagas
Frederico Dubiel
Jacqueline Susann Barbosa
João Vianney Barrozo Costa
Kleberth Bezerra G. dos Santos
Kefreen Ryenz Batista Lacerda
Leonardo Ribas Segala
Lucas Vinícius Bibiano Thomé
Luciana Rocha de Oliveira
Luiz Fernandes de Oliveira Junior
Marco Aurélio Martins Bessa
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauro Cardoso Mortoni
Mauro Regis de Sousa Lima
Paulo Afonso Corrêa
Paulo Oliveira Sampaio Reis
Ronie Dotzlaw
Seire Pareja
Sergio Terzella
Thiago Magela Rodrigues Dias
Vanessa dos Santos Almeida
Wagner Eliezer Rancoletta

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

Sistemas são testados para detectar defeitos ou falhas antes de serem entregues aos usuários finais. Um fato conhecido é a dificuldade em se localizar e corrigir falhas, uma vez que o sistema esteja em uso. Nesta lição discutiremos métodos e técnicas para desenvolver casos de teste.

Estratégias de teste também serão apresentadas para que possamos entender as etapas necessárias para planejar as séries de passos que resultarão em uma construção bem sucedida do sistema.

Ao final desta lição, o estudante será capaz de:

- Utilizar métodos e técnicas no desenvolvimento de testes
- Utilizar métodos de projeto de casos de teste
- Aprender a como testar classes
- Entender o desenvolvimento voltado para testes
- Aprender a testar sistemas
- Utilizar testes de métricas

2. Introdução ao Teste de Software

Um sistema é testado para demonstrar a existência de uma falha ou defeito, pois o objetivo de se testar é descobrir possíveis erros. Em um sentido mais amplo, um teste é bem sucedido somente quando uma falha é detectada ou é resultante de uma falha no procedimento de teste. ***Identificação de Falha*** é o processo para a identificação e localização da causa da falha. ***Correção e Remoção de Falha*** é o processo de efetuar mudanças no software e no sistema para remover falhas.

Teste de Software abrange uma série de atividades com o objetivo principal de descobrir falhas e defeitos. Especificamente, tem os seguintes objetivos:

- Projetar um caso de teste com certa probabilidade de encontrar falhas ainda não descobertos;
- Executar a classe com a intenção de encontrar erros.

2.1. Princípios de Teste de Software

1. Todos os testes devem ser seguidos de acordo com os requisitos.
2. Testes devem ser planejados muito antes dos testes começarem.
3. O Princípio de Pareto aplica-se nos casos de teste. Este princípio indica que 80% de todos os erros descobertos durante os testes serão provavelmente seguidos de 20% em todos os módulos ou classes do projeto.
4. Testes começam em pequenas partes e devem progredir de forma a se atingir as grandes partes do sistema.
5. Testes devem ser realizados de forma exaustiva e da maneira mais completa possível o que possibilitará boas chances de se construir um sistema com um mínimo de falhas.
6. Para ser mais eficaz, um terceiro grupo independente (normalmente um grupo de teste de software) deve conduzir os testes.

Estratégias de Teste de Software integram métodos no projeto de caso de testes em uma série bem planejada de etapas que resultarão em uma implementação de sucesso do projeto. É um mapa do caminho para ajudar os usuários finais, desenvolvedores e grupo de garantia de qualidade a conduzir o teste. Tem como objetivo assegurar que o projeto construído execute uma função específica (**verificação**), e que esteja seguindo as exigências do cliente (**validação**).

Testes são executados por uma variedade de pessoas. **Os desenvolvedores** são responsáveis por testar unidades individuais antes de realizarem a integração destas unidades ao projeto. Entretanto, devem ter interesse em demonstrar que o código está livre de falhas. **O Grupo da Garantia de Qualidade** pode ser encarregado de executar o teste, o objetivo principal é descobrir o máximo de falhas possível e encaminhar para que os desenvolvedores corrijam os erros descobertos.

Os testes começam com um único componente utilizando as técnicas da **caixa-branca** e **caixa-preta** para produzir os casos de teste. Para o desenvolvimento orientado a objeto, o bloco de construção é a classe. Uma vez que a classe seja verificada, a integração da classe através de comunicação e colaboração com outras classes também serão analisadas. Podem ocorrer instâncias nos quais o subsistema de um teste será executado novamente para assegurar que a correção da falha não gere erros adicionais ou novos erros. Finalmente, o sistema é testado como um todo.

A fim de se ter uma maneira organizada de testar o projeto, uma especificação de teste precisa ser desenvolvida, normalmente, pelo líder do grupo de garantia de qualidade. Se não houver nenhum grupo de garantia de qualidade, o líder do projeto deve formular um. ***Especificação de Teste*** é uma estratégia completa que define testes específicos para serem conduzidos. Inclui também os procedimentos sobre como conduzir o teste. Pode ser utilizado para seguir o progresso do grupo de desenvolvimento e definir os caminhos do projeto.

3. Métodos do Projeto do Caso de Teste do Software

Existem duas maneiras de se testar um projeto. Testar o funcionamento interno do projeto, Verificar se as operações internas executam como especificado, e se todos os componentes internos foram adequadamente testados. E testar o projeto como um todo, isto é, saber como o sistema irá trabalhar e testar se está conforme as funcionalidades especificadas nos requisitos. A primeira maneira representa a **caixa-branca** e a segunda representa o teste da **caixa-preta**.

3.1. Técnicas do Teste de Caixa-Branca

Também conhecido como **teste da caixa de vidro**. É uma técnica de projeto de caso de teste que utiliza a estrutura interna do controle do componente como foram definidos seus métodos para produzir os casos de teste. Seu objetivo é assegurar que operações internas executem de acordo com as especificações do componente. Garantir que nenhum erro lógico, compreensões incorretas ou erros tipográficos foram cometidos. Os casos de teste devem:

- Testar os trajetos independentes do componente para que sejam verificados ao menos uma vez.
- Testar as decisões lógicas para ambas condições, verdadeiro e falso.
- Testar os laços em seus limites e dentro de seus limites operacionais.
- Testar as estruturas internas de dados para sua validação.
- Testar os trajetos nos componentes que são considerados "fora do objetivo final".

Existem diversas técnicas que podem ser empregadas na definição de casos de teste usando o método da caixa-branca.

3.1.1. Teste Básico do Trajeto

É a técnica do teste da caixa-branca que permite que o projetista do caso de teste produza uma medida lógica da complexidade baseada na especificação processual de um componente de projeto. Esta medida da complexidade é utilizada como um guia para definir a base da série dos trajetos de execução que serão testados.

Etapas para Produzir Casos de Teste utilizando o Teste Básico do Trajeto

ETAPA 1. Utilizar uma especificação do procedimento como entrada para produzir trajetos básicos para execução do trajeto.

A especificação do procedimento pode ser elaborada em pseudo-código, fluxograma ou código-fonte próprio. Para classes, o projeto processual da operação ou do método será usado. Como exemplo, suponha que o pseudo-código a seguir seja algum método de alguma classe.

```
while (condition1) do
  statement1;
  statement2;
  do case var1
    condition1:
      statement3
    condition2:
      statement4;
      statement5;
    condition3:
      if (condition2) then
        statement6;
      else
        statement7
        statement8
      endif
  endcase
  statement9;
endwhile
```

ETAPA 2. Desenhar o Gráfico do Fluxo da Especificação Processual.

O **gráfico do fluxo** descreve o fluxo lógico do controle da especificação processual. Usa *nodes*, bordas e regiões. Os ***nodes*** representam uma ou mais instruções. Pode ser traçado por sequências ou condições. As **bordas** são as ligações dos *node*s. Representam um fluxo de controle similar ao gráfico do fluxo. As **regiões** são áreas limitadas pelas **bordas** e pelos ***nodes***. Figura 1 mostra um exemplo de código procedural do gráfico de fluxo. O gráfico normalmente não mostra o código. É colocado para finalidade de esclarecimentos.

