Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

ANDRÉ LUIZ SANTIAGO CAVALEIRO

TEST DRIVEN DEVELOPMENT NO AUXÍLIO DA PROGRAMAÇÃO DIÁRIA

Assis/SP
2017

Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

ANDRÉ LUIZ SANTIAGO CAVALEIRO

TEST DRIVEN DEVELOPMENT NO AUXÍLIO DA PROGRAMAÇÃO DIÁRIA

Projeto de pesquisa apresentado ao curso de Ciência da Computação do Instituto Municipal de Ensino Superior de Assis – IMESA e a Fundação Educacional do Município de Assis – FEMA, como requisito à obtenção do Certificado de Conclusão.

Orientando(a): André Luiz Santiago Cavaleiro
Orientador(a): Prof. Douglas Sanches da Cunha

Assis/SP
2017

FICHA CATALOGRÁFICA

C376t CAVALEIRO, André Luiz Santiago
Test drive development: no auxílio da programação diária /
André Luiz Santiago Cavaleiro . – Assis,2017.

41p.

Trabalho de conclusão do curso (Ciência da Computação). –
Fundação Educacional do Município de Assis-FEMA

Orientador: Me.Douglas Sanches da Cunha

1.Teste 2.JAVA

CDD 005.133

TEST DRIVEN DEVELOPMENT NO AUXÍLIO DA PROGRAMAÇÃO DIÁRIA

ANDRÉ LUIZ SANTIAGO CAVALEIRO

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Graduação, avaliado pela seguinte comissão examinadora:

Orientador: **Prof. Douglas Sanches da Cunha**

Examinador: **Prof. Me. Fábio Eder Cardoso**

Assis/SP
2017

RESUMO

TDD é uma técnica de programação feita para diminuir os erros de projeto, esta técnica de maneira geral não é mostrada nas instituições de ensino, da mesma forma que o TDD não é ensinado, os tipos de testes também não são ensinados o que causa diversos erros na programação. Esses erros cometidos por muitas pessoas fazem com que o cliente perca a confiança na equipe de desenvolvimento, e tenha que apelar para uma outra empresa. Este trabalho tem como objetivo criar um software seguindo a técnica de TDD na qual utiliza a linguagem de programação Java em conjunto com o framework JUnit.

Palavras-chave: TDD**,** Teste, JUnit, Java

ABSTRACT

TDD is a programming technique designed to reduce project errors, this technique is generally not shown in educational institutions, in the same way that TDD is not taught, the types of tests are also not taught, which causes various errors In programming. These mistakes made by many people cause the customer to lose confidence in the development team, and have to appeal to another company. This work aims to create a software following the TDD technique in which it uses the Java programming language in conjunction with the JUnit framework.

Keywords: Test, TDD, JUnit, Java

Lista de ilustrações

SUMÁRIO

1. INTRODUÇÃO

Segundo Aniche (2015), todo desenvolvedor de software já fez um trecho de código que não funcionava e que muitas vezes só foi descoberto depois porque o cliente reportou o erro. Desse modo o programador ter que escrever testes de unidade evita que constrangimentos possam ocorrer no futuro.

O fato de que, para ter um bom sistema, é necessário fazer vários testes não é novidade para ninguém, diversas empresas possuem o próprio time de testes que geralmente fica separado do time de desenvolvimento. Mas o programador realizar a própria rotina de testes é algo que em muitas empresas é incomum, pois para muitos o importante é a finalização de seu trabalho atual, sem importar se o programa está funcionando sem erros ou não. Um desenvolvedor que consegue fazer suas rotinas de testes, tem maior valor no mercado de trabalho que um desenvolvedor que não faz essas rotinas.

TDD (Test Driven Development) tem como conceito de melhorar a lógica de um algoritmo, dessa forma é criado um teste que falhe, depois é feito com que o teste seja aceito e então é refeita a lógica. Esse método consiste em fazer o teste antes de começar o código-fonte. Segundo Aniche (2017) o desenvolver deve escrever testes de maneira constante ao longo do desenvolvimento, mas a técnica de TDD implica em realizar os testes propostos antes de começar a programar.

De acordo com Molinari (2008) os testes além de mostrar um outro ponto de vista dos erros que podem ocorrer, eles podem salvar ou definir a empresa na qual os testes de software estão sendo realizados. Segundo Aniche (2014) o TDD ajuda a equipe a garantir que os requisitos funcionem como o desejado, porém ressalta que o desenvolvedor deve sempre decidir qual o momento certo para ser usado a técnica de TDD.

1.1. OBJETIVO

Este trabalho tem como objetivo tornar a prática de TDD (utilizando a linguagem de programação Java e o framework JUnit) como algo mais fácil e comum para ser utilizado em algoritmos simples e complexos, fazer testes com conexões com o banco de dados e aplicações, testes para alguma rotina especifica que um determinado sistema faz. E verificar o comportamento de rotina de teste ao produzir entradas não esperadas, e assim realizar um estudo sobre o TDD.

