ULBRA
CENTRO UNIVERSITÁRIO LUTERANO DE PALMAS
COMUNIDADE EVANGÉLICA LUTERANA "SÃO PAULO"
Credenciado pelo Decreto de 06/07/2000 - D.O.U. nº 130 de 07/07/2000

Rodrigo Cardoso Vaz

JUnit – Framework para Testes em Java

Palmas

2003

ULBRA
CENTRO UNIVERSITÁRIO LUTERANO DE PALMAS
COMUNIDADE EVANGÉLICA LUTERANA "SÃO PAULO"
Credenciado pelo Decreto de 06/07/2000 - D.O.U. nº 130 de 07/07/2000

Rodrigo Cardoso Vaz

JUnit – Framework para Testes em Java

"Relatório apresentado como requisito parcial da disciplina Prática de Sistemas de Informação I do curso de Sistemas de Informação, orientado pelo Professor M. Sc. Eduardo Kessler Piveta".

Palmas

2003

Rodrigo Cardoso Vaz

JUnit – Framework para Testes em Java

"Relatório apresentado como requisito parcial da disciplina Prática de Sistemas de Informação I do curso de Sistemas de Informação, orientado pelo Professor M. Sc. Eduardo Kessler Piveta".

Aprovada em dezembro de 2003.

BANCA EXAMINADORA

Profª Eduardo Leal, M. Sc. - Examinador

Centro Universitário Luterano de Palmas

Prof. Thereza Patrícia Pereira Padilha, M. Sc. - Examinadora

Centro Universitário Luterano de Palmas

Profª Eduardo Kessler Piveta, M. Sc. - Orientador

Centro Universitário Luterano de Palmas

Palmas

2003

AGRADECIMENTOS

Agradeço a Deus em primeiro lugar, que me deu a oportunidade de concluir mais esta etapa de minha vida. Superando até os momentos mais difíceis. À minha família que sempre me deu força e incentivo para continuar até o fim. Aos professores, em especial meu orientador, Eduardo Kessler Piveta, que se dispuseram de seu tempo para a realização, conclusão e sucesso desse trabalho. Não posso deixar de agradecer aos meus amigos que contribuíram com sua experiência, conhecimento e com sua grande demonstração de amizade por estar ao lado não só no processo de desenvolvimento deste trabalho, mas em todos os momentos durante o período da faculdade.

SUMÁRIO

LISTA DE FIGURAS

RESUMO

Este trabalho apresenta a execução de testes em classes implementadas na Linguagem de Programação Java, através da utilização do ***Framework Junit***, que possibilita a realização automatizada dos testes, de forma a poder avaliar os resultados esperados com os obtidos.

1 INTRODUÇÃO

Os processos de engenharia de *Software* descrevem quais as atividades que devem ser realizadas de forma a desenvolver software de maneira previsível e repetível. Exemplos de processos de desenvolvimento de software: *RUP*, *XP*, *Unified Process*, *Catalysis*, etc.

Dentre as atividades de um processo de desenvolvimento, as atividades de testes têm uma importância fundamental para a garantia de qualidade do software que está sendo desenvolvido, a qual é aplicada no decorrer de todo o projeto.

Um dos focos de *XP* é o uso de testes durante todo o processo de desenvolvimento de software de uma maneira intensiva. A idéia é desenvolver classes de testes para todas as classes do sistema, abordando pré-condições, pós-condições, asserções etc. Uma pré-condição representa a condição para que a mensagem seja aceita. Uma pós-condição define o estado obtido pela ativação do método.

Teste é a atividade do ciclo de desenvolvimento de *Software* na qual pode ser observada maior distância entre a teoria (técnicas de teste propostas na literatura) e a prática (aplicação destas técnicas) (MYE, 1979). As técnicas propostas não são aplicadas principalmente porque o teste é visto apenas como uma atividade auxiliar no desenvolvimento de software, separada das etapas iniciais (análise, projeto e codificação) (MYE, 1979). Com esta visão, os esforços gastos nesta atividade são considerados “extras”, ou seja, no início de desenvolvimento de um produto de *Software* há uma determinada previsão de custos, que acaba se excedendo, na maioria das vezes, devido às atividades de teste e de manutenção.

A depuração ocorre em conseqüência de testes bem sucedidos (testes que detectaram erros no programa). Depois de detectado um erro, a depuração tem o objetivo de removê-lo, já que o processo de depuração tenta ligar o sintoma do erro à sua causa, levando assim à correção. A depuração consiste em verificar cada linha do programa observando seu

comportamento e valores assumidos durante sua execução. Desta forma, as atividades de teste e de depuração estão intimamente relacionadas.

Para a aplicação efetiva de um critério de teste faz-se necessário o uso de ferramentas automatizadas que apoiem a aplicação desse critério. A importância de possuir testes automatizados é que contribui para reduzir as falhas produzidas pela intervenção humana, aumentando a qualidade e produtividade da atividade de teste e a confiabilidade do software, e facilitam a condução de estudos comparativos entre critérios.

O desenvolvimento de uma *Suite* de Testes auxilia na tarefa de verificação das especificações dos requisitos funcionais e não funcionais (isto é, verificar se o *Software* está fazendo as operações corretamente) de uma forma automatizada.

O uso de testes automatizados permite ao desenvolvedor verificar se mudanças no código fonte não se propagam para outras classes e outros requisitos. Para auxiliar no uso de testes automatizados vários *frameworks* (ambiente de trabalho) de testes foram desenvolvidos, como: *Junit* (JUNIT, 2002), *Nunit* (NUNIT, 2003), etc.

Os *frameworks* de testes facilitam a tarefa de definição e execução de testes provendo uma infra-estrutura na qual os testes são construídos. Para isso, foi desenvolvida uma aplicação exemplo, para que o leitor possa entender e verificar a necessidade e a adequação dos testes desenvolvidos.

Este trabalho tem como objetivo verificar a usabilidade do *Framework Junit* para testes unitários em programas Java.

Para esta aplicação foi desenvolvido um conjunto de testes visando a melhoria da implementação através da utilização de pré-condições e pós-condições, mostrando a vantagem de realizar os testes durante o processo de desenvolvimento de software. Foi utilizado o ambiente integrado de desenvolvimento de aplicações Eclipse (IBM) porque possui *JUnit* integrado, pode rodar em qualquer plataforma, ser usado para escrever programas em várias linguagens, exemplos de construção, tornando mais fácil a criação, integração e utilização das ferramentas.

Este trabalho tem a seguinte estrutura: Neste capítulo foram apresentadas sobre os testes de software, suas importâncias, a fase de correção dos erros encontrados no processo de desenvolvimento, os tipos de testes como, por exemplo, a *Suíte* de testes, a importância de utilizar testes automatizados, o Framework Junit. O capítulo 2 apresenta a revisão de literatura, contendo a evolução da linguagem Java, uma descrição mais detalhada sobre os testes de software e testes de software orientados a objetos e sobre o *Framework Junit*. No

capítulo 3 tem-se o material e método utilizado, como o Junit, a linguagem Java, a ferramenta eclipse, *rational rose* e as pesquisas bibliográficas. No capítulo 4 estão os resultados e discussões com o diagrama de classes, uma descrição detalhada dos testes e sobre a ferramenta utilizada na realização dos testes. O capítulo 5 apresenta as considerações finais juntamente com a experiência adquirida e as dificuldades encontradas. No capítulo 6 estão as referências bibliográficas e no capítulo 7 está o apêndice com todo o código da implementação das classes testadas.