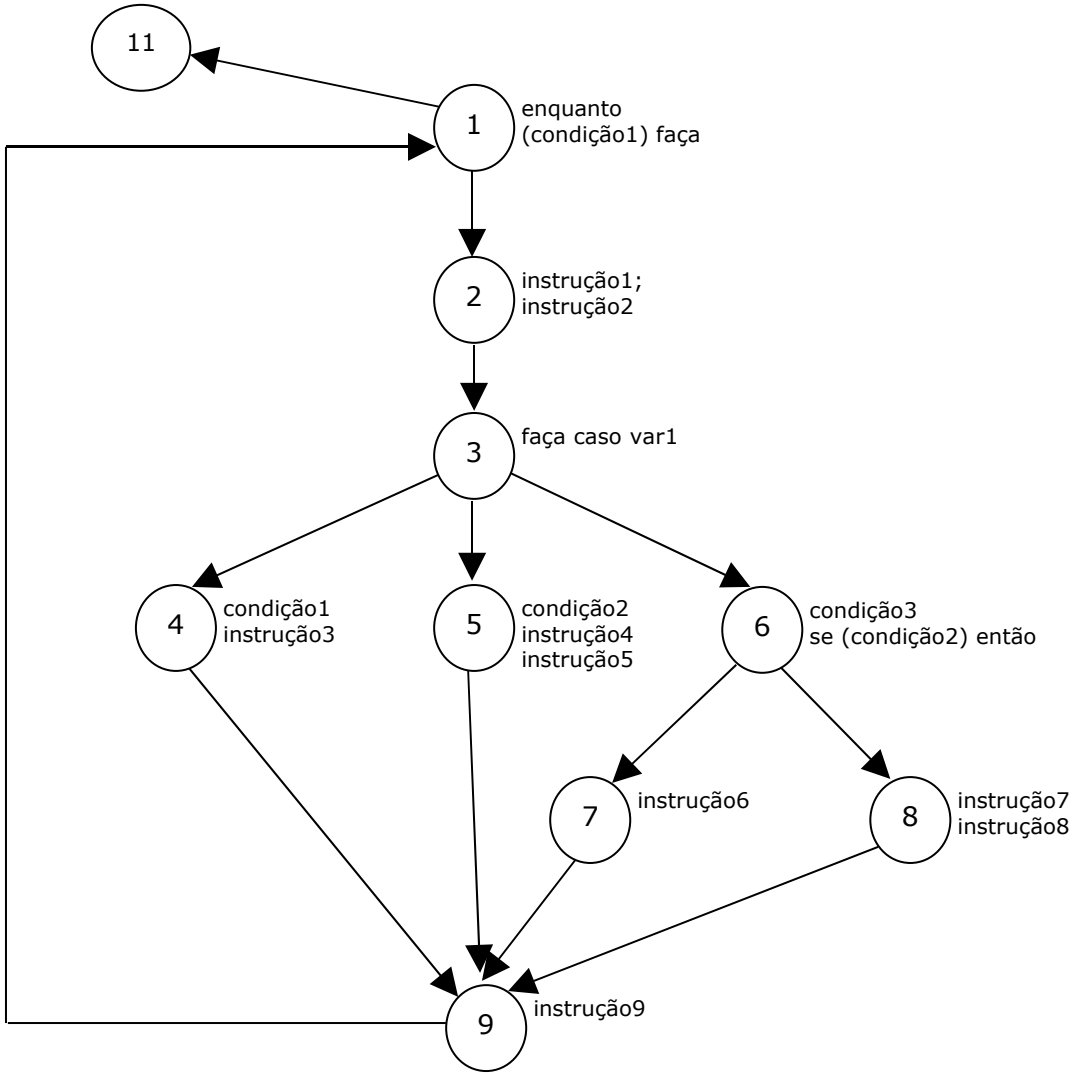

Figura 1: Gráfico de Fluxo

ETAPA 3. Calcular a Complexidade do Código.

A **complexidade ciclomática** é utilizada para determinar a complexidade do código processual. É um número que especifica trajetos independentes no código, que é a base do caminho. Pode ser calculado de três maneiras.

1. O número de regiões do gráfico de fluxo.
2. O número de predicados dos *node*s mais um, isto é, V(G) = P + 1
3. O número de bordas menos os *node*s mais 2, isto é, V(G) = E – N + 2

No exemplo:

O número de regiões é 5.

O número de predicados é 4.

O número de bordas é 13.

O número de *node*s é 10.

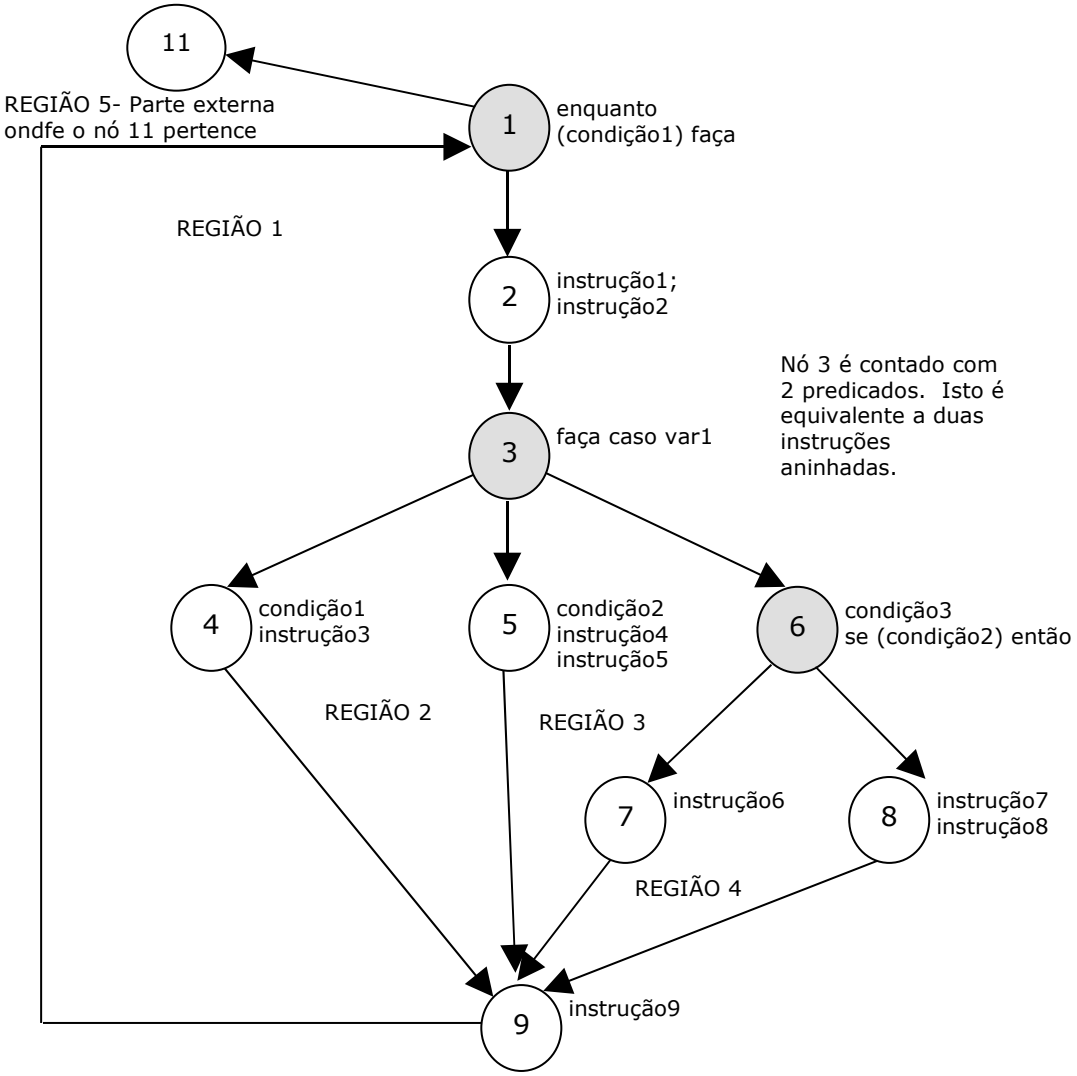

Figura 2: Valores da complexidade ciclomática

Utilizando a fórmula:

- V(G) = 5 (regiões)
- V(G) = 4 (predicados) + 1 = 5
- V(G) = 13 (bordas) – 10 (*node*s) + 2 = 5

ETAPA 4. Determinar caminho básico de trajetos da execução.

A complexidade ciclomática fornece o número de trajetos independentes no código, linearmente. No exemplo acima, 5 trajetos lineares independentes são identificados.

Trajeto 1: *node*-1, *node*-2, *node*-3, *node*-4, *node*-9

Trajeto 2: *node*-1, *node*-2, *node*-3, *node*-5, *node*-9

Trajeto 4: *node*-1, *node*-2, *node*-3, *node*-6, *node*-8, *node*-9

Trajeto 3: *node*-1, *node*-2, *node*-3, *node*-6, *node*-7, *node*-9

Trajeto 5: *node*-1, *node*-11

ETAPA 5: Documentar os casos de teste baseados no trajeto identificado da execução.

Casos de teste são preparados para forçar a execução de cada trajeto. Condições ou predicados de *node*s devem ser corretamente ajustados. Cada caso de teste é executado e comparado com o respectivo resultado. Exemplo:

TRAJETO 1 - Caso de Teste:

Para *node*-1, condição2 deve avaliar como VERDADEIRO (Identifica os valores necessários)
Para *node*-3, avaliação da var1 deve conduzir ao *node*-4 (Identifica valor de var1)
Resultados esperados: deve produzir necessariamente resultado para *node*-9.

3.1.2. Estrutura de Controle de Teste

Estrutura de Controle de Teste é uma técnica de teste caixa-branca que testa três tipos de controles da classe, a saber: testes de condição, testes de repetição e testes de fluxo de dados.

1. **Testes de Condição***.* Método de projeto do caso de teste que testa condições lógicas contidas em uma especificação processual. Focaliza em testar cada condição na classe fornecendo a combinação possível dos valores. Um exemplo, considerando a seguinte condição de um método:

   se ((resultado < 0) && (numeroDeTeste != 100))

Os casos de teste que podem ser gerados são mostrados na Tabela 1.