1.2. PÚBLICO ALVO

Qualquer profissional na área de TI que tem o interesse de economizar com tempo de testes, e de erros no código, que queira entender o conceito de TDD e como usar o JUnit, que tenha vontade de conhecer mais sobre os testes automatizados, que queira verificar os testes realizados, que busque por uma solução viável(ou complexa) para o problema que o profissional se encontra, ou que queira tentar criar um teste(ou rotina de teste) a partir de um possível erro que encontrou. Pelo fato de que neste trabalho de conclusão de curso será utilizada a linguagem de programação Java, junto com o framework JUnit, então é viável que o interessado tenha um certo conhecimento básico em Java para poder ter um início proveitoso.

1.3. JUSTIFICATIVA

Espera-se que os testes feitos possam além de resolver as questões relacionadas com os erros dos códigos diários, possam também verificar algumas questões no interesse da área da Ciência da Computação como reconhecedores de linguagens e testes que podem ser considerados como Turing Computáveis, além de buscar outros métodos para realizar um mesmo teste, ou criar uma rotina de teste especifica para um projeto futuro.

1.4. MOTIVAÇÃO

Um motivo para realizar este trabalho de conclusão de curso foi a grande quantidade de erros na codificação, estes erros que nunca eram previstos e que muitas vezes eram difíceis de encontrar a solução.

Mas o principal motivo para a realização deste trabalho de conclusão de curso é devido ao interesse na área de testes, e melhorar como programador. Outro motivo é adquirir mais conhecimento na linguagem de programação Java. E por último verificar a veracidade de algumas questões que são estudadas nos cursos de TI.

1.5. PERSPECTIVAS DE CONTRIBUIÇÃO

Este trabalho tem a perspectiva de mostrar os testes automatizados de maneira simples, e mostrar que a técnica de TDD deveria ter sido ensinada nos primeiros anos dos cursos de TI, além disso também tem como fator mostrar algumas funções sobre testes e mostrar como criar uma rotina de testes.

2. O QUE É TDD?

TDD é uma técnica de programação que implica basicamente no uso de testes de unidade para auxiliar no desenvolvimento, de modo que reduza o máximo possível de erros durante a programação e simplificando o código à medida do possível.

Segundo Ambler (2008), TDD (Test Driven Development ou Desenvolvimento Orientado a Testes) é uma abordagem (evolutiva) para o desenvolvimento na qual combina teste e refatoração, é o TFD(Test First Development ou primeiro desenvolvimento de teste) onde você escreve um teste antes de escrever código de produção suficiente para cumprir esse teste e então realizar a refatoração que é a essência do TDD.

De acordo com Ambler (2008), uma forma de entender o TDD é que ele visa a especificação e não a validação, em outras palavras, é uma maneira de pensar através de seus requisitos ou design antes de escrever o seu código funcional (levando em consideração que TDD é importante para requisitos e design do desenvolvimento ágil). Uma outra forma de ver o TDD é como uma técnica de programação, basicamente deixando o código o mais limpo possível.

Segundo Ambler (2008), chama-se TFD a primeira parte do TDD(A primeira parte é o TFD a segunda é refatorar), e realizamos em 4(quatro) passos: O primeiro passo é inserir um teste curto e rápido, o suficiente para que falhe, esse teste deve falhar não importa como, sendo assim muitos programadores decidem por usar uma classe vazia para realizar o teste. Em seguida executa-se seus testes, muitas vezes o conjunto completo de testes, embora, por uma questão de velocidade, pode decidir executar apenas um subconjunto, para garantir que o novo teste de fato falha. O próximo passo é atualizar o código para que seja funcional e consiga passar no novo teste, este teste deve ser seguido de maneira coerente com a lógica pensada. O quarto passo é executar de novo o teste, se o teste falhar então o código deve ser atualizado e retestado, se o teste passar então começa esses quatro passos de novo. Quando tudo tiver sido feito e não consiga mais ser realizado o TDD então deve ser procurado uma nova funcionalidade para uma nova classe (se for orientado a objeto), seguindo o método utilizado no trabalho. Sempre que faz o TFD e refatora o código de teste procurando a melhoria dele, então o que está sendo realizado é o TDD (Figura 1).