2 REVISÃO DE LITERATURA

Esta seção fundamentará as teorias sobre o *Framework JUnit*, a Linguagem de Programação Java e Testes de *Software* utilizados na construção deste projeto, permitindo que o leitor compreenda sua utilização e importância no contexto do Processo de Desenvolvimento de *Software*.

2.1 Linguagem Java

A linguagem Java foi concebida em 1995 pela *Sun Microsystems*, sendo baseada nas linguagens C e C++, introduzindo algumas modificações, melhorando e corrigindo deficiências dessas linguagens. As mudanças refletiram tornando a linguagem portável e adequada ao desenvolvimento de aplicativos para *Web*, incluindo novos recursos bastante úteis às necessidades dos usuários, como, por exemplo, a utilização de áudio. A partir daí a linguagem foi disponibilizada gratuitamente para milhões de programadores no mundo todo.

Java possibilitou adicionar as páginas *Web* conteúdo dinâmico, em vez de apenas conteúdos estáticos. As páginas poderiam ter vídeos, animações e interatividade. Mas o que tornou a Java uma das principais linguagens de programação foram os recursos que as organizações precisavam para o processamento de informações. Deitel (2003) afirma que, a Java revolucionou o desenvolvimento de *softwares* que possuem código orientado a objetos que usa intensivamente multimídia, independente de plataforma, para aplicações convencionais baseadas em *internet*, *intranet*.

A portabilidade é obtida pelo fato da linguagem ser interpretada, ou seja, o compilador gera um código independente de máquina chamado *bytecode*. Os *bytecodes* são a linguagem da máquina virtual Java – um programa que simula a operação de um computador e roda sua própria linguagem de máquina (Deitel, 2003). No momento da execução este *bytecode* é

interpretado por uma máquina virtual instalada na máquina. Para portar Java para uma arquitetura *hardware* específica, é necessário instalar a máquina virtual (interpretador) (Deitel, 2003).

Java é uma linguagem de programação baseada em classes e orientada a objetos, pois a linguagem permite a reutilização do código. Um conjunto de palavras-chave pode ser usado para caracterizar a Java. A Java é uma linguagem interpretada, segura, de arquitetura neutra, de alto desempenho (Deitel, 2003).

2.2 Teste de Software

A atividade de teste é uma etapa fundamental para a garantia da qualidade de software e representa a última revisão do processo de engenharia de software.

O desenvolvimento de sistemas de software envolve muitas atividades de produção em que há grande probabilidade de falhas humanas ocorrerem. Os erros podem começar a acontecer logo no começo do processo, onde os requisitos podem estar especificados de forma errônea, além de erros que venham a ocorrer em fase de projeto e desenvolvimento posteriores. Devido à incapacidade que os seres humanos tem de executar e comunicar com perfeição, o desenvolvimento de software é acompanhado por atividades de testes que garantem a qualidade (DEU, 1979).

O objetivo principal do teste de software é tornar o software confiável. É projetar testes que detectam classes de erros e que esse processo seja realizado com pouca quantidade de tempo e esforço. Se a atividade de testes for bem planejada, ou seja, com a intenção de descobrir um erro, que seja capaz de revelar um erro ainda não encontrado, o objetivo será alcançado.

Quando os testes são realizados, todos os resultados são avaliados. Isto é, os resultados de teste são comparados com os resultados esperados. Ao encontrar um erro, é iniciada a fase de depuração, ou seja, correção do erro encontrado. À medida que os resultados são avaliados é feita uma avaliação do software, em relação a sua qualidade e confiabilidade. Se os erros encontrados forem graves e freqüentes podendo ocasionar mudança no projeto, a qualidade e confiabilidade do software são suspeitas. Caso contrário, o software é aceitável ou os testes não são bons suficientes para detectar erros graves.

Tanto o teste de software quanto à depuração são atividades relacionadas diretamente à validação do software. A "validação" é freqüentemente confundida com a "verificação de software", sendo, inclusive, atividades bastante confusas entre si. A verificação garante que o software implementa corretamente uma função específica, enquanto que a validação garante que o software que foi construído é adequado aos requisitos do cliente (BOE, 1981).

As fases de teste e manutenção consomem aproximadamente 60% dos recursos utilizados no desenvolvimento de software (MYE, 1979). Alguns motivos que levam ao aumento dos custos nessas fases são os seguintes:

- a falta de preocupação em testar por parte dos programadores;
- omissão de requisitos e requisitos especificados erroneamente. Não sendo possível fazer a comparação das saídas geradas com as esperadas;
- a quantidade de ferramentas CASE (*Computer Aided Software Engineering*) que automatizem as técnicas de teste e manutenção;
- o caráter evolutivo dos sistemas de software, devido à adição de funcionalidades e à evolução das mesmas.

2.2.1 Teste de Unidade

O Teste de unidades concentra-se na verificação do módulo – menor unidade do projeto de software. Esses testes baseiam-se nos testes de caixa branca, e esse passo pode ser realizado em paralelo (PRE, 1995).

Uma unidade define um conjunto de estímulos (chamada de métodos), e de dados de entrada e saída associados a cada estímulo. As entradas são parâmetros e as saídas são valor de retorno, exceções, estado do objeto.

Ignoram condições ou dependências externas. Testes de unidade usam dados suficientes para testar apenas a lógica da unidade em questão, detectando um maior número de erros e corrigindo-os mais facilmente do que nos outros tipos de testes. Testes de unidade devem ser executados o tempo todo.

2.2.2 Teste de Integração

O teste de integração é uma técnica complexa para a construção da estrutura de programa, e ao mesmo tempo realiza testes para descobrir erros associados à interface (PRE, 1995). Essa complexidade ocorre principalmente devido à dificuldade de integrar os módulos de unidades anteriormente testados e construir a estrutura de programa a qual foi determinada pelo projeto formando a interface.

2.2.3 Teste de Sistema

O teste de sistema é um conjunto de testes diferentes, com a finalidade principal de testar o sistema completo de *software*, ou seja, depois de realizados os testes de unidade, integração e validação os quais têm suas formas de testar diferentes, todo o trabalho deve verificar se os elementos do sistema foram adequadamente integrados e realizam as funções atribuídas (PRE, 1995).

2.3 Teste de Software Orientado a Objetos

O teste de software é um processo de encontrar falhas. Testar pode contribuir para a qualidade e confiabilidade ajudando a identificar os problemas mais cedo no processo de desenvolvimento de software (McGregor, 2001).

Além das dificuldades existentes na realização dos testes, não existe metodologias e ferramentas adequadas, elevando os custos das atividades (McGregor, 2001).

Apesar do paradigma Orientado a Objetos (OO) permitir construir *softwares* de melhor qualidade e proporcionar rapidez no desenvolvimento, a realização de testes nesse paradigma é mais complexa devido à hierarquia de classes, encapsulamento, polimorfismo etc (McGregor, 2001).

2.3.1 Teste Estrutural

O teste estrutural também é conhecido como teste caixa-branca (*White-Box*) ou teste caixa aberta. Nesse tipo de teste, os casos de teste são realizados a partir da análise da estrutura interna do programa. O objetivo dos casos de teste é causar a execução de caminhos identificados no programa, baseados no fluxo de controle e/ou no fluxo de dados. Os principais problemas desta abordagem são:

- programas com laços de repetições que possuem um número infinito de caminhos, já que a análise da estrutura interna do programa é feita estaticamente. Desta forma, seria necessário aplicar o teste exaustivo ao programa, o que é considerado impraticável (MYE, 1979);
- existência de caminhos não executáveis no programa, que ocasionam o desperdício de tempo e recursos financeiros na tentativa de gerar casos de teste que possam executar estes caminhos (MYE, 1979);
- a execução bem sucedida de um caminho do programa selecionado não garante que este esteja correto, já que com outro caso de teste um erro pode ocorrer.