Caso de Teste	*resultado < 0*	*numeroDeTeste != 100*	*combinação*
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F

Tabela 1: Estrutura de Controle de Casos de Teste

2. **Teste de Repetição***.* É um método de projeto de caso de teste que foca exclusivamente na validade dos construtores iterativos ou da repetição. Existem quatro classes de iteração: simples, concatenada, aninhada e não-estruturada. São mostradas na Tabela 2.

Iteração Simples	**Iteração Concatenada**
FAÇA ENQUANTO Instrução FIM ENQUANTO	FAÇA ENQUANTO Instrução FIM ENQUANTO FAÇA ENQUANTO Instrução FIM ENQUANTO
Iteração Aninhada	**Iteração não estruturada**
FAÇA ENQUANTO Instrução FAÇA ENQUANTO Instrução FIM ENQUANTO FIM ENQUANTO	FAÇA ENQUANTO Instrução :rótulo1 FAÇA ENQUANTO Instrução :rótulo2 SE <condição> VÁ PARA rótulo1 FIM SE VÁ PARA rótulo2 FIM ENQUANTO FIM ENQUANTO

Tabela 2: Estrutura de Iteração ou Repetição

Para uma iteração simples, os casos de teste podem ser produzidos pelas seguintes possíveis execuções de iteração ou repetição.

- Salte o laço.
- Somente uma passagem através do laço.
- Duas passagens através do laço.
- m passagens através do laço, onde m < n
- n – 1, n, n + 1 passagens através do laço

Para iterações aninhadas, os casos de teste podem ser produzidos pelas seguintes possíveis execuções de iteração ou repetição.

- Comece com o laço mais interno.
- Conduza testes de laço simples para o laço mais interno ao prender o laço exterior em seus valores de parâmetros mínimos da iteração. Adicione o outro teste para valores fora de limite ou valores excluídos.
- Trabalhe de dentro para fora, conduzindo testes para laço seguinte mas mantendo todos os laços exteriores restantes em valores mínimos e em outros aninhados com valores "típicos".
- Continue até todos os laços serem testados.

Para a Iteração Concatenada, os casos de teste podem ser produzidos pelas seguintes possíveis execuções de iteração ou repetição.

- Se os laços são independentes, o teste para laços simples pode ser utilizado.
- Se os laços são dependentes, o teste para laços aninhados pode ser utilizado.

Para Iteração não estruturada, os casos de testes não devem ser produzidos desde que seja o melhor para replanejar o laço e pode ser que não seja uma boa forma de iteração ou construção de repetição.

3. **Teste de fluxo de dados***.* É um método de produção de caso de teste que seleciona os trajetos de acordo com a localização das definições e utilizações de variáveis no código.

3.2. Técnicas de Teste de caixa-preta

É um teste de projeto focado nos aspectos do projeto em relação aos seus requisitos funcionais. Engenheiros de software devem testar os requisitos funcionais do projeto por meio da entrada de dados. Isto define uma série de casos de teste que poderão indicar a ausência ou a incoerência de alguma função, erros de interfaces, erros na estrutura dos dados, erros no acesso externo às bases de dados, erros de performance, e erros de inicialização e término da aplicação.

3.2.1. Testes Baseados em Gráficos

Técnica de teste caixa-preta que usa os objeto que foram modelados durante o projeto e seus relacionamentos. Entender a dinâmica de como estes objetos se comunicam e colaboram uns com os outros é a forma que podem gerar casos de teste.

Desenvolvendo casos de teste usando Testes Baseados em Gráficos

PASSO 1: Criar um gráfico dos objetos do software e identificar os relacionamentos entre esses objetos.

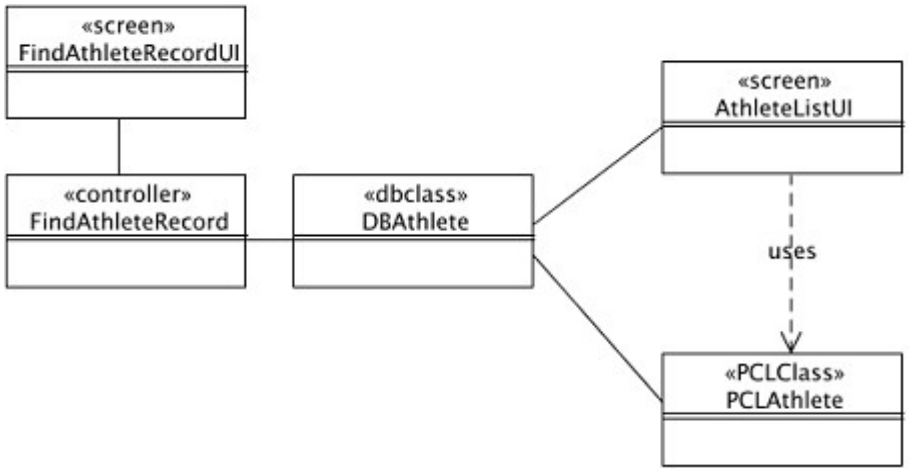

Figura 3: Diagrama de Colaboração para encontrar um atleta

Utilizando *node*s e bordas, crie o gráfico dos objetos do sistema. Os *node*s representam os objetos. Propriedades podem ser usadas para descrever os *node*s e bordas. Para engenharia de software orientada a objetos, o diagrama de colaboração é uma excelente forma de entrada de

dados para os testes baseados em gráficos, pois não se faz necessário a criação de gráficos. O diagrama de colaboração na Figura 3 é usado como um exemplo.

PASSO 2: Utilizar os gráficos para definir casos de teste.

Os casos de teste são:

Caso de Teste 1:

- A classe **FindAthleteUI** envia uma solicitação para pesquisar uma lista de atletas baseada em um critério de busca. A solicitação é enviada para **FindAthleteRecord**.
- **FindAthleteRecord** envia uma mensagem para **DBAthlete** para processar o critério de busca.
- **DBAthlete** solicita ao servidor de banco de dados que execute a instrução SELECT. Isto preenche a classe **PCLAthlete** com as informações do atleta. Retorna a referência de **PCLAthlete** para **AthleteListUI**.
- **AthleteListUI** lista os nomes dos atletas.

Dicas para os Testes Baseados em Gráficos:

1. Identificar o ponto de inicio e término do gráfico. Estes poderão ser os *nodes* de entrada e saída.
2. Nomear os *node*s e especifique as propriedades.
3. Estabelecer seus relacionamentos através das bordas. Especifique suas propriedades.
4. Criar casos de teste e garantir que cobra todos os *node*s e as bordas.

3.2.2. Teste de Equivalência

É um teste caixa-preta que utiliza as entradas de dados do código. Isto divide a entrada de dados em grupos para que os casos de teste possam ser desenvolvidos. Casos de teste são utilizados para descobrir erros que reflitam uma classe de erros. Isto, contudo, diminui os esforços no processo de teste. Para isto se faz uso das classes de equivalência, que são regras que validam ou invalidam a entrada de dados.

Dicas para Identificar as Classes de Equivalência

1. Condição de entrada é especificada como um comprimento do valor. Caso de Teste consiste em uma entrada válida e 2 classes de equivalência inválidas.
2. Condição de entrada requer um valor especifico. Caso de Teste consiste em uma classe válida e duas classes inválidas de equivalência.
3. Condição de entrada especificam o membro de um grupo. Caso de Teste consiste em uma classe válida e uma inválida.
4. Condição de entrada é um Boolean. Caso de Teste consiste em uma classe válida e uma inválida.

Como exemplo, considere uma mensagem de texto para o registro de um telefone móvel em um serviço para obter relatórios de tráfego. Assuma que a mensagem possui a seguinte estrutura:

Número do servidor de Serviço (Número do servidor provedor do serviço)	Código do Serviço (Código único que diz ao provedor de serviços qual serviço esta sendo solicitado)	Número do Telefone Móvel (O número de telefone para o qual as informações serão enviada)
765	234	09198764531

Condições de entrada associados com cada elemento de dado:

Número do servidor de Serviço	Condição de entrada 1: valor correto Condição de entrada 2: valor incorreto
Código do Serviço	Condição de entrada 1: valor correto Condição de entrada 2: valor incorreto
Número do Telefone Móvel	Condição de entrada 1: valor correto Condição de entrada 2: número faltando Condição de entrada 3: tamanho incorreto

3.2.3. Teste de Limite de Valor

É um teste caixa-preta que faz uso dos limites dos valores para se criar casos de teste. A maioria das falhas no projeto ocorrem no limite dos valores de entrada.

Dicas na criação de casos de teste utilizando Teste de Limite de Valor

1. Se a condição de entrada especificar um comprimento por n e m, o caso de teste pode ser feito da seguinte forma:
   - *usando valores n e m*
   - *somente valores acima de n e m*
   - somente abaixo de n e m
   -
2. Se a condição de entrada especificar o número de valores, o caso de teste pode ser:
   - usando o valor mínimo
   - usando o valor máximo
   - usando somente valores acima do valor mínimo
   - usando somente valores acima do valor máximo

4. Testando Classes

Testes de projeto podem ser realizados em várias etapas ou tarefas. Testes de projeto se preocupam com testes individuais (testes de unidade) ou com seus relacionamentos uns com os outros (teste de integração de sistema). Nesta sessão iremos discutir os conceitos e métodos que permitem realizar testes em classes.