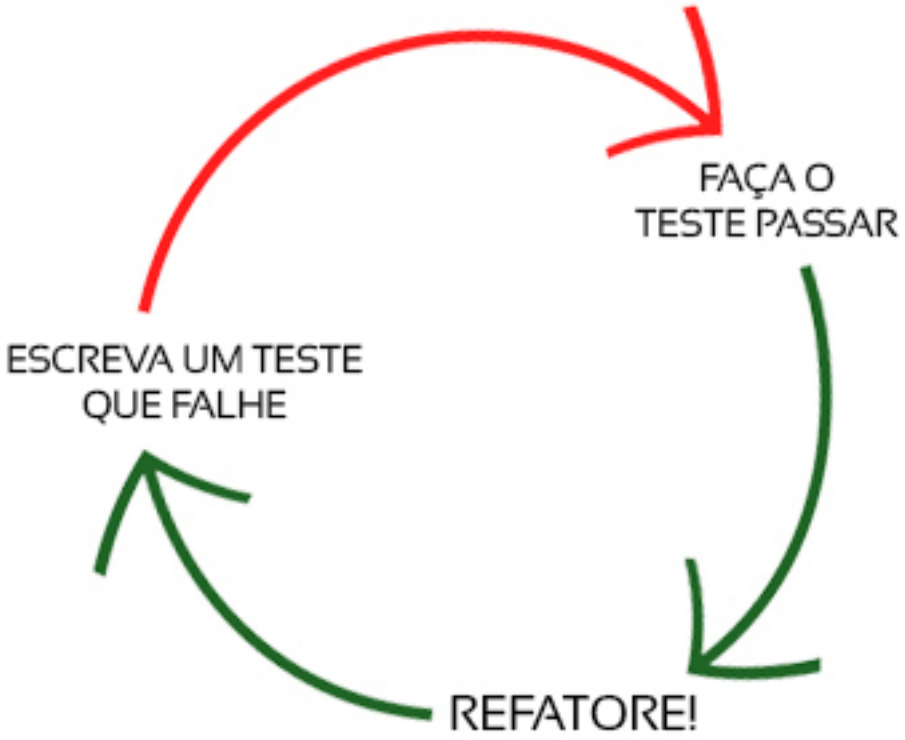

Figura 1: Ciclo do TDD
Fonte: http://tdd.caelum.com.br/

Quando implementar um novo recurso, a primeira pergunta feita é se o design existente é o que melhor se encaixa para implementar essa nova funcionalidade. Em caso afirmativo, deve proceder por meio de uma abordagem TFD. Caso contrário, refatore-o localmente para alterar a parte do design afetada pelo novo recurso, permitindo que você adicione esse recurso o mais fácil possível. Como resultado, o uso da técnica com frequência melhorará a qualidade de seu projeto, tornando mais fácil trabalhar no futuro.

De acordo com Aniche(2010) um dos principais conceitos do TDD é o *Baby Steps* que consiste na ideia de realizar o teste para fazer com que passe da forma mais simples possível, esse conceito basicamente ensina o código para o programador que realizará pedaço por pedaço, essa ideia de auto aprendizagem força o programador a pensar em possibilidades simples para resolver problemas situacionais e mais tarde ir aumentando o código de acordo com a necessidade. O problema é que muitas pessoas acabam por deixar o código mais complexo do que o necessário, e fogem do que realmente seria o TDD.

De acordo com Aniche(2012), em vez de escrever o código funcional primeiro e então o código de teste como é o mais comum a ser feito, o melhor a ser feito(de acordo com o que está sendo estudado) é escrever o código de teste antes do código funcional. Além disso, este código de teste é feito em pequenas etapas (um teste e uma pequena parte de teste funcional em seguida). Um programador que usa TDD deve recusar a ideia de implementar uma nova função até que o teste falhe por causa da falta dessa função. Uma vez que o teste está no lugar, eles fazem o trabalho necessário para garantir que o conjunto de teste passa agora (seu novo código pode quebrar vários testes existentes, bem como o novo). Parece fácil programar usando TDD mas por causa do costume, é muito fácil escrever o código funcional primeiro do que o teste primeiro.

2.1 TIPOS DE TDD E SUA RELAÇÃO

Segundo Ambler (2008), consideramos a existência de dois tipos de TDD que são (Figura 2):

Acceptance TDD (ATDD) – com ATDD é feito um teste de aceitação (ou comportamento de especificação) o suficiente para ser feito a funcionalidade/codigo para realizar o teste. O objetivo do ATDD é a especificação detalhada, e requisitos executáveis para a sua solução em uma base JIT (Just In Time ou hora certa). ATDD também é conhecido como BDD (Behavior Driven Development).

Developer TDD – Nessa abordagem é realizado um teste de desenvolvedor (também conhecido como teste de unidade) o suficiente para realizar o teste. O objetivo do Developer TDD é especificar um projeto (referido também como design) detalhado e executável para sua solução em uma base JIT. Developer TDD é conhecido como TDD.