Para diminuir os problemas identificados anteriormente, são utilizados critérios de seleção de caminhos, de acordo com as características nas quais mais se baseiam: fluxo de controle e fluxo de dados.

Estes critérios, conhecidos como "critérios de cobertura", ou "critérios de seleção", são condições que devem ser preenchidas pelo teste, as quais selecionam determinados caminhos que visam cobrir o código ou a representação gráfica deste. Um critério é válido se a execução de pelo menos um dos casos de teste detectar erros no programa. E é ideal se fornecer um conjunto de casos de teste que detecte todos os erros do programa que o critério se propõe a detectar (BOE, 1981).

2.3.2 Teste Funcional

O teste funcional também é conhecido como teste de caixa preta (*black Box*). Os métodos de teste de caixa preta concentram-se nos requisitos funcionais do software. As

técnicas de teste funcional derivam os casos de teste a partir da análise da funcionalidade (dados de entrada/saída e especificação) do programa, sem levar em consideração a estrutura interna do mesmo (MYE, 1979).

A abordagem funcional tem o objetivo de complementar a abordagem que as técnicas do teste estrutural apresentam. As categorias de erros mais evidenciadas pelo teste funcional são: erros de interface, funções incorretas ou ausentes, erros nas estruturas de dados ou no acesso a bancos de dados externos, erros de desempenho e erros de inicialização e término. O teste funcional é geralmente aplicado quando todo ou quase todo o sistema já foi desenvolvido. O conjunto de casos de teste derivado no teste funcional deve satisfazer os seguintes critérios (MYE, 1979):

- reduzir o número de casos de teste adicionais que devem ser projetados para se conseguir testes satisfatórios;
- revelar algo sobre a presença ou ausência de classes de erros, em vez de um erro associado somente ao teste específico que se está utilizando.

As técnicas mais clássicas de teste funcional para software procedimental são: particionamento de equivalência – o qual procura definir um caso de teste que descubra classes de erros, reduzindo o número total de casos de teste que devem ser desenvolvidos, análise do valor-limite – complementa o particionamento de equivalência, o qual visa selecionar os casos de teste nas “extremidades” da classe, de forma a colocar em prova os valores fronteiriços e grafo causa-efeito – que oferece uma representação das condições lógicas e das ações correspondentes. A causa é a condição de entrada, o efeito é uma ação.

2.4 Junit

Junit é um *Framework* (ambiente de trabalho) que possibilita a criação de testes em Java (E. Gamma/K. Beck).

O objetivo do desenvolvimento do *Framework* é facilitar a criação de casos de teste, que programadores de fato usarão para escrever testes de unidade, para que possa exigir menos do usuário pelo fato de se utilizar uma ferramenta que automatize os testes de

Software, evitando escrever testes duplicados e permitir escrever testes que retenham seu valor ao longo do tempo, ou seja, que possam ser reutilizáveis (DSC/UFCG, 2001).

O *Framework* é simples para a criação das classes de testes. Estas classes contêm um ou mais métodos para que sejam realizados os testes, podendo ser organizados de forma hierárquica, de forma que o sistema seja testado em partes separadas, algumas integradas ou até mesmo todas de uma só vez (D'ÁVILA, 1995).

Toda classe de teste é uma subclasse da classe *TestCase* que é declarada no pacote do *junit.framework* localizado no arquivo *junit.jar*. O nome da classe pode ser qualquer um, já para os métodos (unidades de teste) deve seguir o padrão testNomeDoMétodo(), esse método deve ser público e sem argumentos (DSC/UFCG, 2001).

TestCase é a classe que chama os métodos das classes para realizar os testes. Podendo ser realizado um único teste ou vários testes ao mesmo tempo, através do objeto composto *TestSuite* (D'ÁVILA, 1995).

TestSuite é uma classe que implementa *Test*, mas o método *run*, o qual é diferente do método *run* de *TestCase*, invoca o método *run* de cada um dos testes que a *suíte* de testes contém (DSC/UFCG, 2001).

TestRunner é uma classe de interface de usuário que invoca o *Framework* e exibe os resultados. Há uma versão gráfica (*junit.swingui*) e uma versão com textos (*junit.textui*) (DSC/UFCG, 2001).

O Junit possui também o *TestResult* o qual armazena os resultados dos testes.

Classes de teste devem estar no mesmo pacote que as classes testadas para que JUnit tenha acesso a elas.

O Framework Junit permite a automação dos testes, o próprio framework e suas extensões são free e open source, permite a implementação das classes e métodos de testes, disponibiliza a execução dos testes de forma visual, através da interface gráfica, e textual, permite criação de testes individuais para os métodos pertencentes a uma mesma classe, permite a definição e execução de um conjunto de testes individuais, relato de problemas ocorridos após a realização das baterias de testes com identificação específica de onde ocorreram as falhas, etc.

3 MATERIAIS E MÉTODOS

Com apresentação desta seção é possível saber quais as ferramentas utilizadas no desenvolvimento deste trabalho, bem como os motivos de sua utilização.

Para o desenvolvimento deste trabalho foram utilizadas as normas estabelecidas pela ABNT.

3.1 Materiais

Este trabalho foi desenvolvido nos laboratórios do CEULP/ULBRA, no período de agosto a dezembro de 2003, com orientações de 2 horas semanais. Utilizou-se de um microcomputador com processador Pentium IV de 2.4 GHz, 256 memória RAM, HD 40 GB, sistema operacional *windows 2000 Professional.*

3.1.1 JUnit

Esta ferramenta foi utilizada porque permite a verificação dos testes de forma visual e textual, evita que o desenvolvedor de casos de testes refaça testes que já foram feitos e seus resultados obtidos com sucesso. O uso da ferramenta foi possível porque o fabricante disponibiliza gratuitamente a ferramenta para quem tiver interesse em utilizá-la.

Com a utilização desta ferramenta obteve-se rapidez na execução de testes das classes que foram desenvolvidas, pois se trata de testes automatizados, não sendo necessário fazer laços de repetição para testar.

3.1.2 Linguagem de Programação Java

Esta linguagem foi utilizada neste projeto para fazer a implementação das classes para a realização dos testes.

Poderia ser escolhida outra Linguagem já que o Ambiente *Eclipse* permite implementação em outras Linguagens de Programação, mas como Java é uma linguagem de Programação muito utilizada e por ser baseada em C/C++ foi mais fácil utilizá-la para esta aplicação, pois há conhecimento nestas Linguagens na qual Java foi baseada.

3.1.3 Eclipse

Esta ferramenta é fornecida pela IBM e foi utilizada para criação das classes Java, pois além de ser usada para escrever programas em Java permite que seja feito em outras linguagens. Possui um ambiente integrado de desenvolvimento. Possui também *Junit* integrado, pode rodar em qualquer plataforma, permite interação na construção do *Software*, tornando mais fácil à criação, integração e utilização das ferramentas.

3.1.4 Rational Rose

A ferramenta foi utilizada para a criação do Diagrama de Classes, o qual apresenta as classes utilizadas nos testes.