4.1. Teste de Unidade

É o nível básico dos testes. Tem como intenção testar os menores blocos de código. Este é o processo no qual se executa cada módulo para confirmar se desenvolve o que sua função determina. Essa parte envolve testar as interfaces, estruturas de dados locais, condições limites, caminhos independentes e erro ao lidar com caminhos.

O ambiente no qual os testes de unidade podem ser realizados é mostrado na Figura 4. Para testar os módulos, são utilizados um *driver* e um *stub*. Um d*river* é uma classes que aceita testes de dados de entrada, envia os dados recebidos aos componentes a serem testados e mostra um resultado relevante. Um *stub* é uma classe que realiza o suporte a atividades como manipulação de dados, pesquisa do estado dos componentes a serem testados e impressão. Se o *driver* e *stub* necessitarem de um grande esforço para serem desenvolvidos, os testes de unidade podem ser atrasados até o teste de integração.

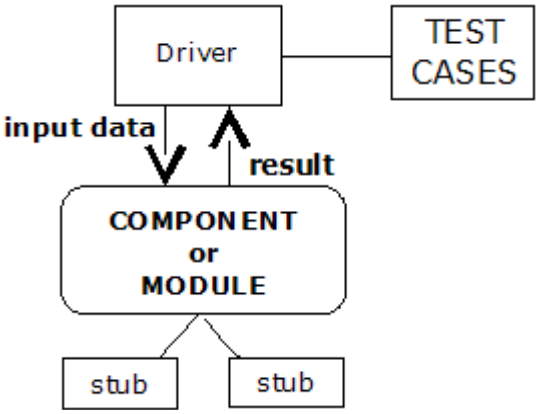

Figura 4 Ambiente de Teste de Unidade

Para se criar testes de unidade que sejam eficazes, faz-se necessário entender o comportamento da unidade no sistema que se está testando. Isto geralmente é feito decompondo os requisitos do sistema em pedaços simples que podem ser testados independentemente. É importante que os requisitos do projeto possam ser transformados em casos de teste.

Na engenharia de software orientada a objetos, o conceito de encapsulamento é utilizado para definir a classe; os dados (**atributos**) e funções (**métodos e operações**) são agrupados numa mesma classe. As menores unidades de teste são as operações definidas nessas classes. Contrária a forma de se realizar testes, operações definidas na classe não podem ser separadas para que o teste seja realizado. Isto deve ser testado no contexto dos objetos da classe instanciada. Como um exemplo, considere o diagrama de classe mostrado na Figura 5.

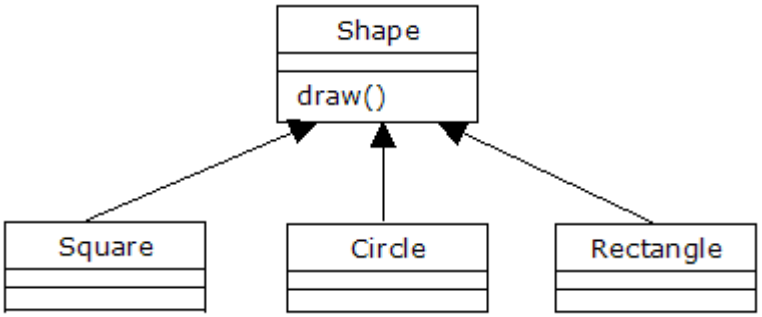

Figura 5: Modelo de Hierarquia de Classe

O método **draw**() é definido na classe **Shape**. Todas as subclasses da classe **Shape** conterão este método por meio de **herança**. Quando uma subclasses utiliza o método **draw**(), ele é executado juntamente com os atributos privados e outros métodos no contexto da subclasse. O contexto no qual o método é usado varia dependendo de que subclasse o executa. Consequentemente, é preciso testar o método **draw**() em todas as subclasses que o utilizam. Como o número de subclasses é definido para a classe base, mais testes serão necessários.

4.2. Teste de Integração

Após os testes de unidade, todas as classes devem ser testadas em integração. **O Teste de Integração do Sistema** verifica se cada componente interage de maneira satisfatória quando agregados e se todas as interfaces estão corretas. Softwares orientados a objetos não possuem uma hierarquia óbvia de controle de sua estrutura, que é uma característica no modo convencional em se desenvolver sistemas. As formas tradicionais de teste de integração (de cima-para-baixo ou de baixo-para-cima) possuem uma relevância muito reduzida nesses projetos. Entretanto, existem duas abordagens que podem ser empregadas na performance dos testes de integração.

4.3. Teste de Aproximação baseado em Thread

Teste de Integração é baseado num grupo de classes que colaboram ou interagem quando uma entrada necessita ser processada ou um evento foi ativado. Uma *thread* é um caminho para a comunicação entre essas classes que precisam processar uma única entrada ou responder a um evento. Todas as possíveis *threads* precisam ser testadas. Diagramas de Seqüência e de Colaboração podem ser utilizados como base para estes testes. Como um exemplo, considere o Diagrama de Seqüência que recupera o registro de um atleta, como mostrado na Figura 6. No Teste de Integração, somente, siga a ordem em que há interação entre os objetos por meio de suas mensagens.

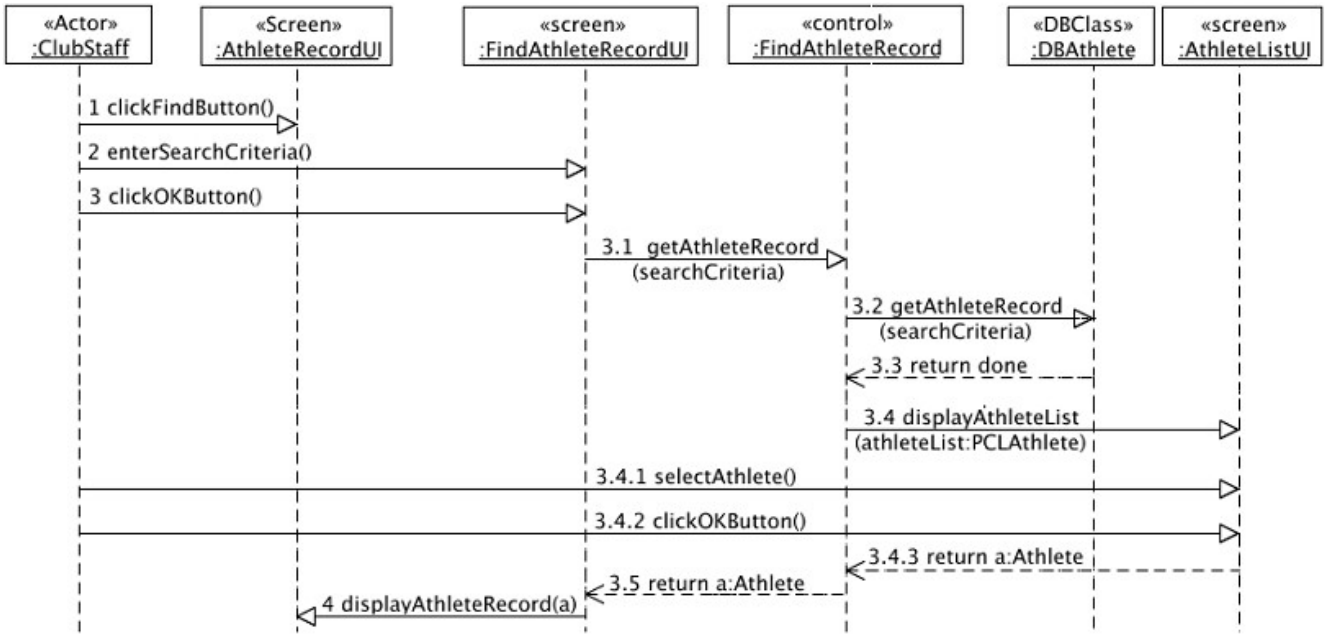

Figura 6: Recuperando um registro de Atleta

A seqüência de testes poderá envolver os seguintes passos:

1. A comissão irá clicar no **Find Button** do **Athlete Record User Interface**.
2. A tela de **FindAthleteRecordUI** será exibida.
3. A comissão insere os critérios de pesquisa.
4. A comissão pressiona o botão **OK** que instanciará o controlador **FindAthleteRecord.**
5. O controlador **FindAthleteRecord** se comunicará com o **DBAAthelte** para popular a **PCLAthleteList.**

6. A tela **AthleteListUI** será exibida contendo uma lista com os atletas que satisfazem os critérios utilizados na pesquisa.
7. A comissão selecionará o atleta que se deseja recuperar.
8. A comissão pressiona o botão **OK** ou pressiona enter.
9. A tela **AthleteRecordUI** deverá ser mostrada contendo as informações do atleta nos campos apropriados.

4.4. Teste de Aproximação baseado em Uso

Os testes de integração começam por identificar as classes independentes. **Classes Independentes** são aquelas classes que não usam, ou usam pouco, outras classes. As classes independentes são testadas primeiro. Após o teste dessas classes, o próximo conjunto de classes a ser testado e chamado de **Classes Dependentes**. As Classes Dependentes são aquelas que fazem uso das Classes Independentes. Como um exemplo, considere as classes mostradas na Figura 7. As classes **Athlete** e **PCLAthlete** podem ser consideradas como classes independentes. Seus métodos podem ser testados independentemente. Se os testes forem satisfatórios, a classe **DBAthlete** poderá ser testada.