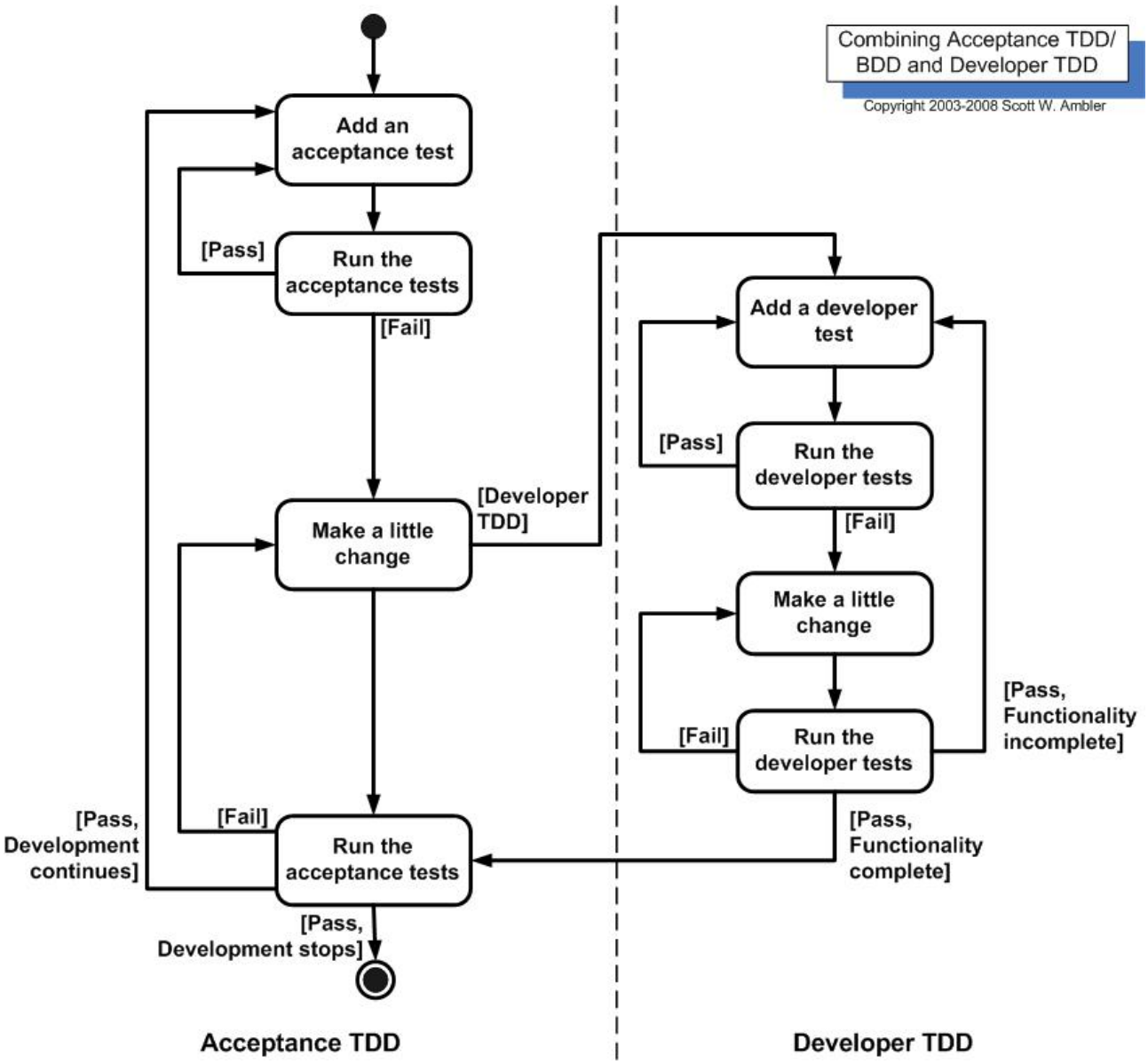

Figura 2:ATDD e TDD
Fonte: http://agiledata.org/essays/tdd.html

Tanto o TDD como o ATDD podem ser realizados de forma dependente ou independente uma da outra, depende de como a equipe de desenvolvimento de software está realizando o as etapas de produção de software.

Como não é realizada a validação de requisitos para o software então não é utilizado o ATDD, somente o TDD neste trabalho.

3. TESTES

Neste capítulo são dados os conceitos de teste, junto com uma explicação do framework JUnit e seu funcionamento com o TDD.

Existem vários tipos de testes de acordo com cada etapa do projeto de software, como ilustrado na figura abaixo (Figura 3), os testes dessas etapas podem ocorrer de forma independente ou sequencial.