3.2 Métodos

Inicialmente os métodos utilizados no desenvolvimento e para o sucesso deste trabalho foram: feitas algumas pesquisas em livros, consultas a *Internet*, trabalhos anteriores e orientações do orientador M.Sc. Eduardo Kessler Piveta.

3.2.1 Pesquisa Bibliográfica

A pesquisa bibliográfica foi realizada com material da biblioteca do CEULP/ULBRA e material da internet, dando apoio teórico sobre as linguagens e ferramentas utilizadas neste projeto.

4 RESULTADOS E DISCUSSÕES

A figura 1 apresenta o resultado dos testes o qual foi realizado em todos os métodos das subclasses, **ContaTest** e **BancoTest**. Nesta figura, é mostrada uma barra de cor verde no caso de todos os testes terem sido realizados com sucesso. Se algum dos testes falhar ou houver erro, a cor da barra fica vermelha. Nota-se que abaixo da barra há um resumo que indica quantos testes foram feitos (*Runs*), quantos erros ocorreram (*errors*) e quantos falharam (*Failures*).

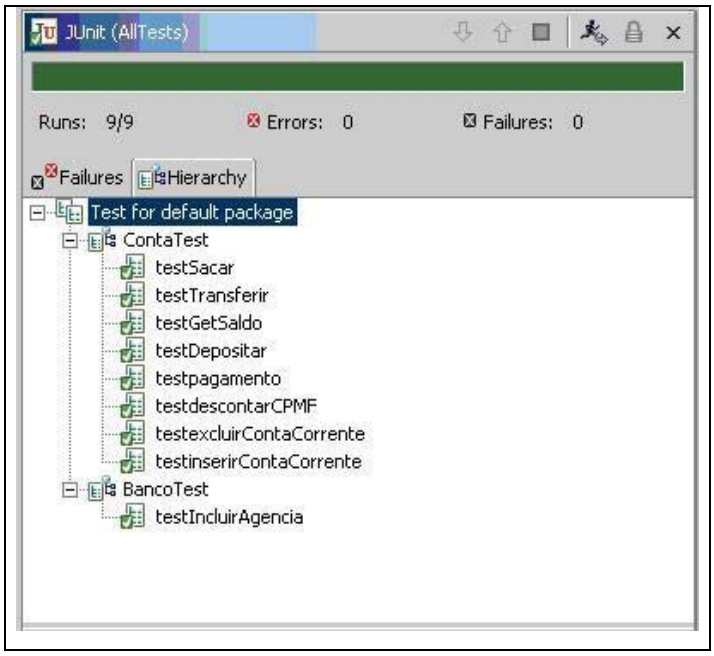

Figura 1 – Exemplo do resultado de uma **Suite de testes.**

4.1 Classes utilizadas para os Testes

O diagrama da figura 2 representa a modelagem das classes utilizadas para a realização dos testes utilizando um domínio bancário. Para as classes que foram testadas, foram criadas classes de testes as quais possuem um nome *test* padrão, antes do nome da classe, ou seja, *test*<nome da classe>, por exemplo, *testBanco*.

Neste domínio, um Banco possui uma ou mais Agências. Uma Agência possui um ou mais Gerentes e uma ou mais Contas que pode ser Conta Corrente e/ou Conta Poupança. Uma Conta pertence a um ou mais Clientes que pode ser Pessoa Física e/ou Pessoa Jurídica.

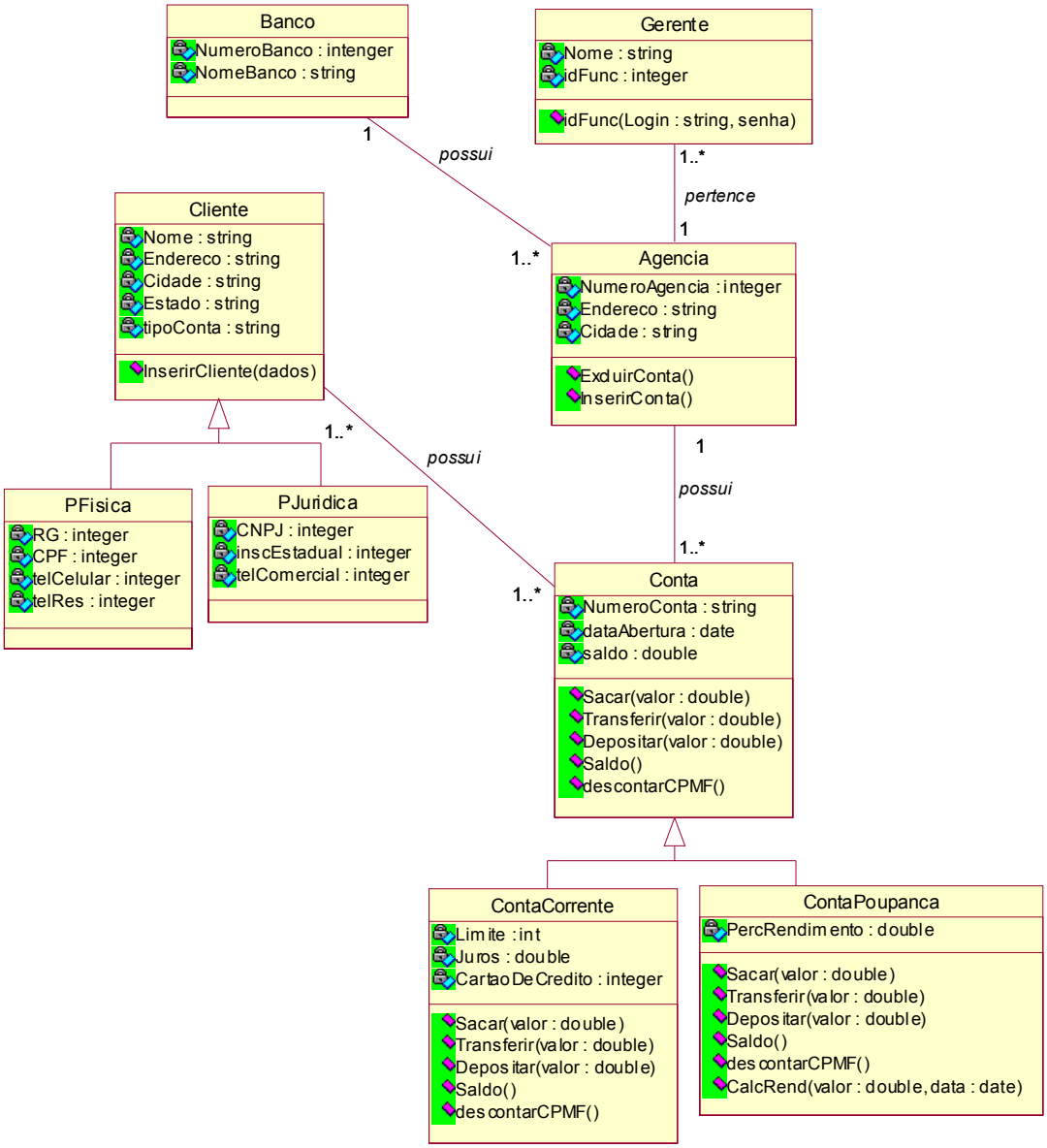

Figura 2 – Diagrama de classes.

O diagrama apresentado mostra um conjunto de classes, interfaces e seus relacionamentos.

Sobre esse domínio foram realizados os testes apenas com as classes Banco e Conta, devido às mesmas serem suficientes para apresentar os resultados com a ferramenta *Junit*.