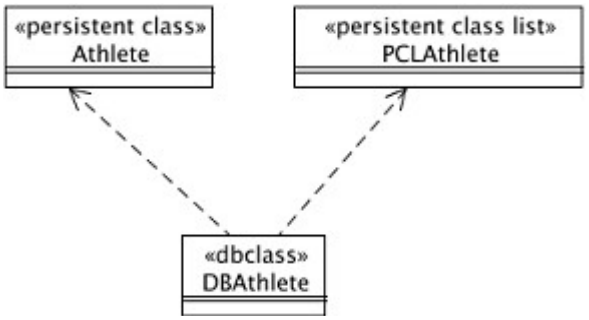

Figura 7: Cluster DBAthlete

Para auxiliar na integração, ***Clustering*** pode ser usado. *Clustering* é o processo que define um grupo de classes que podem ser integradas e testadas juntas. Estas são chamadas de clusters porque são consideradas como sendo uma unidade. No exemplo, as três classes combinadas podem ser identificadas como sendo um **Cluster DBAthlete**.

4.5. Teste de Regressão

Algumas vezes quando algum erro é corrigido, é necessário refazer o teste dos componentes. O **Teste de Regressão** é uma reexecução de alguma parte do componente para assegurar que mudanças realizadas na correção dos erros não produziram erros não-intencionais. Um novo teste ocorre para as funções do sistema que foram afetadas, componentes que foram modificados e os que podem ser afetados por uma mudança.

5. Metodologia de Desenvolvimento voltada para o Teste

Desenvolvimento voltado para o Teste (TDD) é um método de desenvolvimento que adota a abordagem de primeiro testar e refazer. Esta abordagem de se testar primeiro é uma técnica de programação que envolve análise, projeto, codificação e teste em conjunto. Dois passos básicos estão envolvidos nessa abordagem:

- Escrever somente o necessário para se descrever o próximo incremento de comportamento.
- Escrever somente o código necessário para se passar no teste.

Assim como o implica o nome, o teste é desenvolvido antes.

Refatoração, por outro lado, significa melhorar um código já existente. É um modo sistemático de se melhorar um código eliminando as partes redundantes ou duplicadas. Este é um processo no qual se altera a estrutura interna do código sem que sejam feitas alterações no comportamento do projeto. Muitos programadores utilizam refatoração em seus códigos regularmente tendo por base um senso comum.

No desenvolvimento voltado para testes, cada novo teste gera novas informações que ajudam a tomar as decisões de projeto. Muitas vezes, aquele que não realiza o projeto tem problemas ate que ele seja utilizado. Testes nos permitem descobrir problemas no sistema.

5.1. Benefícios do Desenvolvimento Voltado para Testes

Existem vários benefícios. Alguns deles são enumerados abaixo.

1. **Permitir o desenvolvimento simples de incremento**. Um dos benefícios imediatos do uso do TDD é que essa prática gera um sistema quase que imediatamente. A primeira iteração do projeto é bastante simples e pode não conter muitas funcionalidades, entretanto as funcionalidades serão incrementadas com o contínuo desenvolvimento. Isso é menos arriscado do que construir o sistema inteiro e esperar que suas partes funcionem juntas.
2. **Envolver um processo de desenvolvimento mais simples**. Desenvolvedores ficam mais concentrados porque o único detalhe com o qual precisam se preocupar deve ser passar no próximo teste. Concentrar-se numa pequena parte do projeto, colocar para funcionar, e partir para próxima etapa.
3. **Prover um constante teste de regressão**. A modificação em um módulo pode ter conseqüências não esperadas no resto do projeto. TDD executa um grupo de testes de unidade toda vez que uma mudança é realizada. Isto significa que toda modificação no código que pode gerar um erro não esperado em outra parte do projeto será detectado imediatamente e corrigido. Outro benefício dos constantes testes de regressão é que sempre teremos um sistema funcionando a cada iteração do desenvolvimento. Isto lhe permite parar com o desenvolvimento do sistema a qualquer hora e rapidamente fazer modificações nos requisitos.
4. **Melhorar a comunicação**. Os testes de unidade servem como uma linguagem comum que pode ser usada na comunicação do comportamento exato de um componente sem ambigüidades.
5. **Melhorar no entendimento do comportamento do sistema**. Escrever os testes de unidade antes de escrever o código ajuda ao desenvolvedor a se concentrar em entender os requisitos. Ao escrever testes de unidade, o desenvolvedor adiciona critérios de sucesso ou falha no comportamento do projeto. Cada um desses critérios adicionados servem para perceber como o sistema deve se comportar. Com a adição de mais testes de unidade para as novas características ou correções, um conjunto de testes representa o comportamento do sistema.
6. **Centralizar o conhecimento**. Os testes de unidade servem como um repositório que provê informação sobre as decisões de projeto que vierem a serem tomadas em algum módulo. Os testes de unidade provêem um lista dos requisitos enquanto o código-fonte realiza a

implementação dos requisitos. Usar essas duas fontes como informação torna muito mais fácil para o desenvolvedor entender o módulo e realizar as mudanças que não irão introduzir falhas no projeto.

7. **Melhorar o projeto**. Fazer com o que o desenvolvedor escreva os testes de unidade antes da codificação ajuda o mesmo a entender melhor o comportamento do projeto como um todo e ajudar a pensar no sistema do ponto de vista do usuário.

5.2. Passos de Desenvolvimento voltados para Teste

Desenvolvimento voltado para testes é basicamente composto dos seguintes passos:

1. Escrever um teste que defina como o sistema deverá se comportar. Como um exemplo, suponha que "Ang Bulilit Liga" tenha as seguintes exigências, "O software deve calcular a média de lançamento de cada atleta por jogo e por temporada." Existe um número de comportamentos que o software precisa ter para que possa implementar as exigências dadas:
   - O software deverá identificar um atleta.
   - Deverá haver um modo para que seja realizada a entrada dos dados para que a média possa ser calculada.
   - Deverá haver uma maneira para identificar o jogo.
   - Deverá haver uma forma de receber a média de lançamento em um determinado jogo.
   - Deverá haver uma forma de receber a média de toda a temporada.

Após serem feitas algumas suposições de que os atletas poderão ser identificados pelos seus ids e data dos jogos, podemos escrever o caso de teste. Como um exemplo, uma parte de um teste de unidade é mostrada no código abaixo.

```
Athlete timmy = PCLAthleteList.getAthlete(3355);
AthleteStatistics timmyStatistics = new AthleteStatistics(timmy);

// Adicionando a média de lançamento para as tentativas de 1 a 9
// último jogo 8/9/2004.
timmyStatistics.addShootingAverage("8/9/2004",1,9);

// Adicionando a média de lançamento para as tentativas de 4 a 10
// último jogo 8/16/2004.
timmyStatistics.addShootingAverage("8/16/2004",4,10);

// Adicionando a média de lançamento para as tentativas de 7 a 10
// último jogo 8/25/2004.
timmyStatistics.addShootingAverage("8/25/2004",7,10);

// Calculando a média para o dia 16/08
float gameShootingAverage =
 timmyStatistics.getGameShootingAverage("8/16/2005");

// Calculando a média para a temporada
float seasonShootingAverage = timmyStatistics.getSeasonShootingAverage();

//checando a média para o jogo
assertEquals(200, gameShootingAverage);

//checando a média para a temporada
assertEquals(214, seasonShootingAverage);
```

2. Fazer com que o teste rode de forma mais fácil e rápida possível. Não é necessário preocupar-se com o código-fonte; deve fazer com que funcione. No exemplo de caso utilizado, o código-fonte que estamos fazendo referência é a classe **AthleteStatistics** que contém o método que calcula a média de um atleta em um jogo ou em uma temporada. Uma vez o teste escrito, ele é executado para verificar a possível existência de falhas. A classe falha pois o código para a implementar o comportamento ainda não foi escrito. Uma vez que

o teste de unidade foi verificado, o código para implementar o comportamento é desenvolvido e o teste de unidade é novamente executado para assegurar que o software funciona como o esperado. Uma classe inicial **AthleteStatistics** é mostrada abaixo.

```
public class AthleteStatistics{
  private Athlete a;
  private Statistics stats;

  // Construtor com um atleta como parâmetro
  public AthleteStatistics(Athlete theAthlete){
    // Lugar para inicialização dos atributos do atleta
    a = theAthlete;
    stats = new Statistics();
  }
  public void addShootingAverage(String dateOfGame, int shots,
    int attempts) {
    // Lugar onde o código para adição das estatísticas de um jogo.
  }
  public float getGameShootingAverage(String dateOfGame) {
    // Cálculos.
  }
  public float getSeasonShootingAverage(){
    // Cálculos.
  }
}
```

3. Arrumar o código-fonte. Agora que o código se encontra-se funcionando corretamente, refazer o código e remover qualquer duplicidade ou qualquer outro problema que foi introduzido para que o teste pudesse ser feito. Os desenvolvedores devem saber que escrever o código em primeiro lugar não é uma boa prática de implementação, não devem ter medo de refazer. Ao corromper o código, o teste de unidade emitirá imediatamente um aviso.