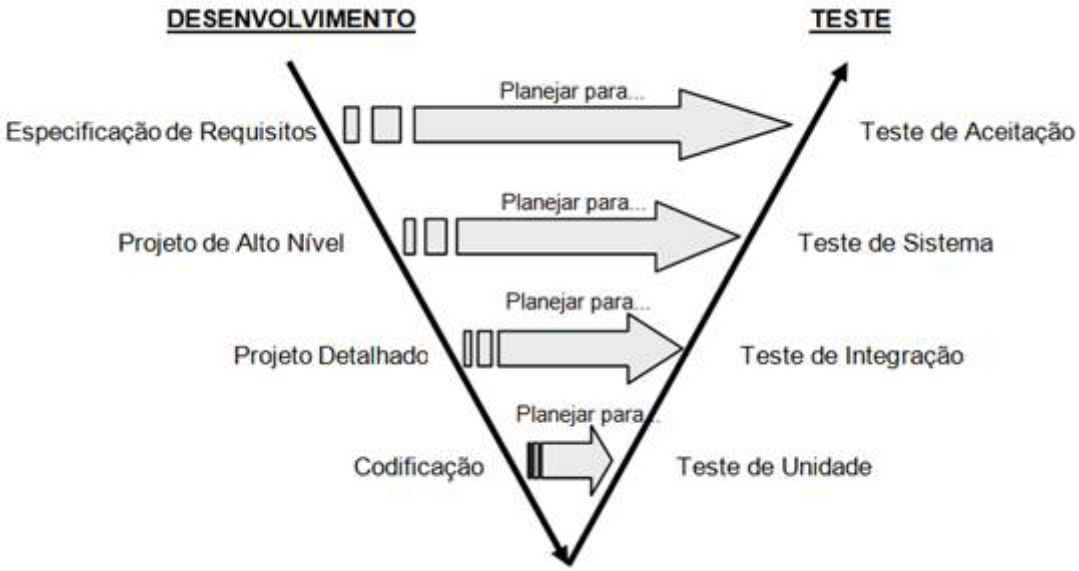

Figura 3: Desenvolvimento e Teste

Fonte: http://www.devmedia.com.br/imagens/engsoft/artigo7/image03.jpg

- Teste de Aceitação: é o teste que envolve a interação do usuário com o sistema, é o teste que mostra o quão bem-aceito o sistema é pelo publico, o teste determina se uma funcionalidade do sistema é aceita ou não. Tem por função verificar o sistema em relação aos seus requisitos;
- Teste de Sistema: é o teste que garante que o sistema funcione por completo, ele testa as funcionalidades do sistema como um todo, testa requisitos funcionais e não funcionais (como a expectativa do cliente);
- Teste de Integração: é aquele que testa a integração entre duas partes do seu sistema, ou seja, é o teste de um sistema interno com um sistema externo (seu

sistema para o banco de dados por exemplo). O teste de integração testa os métodos programados em conjunto;

- Teste de Unidade: É aquela que testa uma única unidade do sistema, testa a dependência que a unidade possui, ou seja, validação de dados, sejam esses dados validos ou não.

Pelo fato de utilizar o TDD então o que é dado mais importância neste trabalho é o teste de unidade. O teste de unidade pode testar qualquer tipo de funcionalidade do sistema, porém ele testa uma pequena quantidade desse sistema, pode ser testado desde variáveis simples até classes (se for orientado a objeto), geralmente sendo usado quando o programador está trabalhando no código do sistema.

3.1 TESTE DE UNIDADE COM JUNIT

Como que foi utilizado Java com o framework JUnit neste trabalho, então os exemplos de testes realizados (e aqui mostrados) no formato da linguagem Java. O JUnit testa o contexto, verificando se é ou não o que se espera, abaixo temos como o JUnit funciona (figura 4).

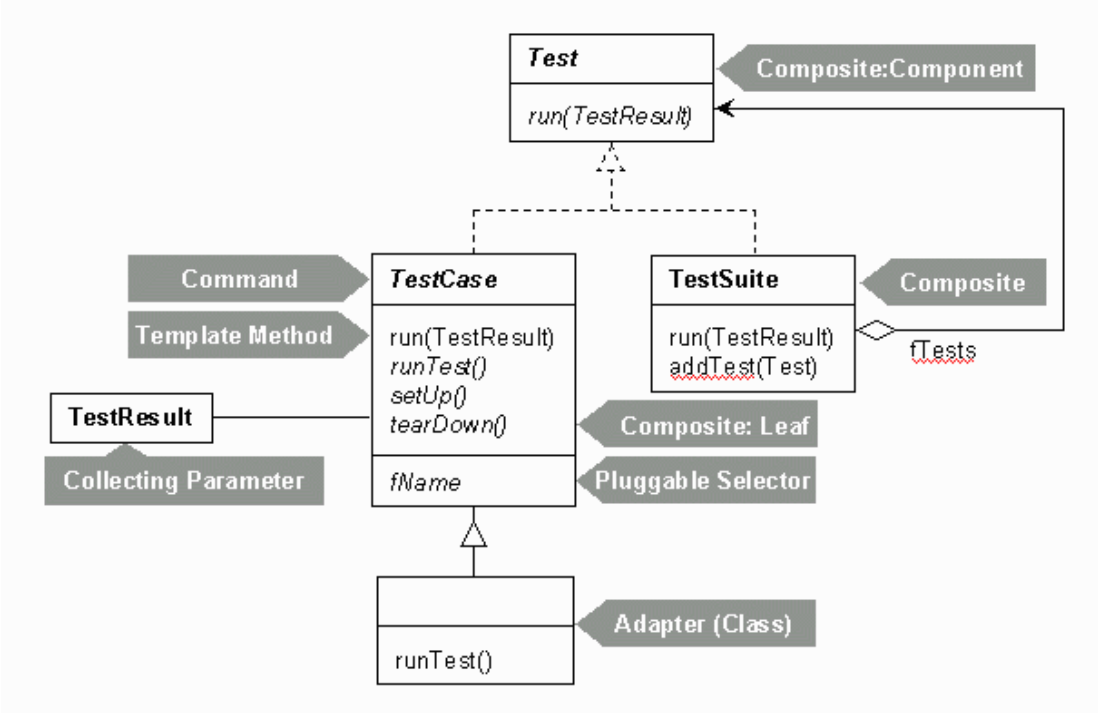

Figura 4: Funcionamento do JUnit

Fonte: http://junit.sourceforge.net/doc/cookstour/Image6.gif

O JUnit tem algumas anotações (annotations) que indicam o que cada método faz, annotations são metadados que realizam tarefas pré-definidas sem a necessidade de um xml, é possível criar annotations, e as annotations principais do JUnit são:

- @Test: Identifica que o método é um teste;
- @Before: Indica que é um método que vem antes do teste, normalmente usado para preparar o ambiente;
- @After: Indica que é um método que vem depois do teste, usado para “limpar” o ambiente;
- @BeforeClass: É o método que é executado antes de executar a rotina, ou seja, antes do @Before;
- @AfterClass: É o método que é executado depois de finalizar a rotina, depois do @After;
- @Ignore: É o método que não permite que o teste seja executado, utilizado quando não quer que o método seja testado.