4.2 Descrição das estratégias de Testes

Após a implementação das classes Banco, Conta, testBanco e testConta, foram iniciados os testes. O Junit fornece tanto uma interface gráfica como uma textual, e ambas indicam a quantidade de testes que foram executados, algum erro ou falha e um resumo complementar.

Será exibido o código de três métodos para melhor entendimento do funcionamento dos testes. Logo abaixo, o código das Classes TestCase, o resultado dos testes realizados nos métodos das classes testBanco e testConta, apresentando a interface gráfica do Junit.

4.3 Métodos da Classe Conta Testados

4.3.1 TestSacar – Resultado Positivo

```
1 Import junit.framework.TestCase;
2 Public class ContaTest extends TestCase {
3     public void testSacar () {
4           Conta c = new ContaCorrente();
5           c.setSaldo(200);
6           c.sacar(100);
7           assertTrue(c.getSaldo()==99.62);
8     }
```

Figura 3 – Código do método **testSacar()**

A figura 3 mostra o código do método **testSacar()**, onde o resultado dos testes foram positivos. Inicialmente uma conta que possui R$200,00 de saldo (linha 5), ao ser realizada uma transação de saque no valor de R$100,00 da mesma (linha 6), o valor do saldo atualizado com o desconto de CPMF é de R$99,62 (linha 7). Na figura 4, pode ser visto o resultado do teste na interface do **Junit**.

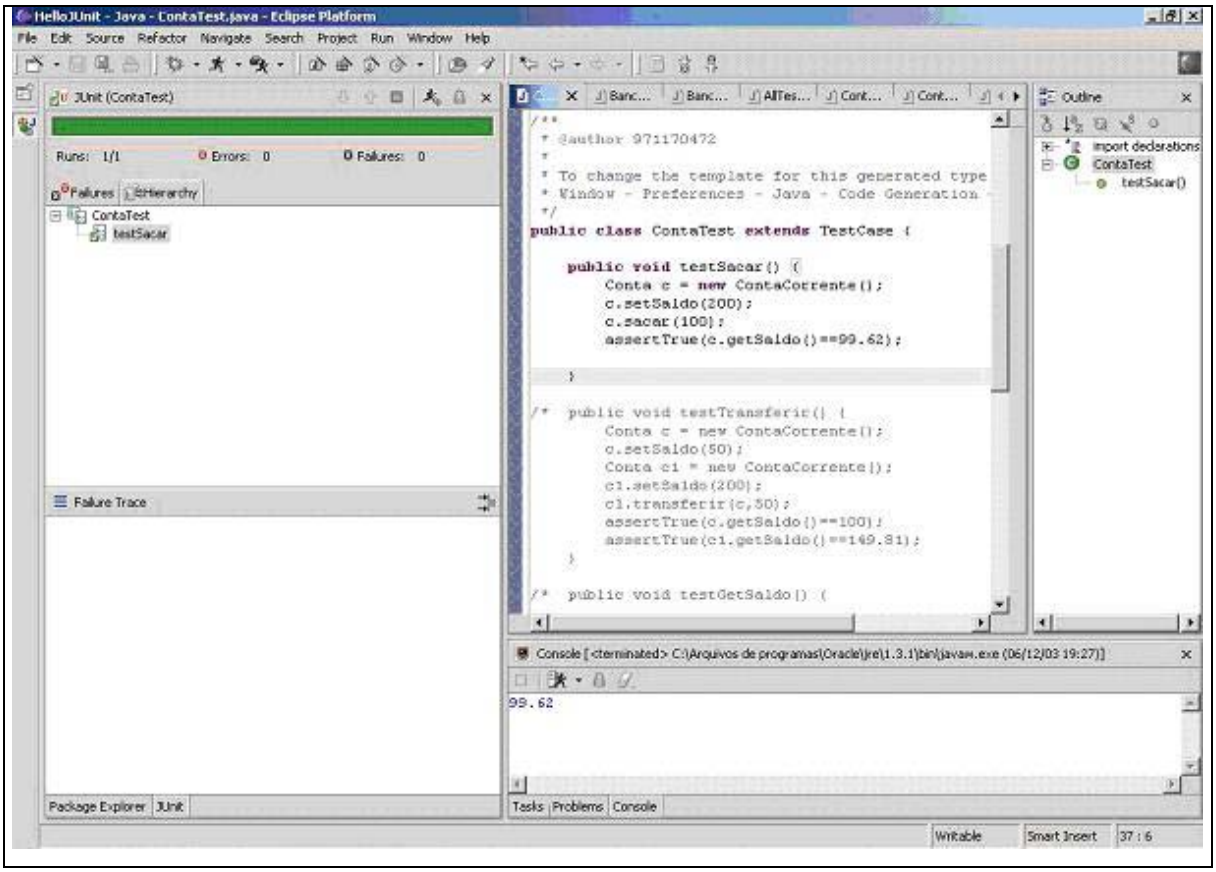

Figura 4 – Resultado do teste método **testSacar()**, sem alteração no código.

4.3.2 TestSacar – Resultado Negativo

```
1 Import junit.framework.TestCase;
2 Public class ContaTest extends TestCase {
3     public void testSacar() {
4           Conta c = new ContaCorrente();
5           c.setSaldo(100);
6           c.sacar(100);
7           assertTrue(c.getSaldo()==99.62);
8     }
```

Figura 5– Código do método **testSacar()** com alteração

A figura 5 mostra o código do método **testSacar()**, com a 5.ª linha (**c.setSaldo(100)**) alterada propositadamente, a fim de provocar uma falha. Pois o Saldo da Conta atualizado após a operação de saque teria que receber um valor negativo e verificado com o valor do

assertTrue, o qual retorna um valor booleano como resultado do teste, observa-se a diferença nos valores. Na figura 6, pode ser visto o resultado do teste na interface do **Junit.**