5.3. Testando Classes Java com JUnit

JUnit é utilizado para realizar os chamados testes unitários. Isso significa que é utilizado para escrever e organizar casos de teste para cada classe separadamente. Os casos de teste poderiam ser escritos sem a ajuda do *Junit*. Porém os testes do *JUnit*, além de serem mais fáceis de codificar, permitem ao programador uma maior flexibilidade para que possa agrupar pequenos testes, permitindo que execuções simultâneas sejam realizadas para facilitar o processo de depuração. Esta sessão irá discutir como o *JUnit* é utilizado com o Java Studio™ Enterprise 8[1].

5.3.1. Testes de Unidade com *JUnit*

Os testes de unidades com o *JUnit* podem ser realizados de acordo com os seguintes passos:

1. Construir uma subclasse da classe **TestCase.**.
2. Criar um método chamado **setUp()** para iniciar dados (atributos ou construtor) antes do teste.
3. Escrever quantos métodos para o teste forem necessários.
4. Criar um método chamado **tearDown()** se necessitar que alguma informação seja apagada após o teste.

Para executar vários testes ao mesmo tempo, é preciso criar uma suite de testes. O *JUnit* possui um objeto, **TestSuite**, que pode rodar inúmeros testes ao mesmo tempo assim como testar várias outras suites.

5.3.2. Utilizando o *JUnit* no Java™ Studio Enterprise 8 da Sun

Criar testes no Java™ Studio Enterprise 8 da Sun é muito simples. Utilize o modelo mostrado na

1 Foi mantido o Software Java Studio™ Enterprise 8 conforme no original em inglês, entretanto, esta seção para as vídeo-aulas utilizará o software NetBeans (N.T.).

Figura 8 como a classe a ser testada. Crie um novo teste selecionando **Tools → JUnit Tests → Create Tests** na barra de menus. Tenha certeza de que a classe a testar está devidamente selecionada. De outro modo, os testes para uma classe podem ser criados com um clique direito do mouse sobre o nome da classe e selecionando **Tools → JUnit Tests → Create Tests**.

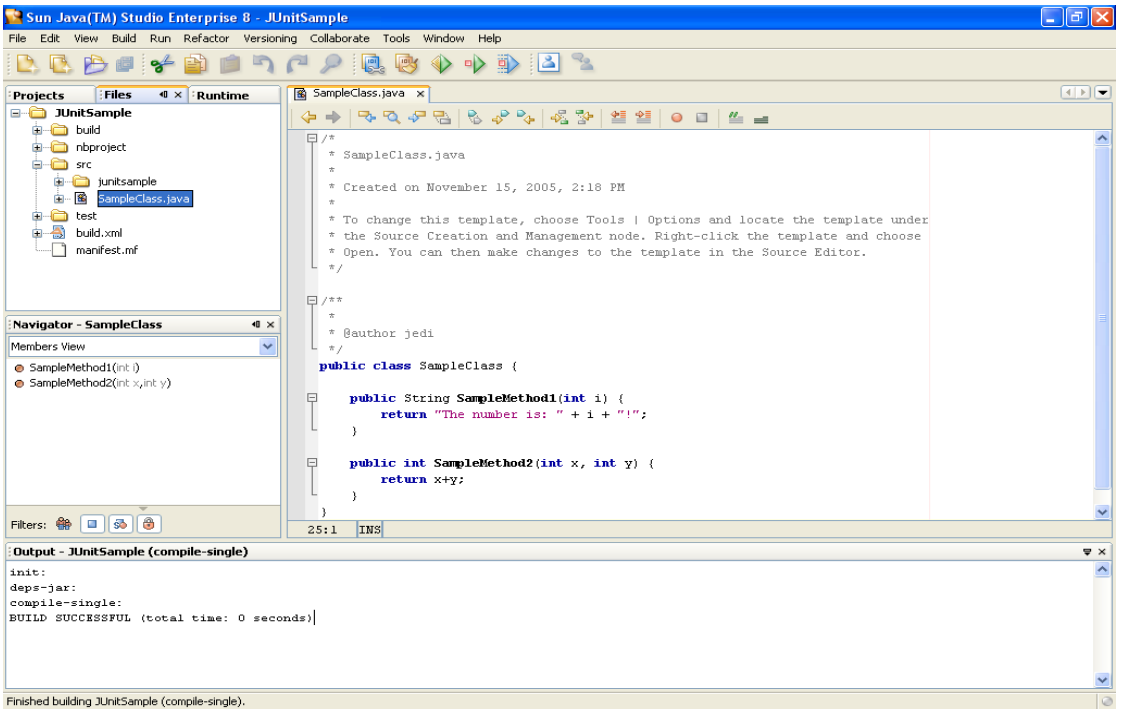

Figura 8: Modelo de código a ser testado

Na janela **Create Tests**, selecionar as opções de **Code Generation** para a realização do teste. A IDE gera um modelo com as opções que foram especificadas. Neste caso, nada será marcado com a exceção da opção **Public**. O nome da classe para o teste pode, inclusive, ser modificado (Figura 9). As configurações do *JUnit* podem ser modificadas no ***JUnit Module Settings*** (Módulo de Configuração do *JUnit*) ou ainda na própria janela para a criação de testes. Pressionar o botão **OK** irá gerar uma estrutura básica de testes (Figura 10).

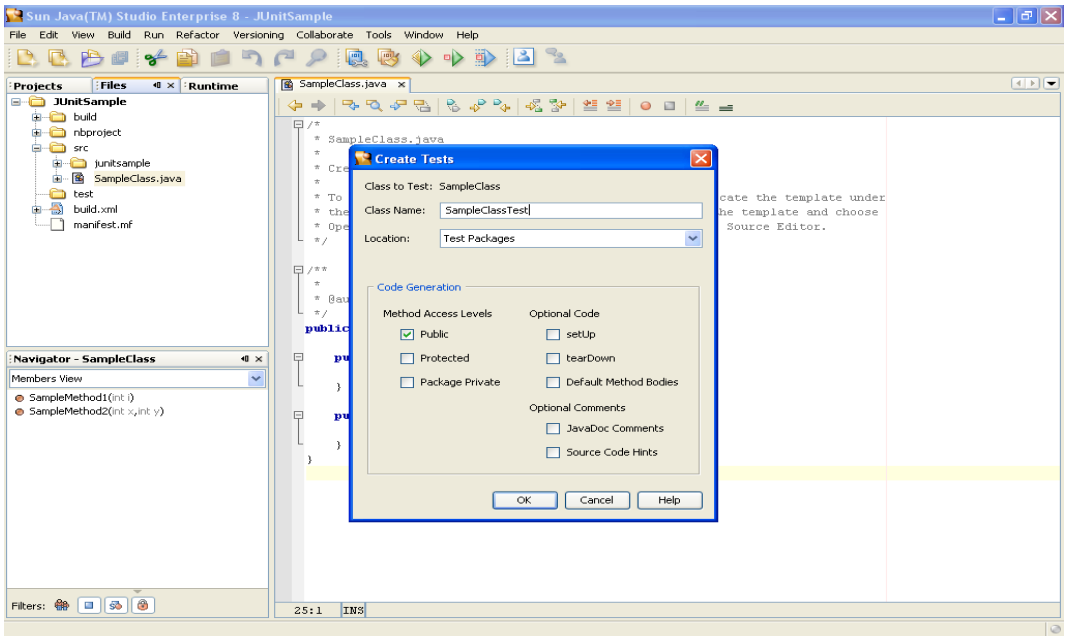

Figura 9: Janela do Junit para a criação de testes

Os métodos podem ser escritos utilizando-se expressões de testes da classe *Assert* (Classjunit. framework.Assert). Figura 11 nos mostra alguns métodos escritos para testar o código da Figura

1. Para executar o teste, selecione **Run → Test "project-name"** na barra de menus. A saida do teste será gerado pela IDE.

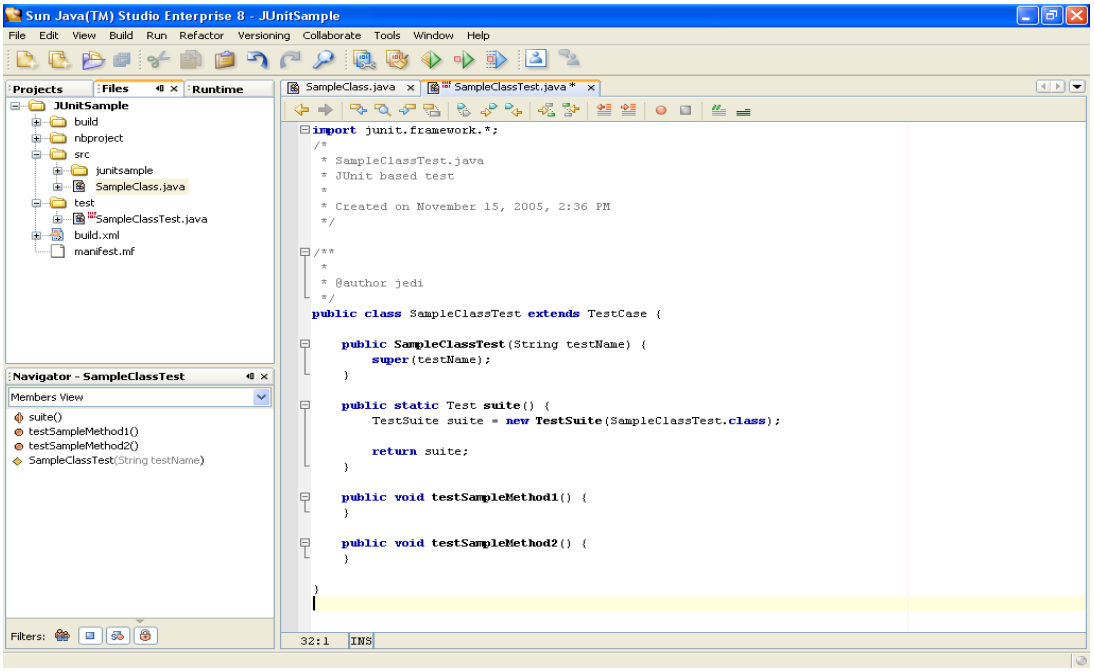

Figura 10: Teste gerado pela IDE

Se ocorrer alguma falha nos testes o erro será mostrado e, em seguida, o que causou o erro. Neste caso, o **testSampleMethod1** falhou, pois o que era esperado foi diferente do que foi obtido. Já o **testSampleMethod2** não falhou, por isso, nenhuma mensagem de erro foi mostrada.