Logo abaixo (figura 5) temos uma classe como exemplo do uso do @Test, vale ressaltar que possui mais annotations durante os testes mas para começar o teste tem que usar a @Test.

Página Inicial | Calculos.java | TesteCalculos.java

Código-Fonte | Histórico

```
9  import junit.framework.Assert;
10 import org.junit.Test;
11
12 /**
13  *
14  * @author André
15  */
16 public class TesteCalculos {
17     @Test
18     public void test(){
19         Calculos c = new Calculos();
20         Assert.assertEquals(4.0, c.somar(2, 2));
21         Assert.assertEquals(6.0, c.multiplicar(2, 3));
22         Assert.assertEquals(2.0, c.dividir(10, 5));
23         Assert.assertEquals(0, c.subtrair(8, 8));
24     }
25
26     @Test
27     public void test2(){
28         Calculos c = new Calculos();
29         Assert.assertEquals(true, c.somar(2.0, 2.0) == 4);
30     }
31     @Test
32     public void test3(){
33         //falhando
34         Calculos c = new Calculos();
35         Assert.assertEquals(false, c.somar(2.0, 2.0) == 4);
36         Assert.assertEquals(10, c.subtrair(2, 5));
37     }
38 }
```

OutroTest.TesteCalculos > test2

Figura 5: Classe de Teste

O JUnit possui métodos que realizam a validação do teste, esses métodos realizam o teste mostrando assim o resultado, testam de acordo com o que o programador colocar para testar, e eles são os Asserts:

- fail(String): Faz o método falhar, usado para verificar se uma determinada parte do código está sendo acessada;
- assertTrue(true)/(false): Sempre *True* ou *False*, pode se usar para predefinir o resultado de um teste ainda não implementado;
- assertEquals(mensagem, esperado, real): Testa dois valores, verifica se são iguais;
- assertEquals(mensagem, esperado, tolerância, real): Testa valores reais, sendo que a tolerância é o número de casas;
- assertNull(mensagem, objeto): Verifica se o objeto é nulo;
- assertNotNull(mensagem, objeto): Verifica se não é nulo;
- assertSame(String, esperado, real): Verifica se as duas variáveis se referem ao mesmo objeto;
- assertNotSame(String, esperado, real): Verifica se as duas variáveis não se referem ao mesmo objeto.

Os métodos do JUnit são bem parecidos entre si e se comportam de forma similar, essa forma é realizar o teste para conferir se o resultado é igual ao esperado.

Seguindo a imagem de cima (figura 5) temos a classe que é testada (figura 6), devido ao fato de que para fazer um teste preciso precisamos de uma classe específica:

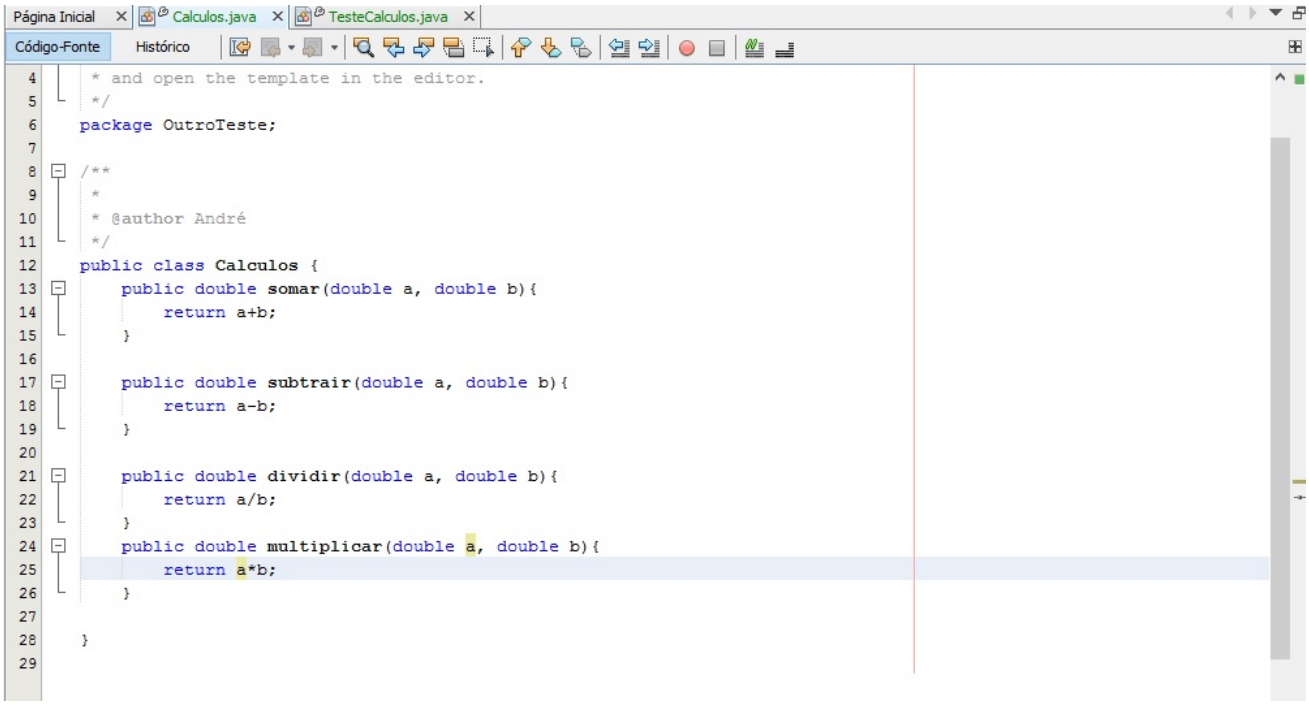

Figura 6: Classe a ser testada

Na imagem abaixo (figura 7) temos o resultado dos testes de forma bastante completa mostrando erros, falhas, testes que foram executados, testes que foram pulados e o tempo de execução. É possível notar que o JUnit mostra qual a linha de teste que falhou e o motivo da falha.

Figura 7: Saída do Teste

Vale ressaltar que o código falhar no teste é algo positivo, já que foi encontrado um erro e essa funcionalidade pode ser arrumada antes de ser entregue ao cliente, e claro pode ocorrer de ser encontrado outros erros no código. O esperar que o código falhe entra no contexto de TDD que é começar o teste com uma falha.

3.2 TDD COM JUNIT

Um projeto utilizando o conceito de TDD deve começar sempre vazio, por ser utilizado o NetBeans (neste trabalho de conclusão de curso) então é criado um pacote de códigos separado especificamente para os testes. Segundo o website do Netbeans (2017), o Netbeans já vem com o JUnit no seu instalador por esse motivo que é utilizado neste trabalho.

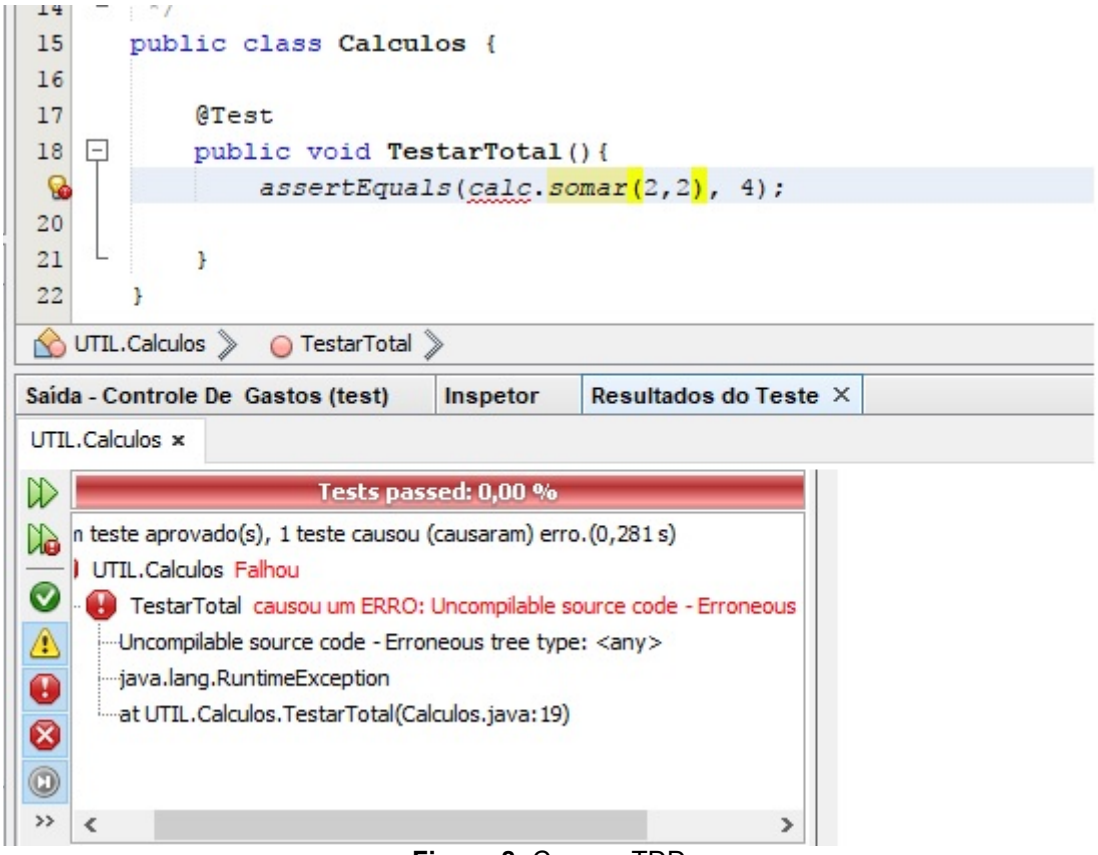

Figura 8: ComeçoTDD

Depois deve ser feito um teste falhando(figura 8), se for seguir os *baby steps* então deve ser feita uma classe com o retorno do resultado(figura 9) que foi testado, fazendo assim com que o teste passe.

```
12  public class Calcular {
13      public int somar (int a, int b){
14          return 4;
15      }
16  }
17
```