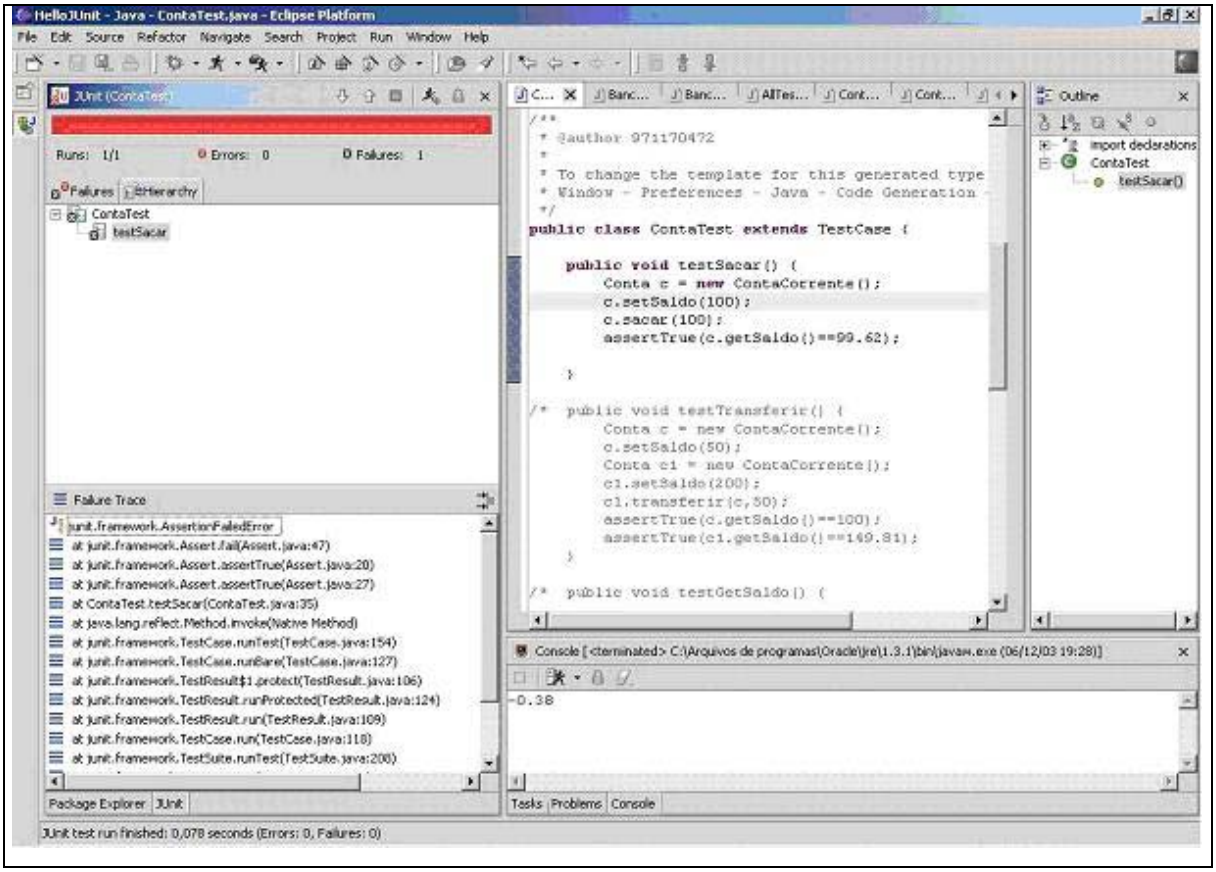

Figura 6 – Resultado do teste método **testSacar()**, com alteração no código.

4.3.3 TestTransferir – Resultado Positivo

```
 1 Import junit.framework.TestCase;
 2 Public class ContaTest extends TestCase {
 3    public void testTransferir() {
 4          Conta c = new ContaCorrente();
 5          c.setSaldo(50);
 6          Conta c1 = new ContaCorrente();
 7          c1.setSaldo(200);
 8          c1.transferir(c,50);
 9          assertTrue(c.getSaldo()==100);
10          assertTrue(c1.getSaldo()==149.81);
11     }
```

```
12  }
```

Figura 7 – Código do método **testTransferir()**

A figura 7 mostra o código do método **testTransferir()**, onde o resultado dos testes foram positivos. Na figura 8, pode ser visto o resultado do teste na interface do **Junit**.

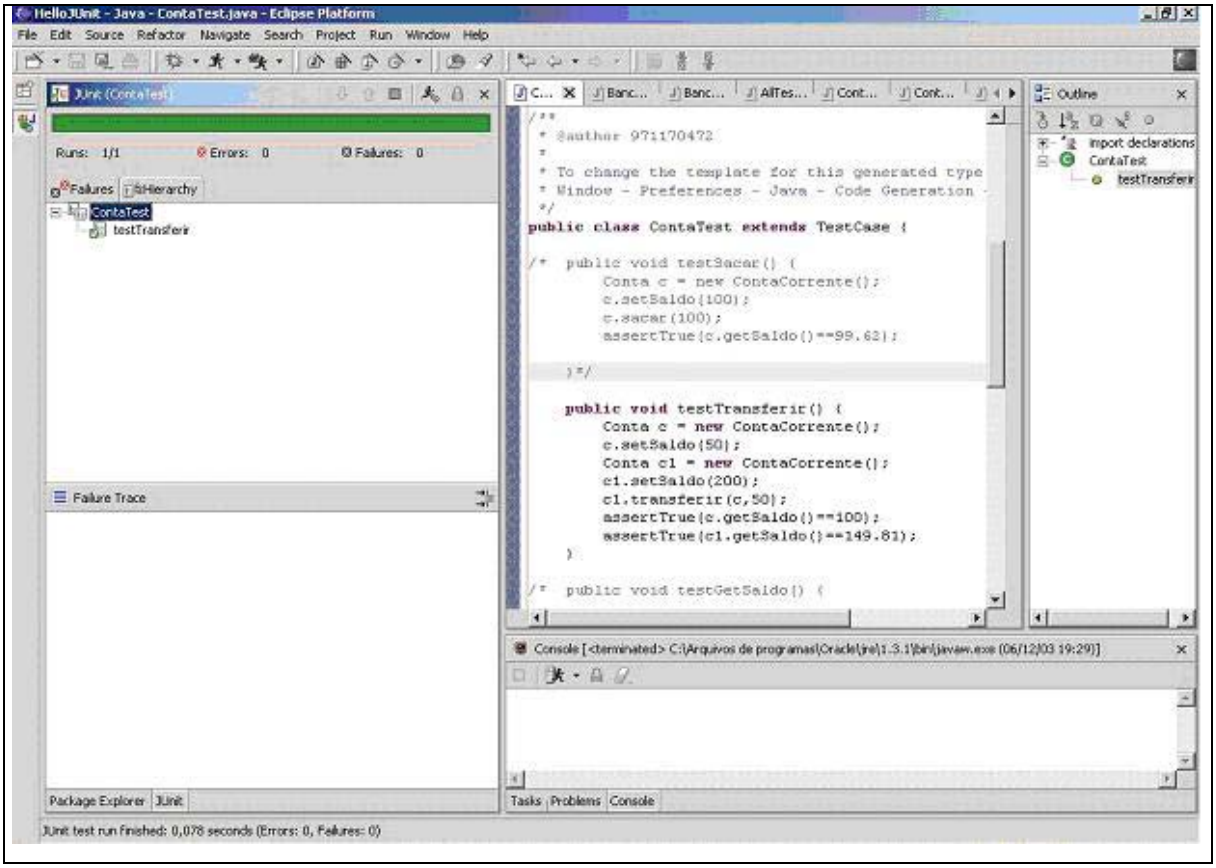

Figura 8 – Resultado do teste método **testTransferir()**, sem alteração no código.

4.3.4 TestTransferir – Resultado Negativo

```
1  Import junit.framework.TestCase;
```

```
 2  Public class ContaTest extends TestCase {
 3    public void testTransferir() {
 4          Conta c = new ContaCorrente();
 5          c.setSaldo(50);
 6          Conta c1 = new ContaCorrente();
 7          c1.setSaldo(100);
 8          c1.transferir(c,50);
 9          assertTrue(c.getSaldo()==100);
10          assertTrue(c1.getSaldo()==149.81);
11    }
12  }
```

Figura 9– Código do método **testTransferir()** com alteração

A figura 9 mostra o código do método **testTransferir()**, com a 7.ª linha (**c.setSaldo(100)**) alterada propositadamente, a fim de provocar uma falha. Pois o Saldo da Conta atualizado é diferente do resultado verificado com o **assertTrue**, o qual retorna um valor booleano, como resultado do teste. Na figura 10, pode ser visto o resultado do teste na interface do **Junit.**