A suite de testes pode ser editada de forma a adicionar novos testes fora do arquivo. Outras suites de testes podem ser incluídas no nosso teste. Outros métodos como **setUp()** e **tearDown()** podem ser incluídos nos testes. Estes métodos são chamados de testes de fixação pois os mesmo são executados antes ou depois dos testes.

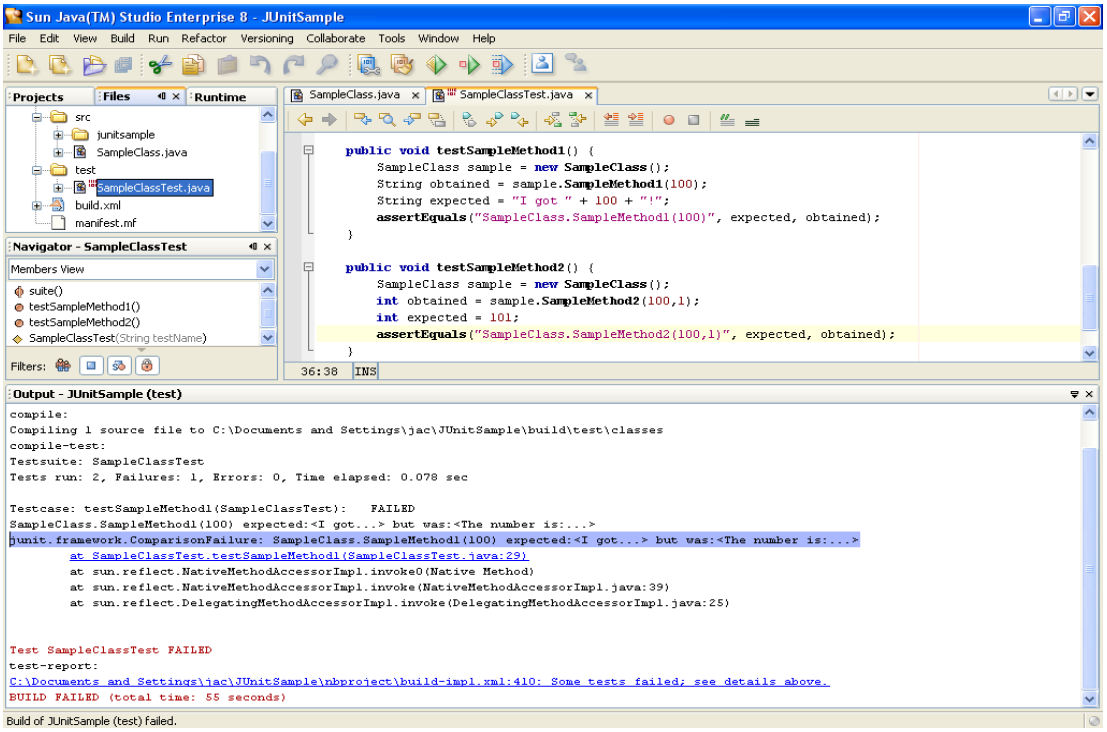

Figura 11 Caso de teste no JUnit utilizando assertEquals()

Partes do projeto são mostradas nos códigos seguintes.

```
/*
 * SampleClass.java
 */
public class SampleClass {
  public String sampleMethod1(int i) {
    return "The number is: " + i + "!";
  }
  public int sampleMethod2(int x, int y) {
    return x+y;
  }
}

/*
 * SampleClassTest.java
 */
import junit.framework.*;

public class SampleClassTest extends TestCase {
  public SampleClassTest(String testName) {
    super(testName);
  }
  public static Test suite() {
    TestSuite suite = new TestSuite(SampleClassTest.class);
    return suite;
  }
  public void testSampleMethod1() {
    SampleClass sample = new SampleClass();
    String obtained = sample.SampleMethod1(100);
    String expected = "I got " + 100 + "!";
    assertEquals("SampleClass.SampleMethod1(100)", expected, obtained);
  }
  public void testSampleMethod2() {
    SampleClass sample = new SampleClass();
    int obtained = sample.SampleMethod2(100,1);
    int expected = 101;
    assertEquals("SampleClass.SampleMethod2(100,1)", expected, obtained);
  }
}
```

6. Testando o Sistema

O teste de sistema é preocupa-se em testar um sistema inteiro baseado em sua especificação. O software a ser testado é comparado com a sua especificação funcional. Depois do teste de integração, o sistema é testado como um todo com foco na funcionalidade e exatidão. Isto deve incluir testes como o de performance, que testa o tempo de resposta e utilização de recursos; teste de estresse, que testa o software sujeito a quantidade, freqüência e volume de dados, pedidos e respostas anormais; teste de segurança, que testa os mecanismos de proteção do software; teste de recuperação, que testa o mecanismo de recuperação do software quando ele falha; etc.

No contexto de engenharia de software orientada a objeto, os requisitos de teste são derivados dos artefatos de análise ou produtos que usem UML, como diagramas de classe, de seqüência e de colaboração. Para derivar os casos de teste de sistema, o uso do diagrama de caso de uso é uma boa fonte. Eles representam as funcionalidades de alto nível providas ao usuário pelo sistema. Conceitualmente, a funcionalidade pode ser vista como um conjunto de processos, que são executados horizontalmente pelo sistema, e de objetos, como componentes de subsistema que são executados verticalmente. Cada funcionalidade não usa cada objeto; entretanto, cada objeto pode ser usado por muitas requisições funcionais. Esta transição de visão funcional para orientada à objeto é conseguida com casos e cenários.

Entretanto, estes casos não são independentes. Eles não têm apenas as dependências de extensão e inclusão. Têm também dependências seqüenciais que vêm da lógica do negócio em que o sistema se apóia. Quando planejamos casos de teste, precisamos identificar nas possíveis seqüências de execução como elas podem disparar diferentes falhas.

Um cenário é uma instância de um caso de uso ou um caminho completo de um caso de uso. Os usuários finais do sistema completo podem percorrer muitos caminhos enquanto executam a funcionalidade especificada no caso de uso. Cada caso de uso deve ter um fluxo básico e um fluxo alternativo. O fluxo básico cobre o fluxo normal do caso de uso. O fluxo alternativo cobre o comportamento de uma exceção ou de um caminho opcional, levando em consideração o comportamento normal do sistema.

A seção seguinte fornece os passos para derivar os casos de teste dos casos de uso. Ela consiste em uma adaptação de *Heumann*[2].

6.1. Gerando Casos de Teste de Sistema

PASSO 1: Utilizar RTM e priorizar os casos de uso.

Muitos projetos de desenvolvimento de software podem ser obrigados a entregar o software tão rápido quanto possível e com um pequeno time de desenvolvedores. Neste caso, priorizar quais funcionalidades devem ser desenvolvidas e testadas primeiro é muito crucial. A importância é estimada com base na freqüência com que cada função do sistema é usada. O RTM é uma boa ferramenta para determinar quais funcionalidades devem ser desenvolvidas e testadas primeiro.

RTM ID	*Requisito*	*Anotações*	*Solicitante*	*Data*	*Prioridade*
Caso de Uso 1.0	O sistema deve ser capaz de dar manutenção nas informações pessoais de um atleta.		RJCS	12/06/05	Deve ter
Caso de Uso 2.0	O sistema deve ser capaz de permitir ao treinador mudar o status de um atleta.		RJCS	12/06/05	Deve ter
Caso de Uso 3.0	O sistema deve ser capaz de exibir as informações pessoais de um atleta.		RJCS	12/06/05	Deve ter
Caso de Uso 1.1	O sistema deve ser capaz de adicionar o registro de um atleta.		RJCS	13/06/05	Deve ter

2 Heuman, J. Generating Test Cases for Use Cases. The Rational Edge, Rational Rose Software, 2001.

RTM ID	Requisito	Anotações	Solicitante	Data	Prioridade
Caso de Uso 1.2	O sistema deve ser capaz de editar o registro de um atleta.		RJCS	13/06/05	Deve ter
Caso de Uso 1.3	O sistema deve ser capaz de excluir o registro de um atleta.		RJCS	13/06/05	Deve ter

Tabela 3: Amostra Inicial de RTM para a Manutenção dos Membros de um Clube

Considere o RTM formulado durante a análise mostrado na Tabela 3. Os usuários devem decidir fazer o Caso de Uso 1.0 e seus sub-casos de uso antes do resto.