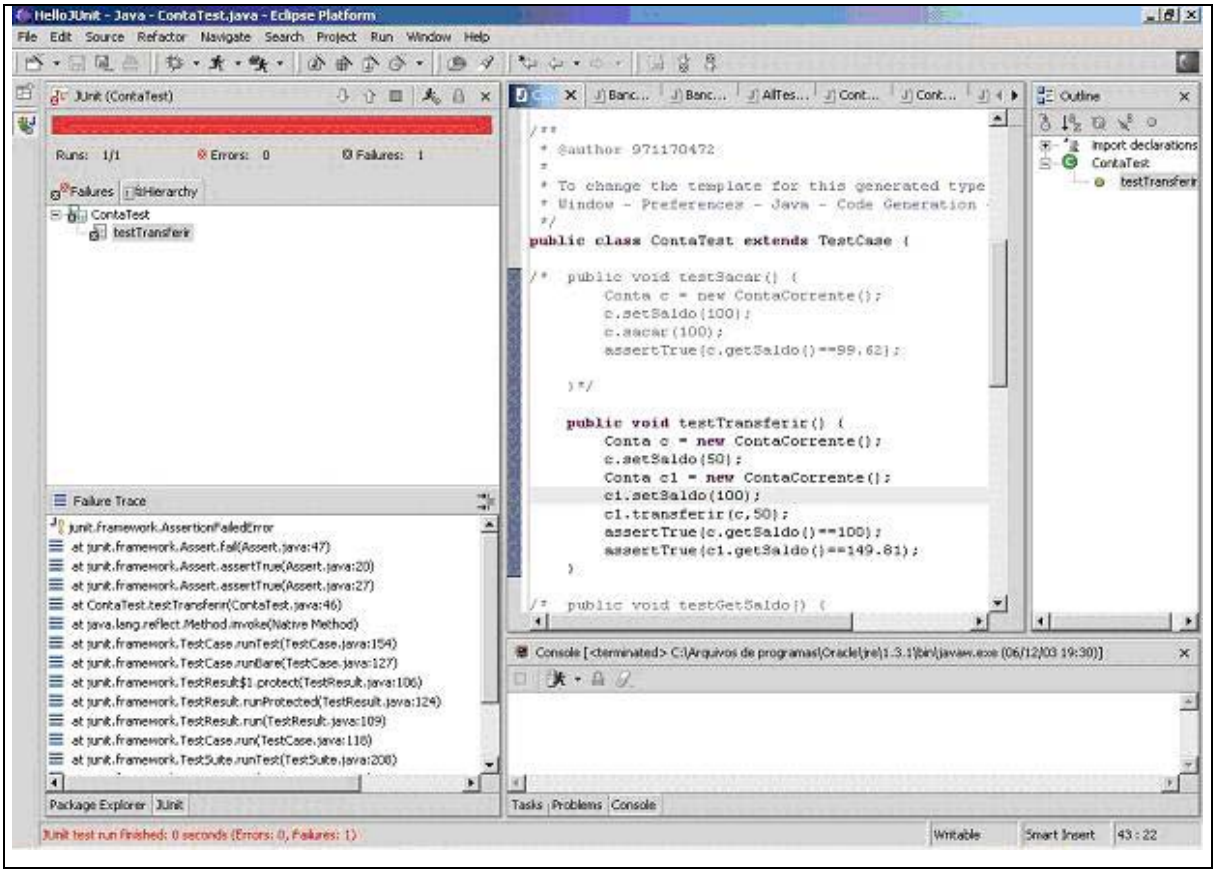

Figura 10 – Resultado do teste método **testTransferir()**, com alteração no código.

5 CONSIDERAÇÕES FINAIS

Com este trabalho foi possível realizar testes de s*oftware* utilizando o **Junit**, que foi desenvolvido com a finalidade de facilitar as tarefas de testes para o programador, visto que automatiza tais tarefas.

Através dos testes foi possível comprovar que o **Junit** proporciona realmente facilidades e rapidez nos testes, pois diminui o trabalho do programador na elaboração e execução dos testes, de forma que o mesmo não necessita ficar rescrevendo-os.

A vantagem de se trabalhar com ferramentas de testes automatizados, é que diminui os erros provocados pelo programador, torna o software mais confiável, possibilita observar se mudanças no código se propagam para as classes e métodos.

O programador conta com uma ferramenta que permite a verificação do que realmente acontece com classes implementadas de formas errôneas. Proporcionando também ajuda ao desenvolvedor para o sucesso de seu trabalho, diminuindo os riscos de cancelamento do projeto.

5.1 A experiência adquirida

A realização deste trabalho proporcionou abrangências ao meu conhecimento, na minha formação enquanto profissional, me preparando para disputar o tão concorrido mercado de trabalho, através da vantagem que um programador tem usando uma ferramenta de testes automatizados.

Foi adicionado ao meu currículo profissional mais uma linguagem de programação, a linguagem JAVA, que possibilita o desenvolvimento de aplicações *Web*, de distribuição gratuita e funcional multiplataformas.

A inter-relação destas ferramentas me permite oferecer programas mais confiáveis para o mercado de trabalho, para empresas que se encontram ou não informatizadas, diminuindo ao máximo os custos que se tem quando se desenvolve software sem testar, ou que são testados somente ao final do ciclo de desenvolvimento.

Ao desenvolver este trabalho colaborei para que meus colegas de faculdade tivessem conhecimento sobre o *Framework Junit*, seu funcionamento e a criação de tipos de testes, como, por exemplo, testes de unidades, integração, etc., um projeto de testes que facilita e minimiza custos do ciclo de desenvolvimento do *Software*.

Este trabalho pode ser utilizado como uma fonte de referência às pessoas que precisam trabalhar com testes de *Softwares*, tendo como base e podendo ser utilizado o *Framework Junit*.

Atualmente existem organizações e autônomos que sofrem muito com a falta de utilização de testes durante o processo de desenvolvimento de *Softwares*, podendo levar ao cancelamento do projeto, dependendo do grau de impacto que a correção tardia do erro pode causar ao projeto. Contribuirei para estas pessoas, disponibilizando a minha proposta de desenvolvimento a documentação do meu trabalho.

A exploração do *Junit*, pois a ferramenta possui inúmeras utilidades e facilidades para o desenvolvedor de casos de testes, mostrando à localização precisa dos erros, ajudando na implementação das classes, etc.

5.2 As dificuldades encontradas

Inicialmente, as dificuldades foram: a falta de literatura para uma maior exploração do *Framework Junit*, a pouca prática com a Linguagem de Programação Java e dúvidas que surgiam durante o estudo e desempenho do trabalho.

Mas com tempo e a disponibilidade do mesmo, o trabalho passou a ganhar forma e estrutura. A Linguagem Java que estava sendo estudada e utilizada em outro trabalho foi sendo entendida para o processo de desenvolvimento deste trabalho. Isso através do auxilio do meu orientador Eduardo Kessler Piveta.

6 REFERÊNCIAS BIBLIOGRÁFICAS

BOEHM, B. **Software Engineering Economics,** Prentice Hall. 1981

DEITEL, H. M. **Java, como programar** / H. M. Deitel e P. J. Deitel; trad. Carlos Arthur Lang Lisboa. – 4. ed. – Porto Alegre: Bookman, 2003.

DEUTSCH, M. "**Verification and Validation", in Software Engineering**. (R. Jensen e C. Tonies, Eds), Prentice Hall, 1979

MCGREGOR, John D.; SYKES, A. David. **Pratical Guide to Testing Object-Oriented Software,** Boston, Series Editors, 2001.

MYERS, G. **The Art of Software Testing,** Wiley, 1979

PRESSMAN, Roger S. **Engenharia de Software**. São Paulo: Makron Books, 1995

Sites Pesquisados:

CIRNE, Walfredo, Campina Grande, 2000. Disponível em:

<http://walfredo.dsc.ufpb.br/cursos/2002/progII20021/aulas/testes.htm>, acessado em 21/10/2003

COMUNIDADE ECLIPSE, Pontifícia Universidade Católica do Rio de Janeiro, nov. 2003. Disponível em:

<http://web.teccomm.les.inf.puc-rio.br:8668/eclipse/space/Eclipse>, acessado em 10/11/2003

D'ÁVILA, Márcio H. C., Márcio's Hyperlink, set. 2003. Disponível em:

http://www.mhavila.com.br/link/prog/soft-eng.html, acessado em 15/09/2003.

HÜBNER, Jomi Fred, Universidade Federal de Blumenau (FURB), nov. 2003. Disponível em:

<http://www.inf.furb.br/~jomi/java/apresentacoes/JavaBasico/sld009.htm>, acessado em 13/11/2003

INSIDE INFORMATION SYSTEMS, Rio de Janeiro, 2003. Disponível em:

< http://www.inside.com.br/~coriceu/prodsoft/14teste.htm >, acessado em 28/10/2003

OSDN, Open Source Development Network, New York, out. 2002. Disponível em:

<http://prdownloads.sourceforge.net/junit/junit3.8.1.zip?download>, acessado em 30/10/2003

OSDN, Open Source Development Network, New York, out. 2003. Disponível em:

<http://sourceforge.net/projects/nunit/>, acessado em 30/10/2003.

REVISTA DO LINUX, Conectiva S/A, out 2003. Disponível em:

<http://www.revistadolinux.com.br/ed/041/assinantes/eclipse.php3>, acessado em 05/11/2003

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE, Departamento de Sistemas e Computação, 2001. Disponível em:

<http://www.dsc.ufpb.br/~jacques/cursos/map/html/frame/junit.htm>, acessado em 10/10/2003

7 APÊNDICE

Código para implementação das Classes

```
//Implementação da Classe Conta

Import java.util.Vector;
Public abstract class Conta {
     public double saldo;
     public Vector contas = new Vector(0,1);

     public void sacar(double valor){
          setSaldo(getSaldo()- valor);
     }

     public void transferir(Conta destino, double valor){
          destino.setSaldo(destino.getSaldo() + valor);
          setSaldo(getSaldo() - valor);
     }

     public void depositar(float valor){
               saldo+=valor;
     }

     void setSaldo(double saldo) {
          this.saldo = saldo;
     }

     double getSaldo() {
          return saldo;
     }

     public void pagamento(double valorTitulo){
          setSaldo(getSaldo() - valorTitulo);
          descontarCPMF(valorTitulo);
     }


     protected void descontarCPMF(double valor) {
          setSaldo(getSaldo()-(valor*0.0038));
     }
```

```
}


//Implementação da Classe Banco

import java.util.Vector;

public class Banco {

     Vector agencias;
     Vector clientes;

     Banco(){
        agencias = new Vector(0,1);
        clientes = new Vector(0,1);
     }

     public boolean incluirAgencia(Agencia a){
          if (!agencias.contains(a)) {
             agencias.addElement(a);
             return true;
          }
          else
             return false;
     }

     public boolean inserirCliente(Clientes c){
          if (!clientes.contains(c)){
               clientes.addElement(c);
               return true;
          }
          else
          return false;
     }

     public static void main(String[] args) {

     }
}




//Implementação da Classe Conta Corrente

public class ContaCorrente extends Conta{


     public void sacar(double valor){
          super.sacar(valor);
          descontarCPMF(valor);
          System.out.println(getSaldo());
     }
```

```
     public void transferir(Conta destino, double valor){
          super.transferir(destino,valor);
          descontarCPMF(valor);
     }

     public static void main(String[] args) {
          Conta c = new ContaCorrente();
          c.setSaldo(100);

           c.sacar(50);
          System.out.println(c.getSaldo());
     }

}



//Implementação da Classe Agência

import java.util.Vector;

public class Agencia {

     String nome;
     Vector contas;

     Agencia(){
          Contas = new Vector(0,1);
     }

     public void setNome(String aNome){
        nome = aNome;
     }

     public String getNome(){
        return nome;
     }


     public void excluirContaCorrente(Conta c){
          if(contas.contains(c)){
               contas.removeElement(c);
          }
     }

     public void inserirContaCorrente(Conta c){
          if(!contas.contains(c)){
               contas.addElement(c);
          }
     }

     public static void main(String[] args) {

     }
}
```

```
//Implementação a Classe Cliente


public class Cliente {
     private String nome;
     private String endereco;
     private String cidade;
     private String estado;

     private int idCliente;


     public void setNome(String nome) {
               this.nome = nome;
     }

     public String getNome() {
          return nome;
     }

     public void setEndereco(String endereco) {
               this.endereco = endereco;
     }

     public String getEndereco(){
          return endereco;
     }

     public void setCidade(String cidade) {
               this.cidade = cidade;

     }


     public String getCidade() {
          return cidade;
     }

     public void setEstado(String estado) {
               this.estado = estado;
     }

     public String getEstado() {
          return estado;
     }

     public void setIdCliente(int idCliente) {
               this.idCliente = idCliente;
     }

     public int getIdCliente() {
               return idCliente;
     }

     public void cadastrarCliente(Cliente c){
          Clientes cliente = new cliente();
```

```
          Clientes.inserir(c);
     }

}



//Implementação da Classe Cliente Pessoa Física


public class ClientesPFisica extends Clientes{

     private int RG;
     private int CPF;
     private int telCelular;
     private int telRes;

     public void setRG(int RG) {
                    this.RG = RG;
     }

     public int getRG(){
               return RG;
     }

     public void setCPF(int CPF) {
               this.CPF = CPF;
     }

     public int getCPF(){
               return CPF;
     }

     public void settelCelular(int telCelular) {
               this.telCelular = telCelular;
     }

     public int gettelCelular(){
               return telCelular;
     }

     public void settelRes(int telRes) {
               this.telRes = telRes;
     }

     public int gettelRes(){
               return telRes;
     }

     public static void main(String[] args) {
     }
}



//Implementação da Classe Cliente Pessoa Jurídica
```

```
public class ClientesPJuridica extends Clientes{

     private int CNPJ;
     private int inscEstadual;
     private int telComercial;

     public void setCNPJ(int CNPJ) {
               this.CNPJ = CNPJ;

     }

     public int getCNPJ() {
               return CNPJ;
     }

     public void setinscEstadual(int inscEstadual) {
               this.inscEstadual = inscEstadual;

     }

     public int getinscEstadual() {
               return inscEstadual;
     }

     public void settelComercial(int telComercial) {
               this.telComercial = telComercial;
     }

     public int gettelComercial() {
               return telComercial;
     }

     public static void main(String[] args) {
     }

}



//Implementação da Classe Conta Poupança

public class ContaPoupanca extends Conta{

     public void sacar(double valor){
          super.sacar(valor);
          descontarCPMF(valor);
          System.out.println(getSaldo());
     }

     public void transferir(Conta destino, double valor){
          super.transferir(destino,valor);
          descontarCPMF(valor);
     }

     public static void main(String[] args) {
          Conta c = new ContaPoupanca();
          c.setSaldo(100);
```

```
            c.sacar(50);
            System.out.println(c.getSaldo());
      }

}
```