PASSO 2: Para cada caso de uso, crie os cenários.

Para cada caso de uso, começando da prioridade maior para a menor, crie um conjunto suficiente de cenários. Ter uma descrição detalhada de cada cenário como suas entradas, pré-condições e pós-condições pode ser muito útil nos estágios futuros da geração de testes. A lista de cenários deve incluir o fluxo básico e, pelo menos, um fluxo alternativo. Quanto mais fluxos alternativos, mais compreensiva será a modelagem e, conseqüentemente, testes mais completos. Como exemplo, considere a lista de cenários para incluir um atleta ao sistema na Tabela 4.

Cenário ID	Nome do Cenário
SCN-01	Dados de um atleta adicionados com sucesso
SCN-02	Dados de um atleta estão incompletos
SCN-03	Dados de um atleta são inválidos
SCN-04	Entrada duplicada
SCN-05	Sistema indisponível
SCN-06	Sistema falha durante a entrada

Tabela 4: Lista de cenários para incluir o registro de um atleta

PASSO 3: Para cada cenário, obter pelo menos um caso de teste e identificar as condições de execução.

As pré-condições e o fluxo de eventos de cada cenário podem ser examinadas para identificar as variáveis de entrada e as restrições que trazem o sistema a um estado específico representado pelas pós-condições. Uma matriz é apropriada para uma documentação clara dos casos de teste para cada cenário. Como exemplo, considere a matriz mostrada na Tabela 5.

Caso de Teste ID	Cenário	Dados do Atleta	Não está na Base de Dados	Resultados Esperados
TC-01	Dados de um atleta adicionados com sucesso	V	V	Mostrar mensagem: Registro de atleta incluído.
TC-02	Dados de um atleta estão incompletos	I	V	Mensagem de erro: Dados incompletos.
TC-03	Dados de um atleta são inválidos	I	V	Mensagem de erro: dados inválidos.
TC-04	Entrada duplicada	V	I	Mensagem de erro: Registro deste atleta já existe.
TC-05	Sistema indisponível	V	I	Mensagem de erro: Sistema indisponível.

Caso de Teste ID	Cenário	Dados do Atleta	Não está na Base de Dados	Resultados Esperados
TC-06	Sistema falha durante a entrada	V	I	Sistema falha, mas deve ser capaz de recuperar os dados.

Tabela 5: Amostra de matriz de caso de teste para adicionar o registro de um atleta

A matriz de teste representa um *framework* de teste sem envolver valores específicos de dados. O V indica válido, o I indica inválido e N/A indica não aplicável. Esta matriz é um passo intermediário e provê um bom método para documentar as condições que estão sendo testadas.

PASSO 4: Para cada caso de teste, determinar os valores para os dados.

Uma vez que todos os testes foram identificados, eles devem ser completados, revisados e validados para garantir a acurácia e identificar os casos de teste redundantes ou ausentes. Se isto é alcançado, o próximo passo é determinar os valores para cada V e I. Antes de distribuir o aplicativo, ele deve ser testado usando dados reais para ver se outros problemas, como performance, são identificados. A Tabela 6 mostra algumas amostras de valores.

Caso de Teste ID	Cenário	Dados do Atleta	Não está na Base de Dados	Resultado Esperado
TC-01	Dados de um atleta adicionados com sucesso	Johnny De la Cruz Lot 24 block 18 St. Andrewsfield, Quezon City 24/9/1998 Masculino Status Responsável: Johnny De la Cruz, Sr. -mesmo endereço- 555-9895	V	Mostrar mensagem: Registro de atleta incluído.
TC-02	Dados de um atleta estão incompletos	Sem responsável	V	Mensagem de erro: Dados incompletos.
TC-03	Dados de um atleta são inválidos	Falta data de nascimento	V	Mensagem de erro: dados inválidos.
TC-04	Entrada duplicada	Johnny De la Cruz Lot 24 block 18 St. Andrewsfield, Quezon City 24/9/1998 Masculino Status Responsável: Johnny De la Cruz, Sr. -mesmo endereço- 555-9895	I	Mensagem de erro: Registro deste atleta já existe.

Tabela 6: Amostra de matriz de caso de teste para adicionar o registro de um atleta com valores

6.2. Testes de validação

Começam depois do ápice do teste de sistema. Consiste numa série de casos de teste caixa-preta que demonstram conformidade com os requisitos. O software é completamente integrado como um sistema e erros de interface entre componentes de software foram descobertos e corrigidos. O foco é nas ações visíveis ao usuário e nas saídas reconhecíveis pelo usuário.

Um ***Critério de Validação*** é um documento contendo todos os atributos do software visíveis ao usuário. Ele é a base para um teste de validação. Se um software exibe estes atributos, então o software atende aos requisitos.

6.3. Testes Alfa & Beta

Uma série de testes de aceitação que permitem ao usuário final validar todos os requisitos. Podem ir desde testes informais a testes executados de forma planejada e sistemática. Este teste permite aos usuários finais descobrir erros e defeitos que só eles podem encontrar. Um **Teste Alfa** é conduzido em um ambiente controlado, geralmente no ambiente de desenvolvimento. Usuários finais são solicitados a usar o sistema como se estivessem trabalhando naturalmente, enquanto os desenvolvedores estão anotando erros e problemas de utilização. ***Testes Beta***, ao contrário, são conduzidos em um ou mais ambientes do usuário e os desenvolvedores não estão presentes. Neste caso, são os próprios usuários finais que anotam todos os erros e problemas de utilização. Eles encaminham estes erros para os desenvolvedores, para que façam os consertos.

7. Mapeando os Produtos do Teste de Software RTM

Quando um componente é implementado, a especificação de teste e os casos de teste devem ser desenvolvidos. Precisamos estar atentos a ele e, ao mesmo tempo, termos certeza de que cada teste é mapeado para um requisito. Componentes RTM adicionais são adicionados. Abaixo temos a lista dos elementos recomendados para teste de software RTM.

Componentes de Teste de Software RTM	***Descrição***
Especificação de Testes	O nome do arquivo que contém o plano de como testar os componentes do software.
Casos de Teste	O nome do arquivo que contém os casos de teste para serem executados como parte da especificação de teste.

Tabela 7: Elementos de Teste de Software RTM

Estes componentes devem ser relacionados a um componente de software definido no RTM.

8. Métricas de Teste

As métricas usadas para medir a qualidade de um projeto orientado a objeto mencionadas na Engenharia de Projeto também podem prover uma indicação da quantidade de esforço de teste necessária para testar software orientado a objeto. Além disso, outras métricas podem ser consideradas para encapsulamento e herança. Exemplos de tais métricas seguem abaixo:

1. **Falta de Coesão nos Métodos** - *Lack of Cohesion in Methods (LCOM)*. Se o valor do LCOM é alto, mais estados da classe precisam ser testados.
2. **Porcentagem Public e Protected** - *Percent Public and Protected (PAP)*. Esta é a medida da porcentagem dos atributos de classe que são public ou protected. Se o valor do PAP é alto, aumenta a probabilidade de efeitos colaterais em outras classes por lidar com um alto acoplamento.
3. **Acesso Público a Data Members** - *Public Access To Data Members (PAD)*. Esta é uma medida do número de classes ou métodos que acessam atributos de outras classes. Se o valor do PAD é alto, poderão ocorrer muitos efeitos colaterais.
4. **Número de Classes Raiz** - *Number of Root Classes (NOR)*. Se o número de classes root aumenta, o esforço de teste também aumenta.
5. **Número de Filhos** - *Number of Children (NOC)* – e **Profundidade de Árvore de Herança** - *Depth of the Inheritance Tree (DIT)*. Métodos de superclasses devem ser novamente testados para cada subclasse.

9. Exercícios

9.1. Especificando um Caso de Teste

1. Usando o Teste de Fluxo Básico, escreva os casos de testes que devem ser feitos para o método Obter o Registro de um Atleta. Determine a complexidade ciclomática do código para determinar o número de caminhos independentes. Cada um destes caminhos tornar-se-á um caso de teste.

9.2. Especificando Casos de Teste de Sistema

1. Escreva os casos de teste de sistema para o Sistema de Manutenção das Informações de Treinadores.
2. Escreva os casos de teste de sistema para o Sistema de Manutenção das Informações de Equipes.

9.3. Atividades de Projeto

O objetivo das atividades de projeto é reforçar o conhecimento e as habilidades adquiridos neste capítulo. Particularmente, eles são:

1. Definir os casos de teste.
2. Definir os casos de teste de sistema.
3. Rastrear a revisão e teste usando a Matriz de Rastreabilidade de Requisitos.

PRODUTOS PRINCIPAIS:

1. Casos de Teste
2. Casos de Teste de Sistema
3. Resultados dos Testes

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.