Figura 9: BabySteps

É notável que mesmo o teste ter passado (figura 10) ele ainda não deve ser considerado como último teste, já que o resultado é uma parte do processo do TDD, o processo do TDD deve ser finalizado quando o problema foi inteiro solucionado ou solucionado de forma que o resultado não interfira no software (depende do que o programador decidir).

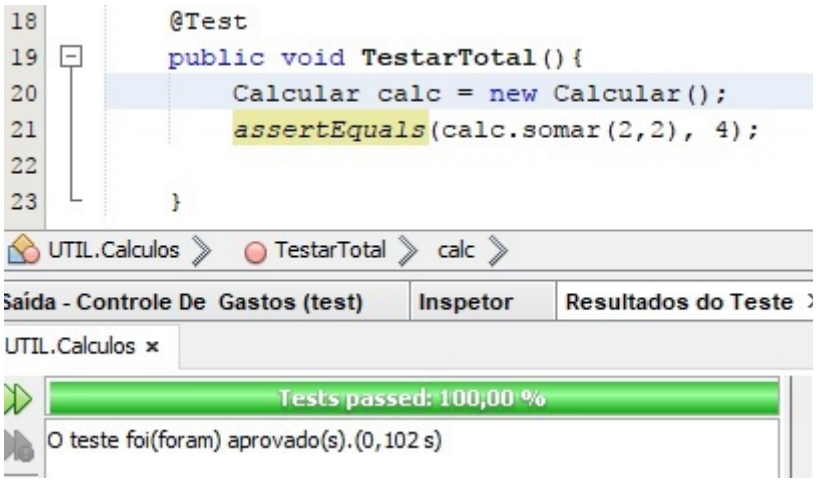

Figura 10: Primeira passagem

O TDD seguindo os *Baby steps,* deve seguir essa ordem, de forma praticamente obvia que o que foi feito está errado, o próximo passo é fazer um outro teste seguindo o padrão e ver se ele passa (figura 11).

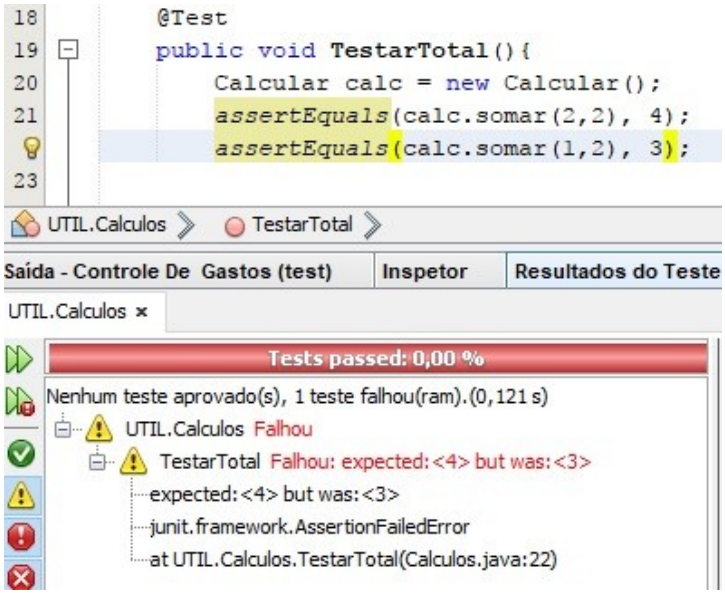

Figura 11: Testando novamente

O próximo passo é pensar numa forma simples de resolver o problema, que não gere o resultado anterior. O TDD sempre deve ser feito de forma simples e objetiva(figura 12).

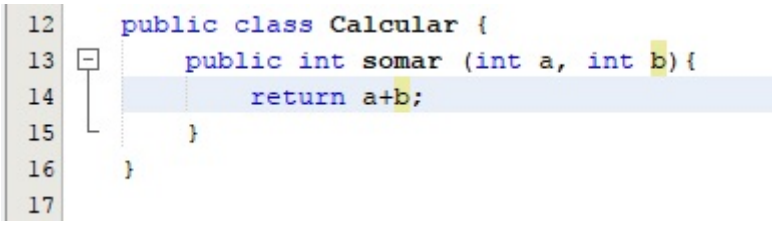

Figura 12: SoluçãoSimples

O TDD termina com o teste sendo finalizado com um resultado positivo e quando não há mais como refatorar (Figura 13).

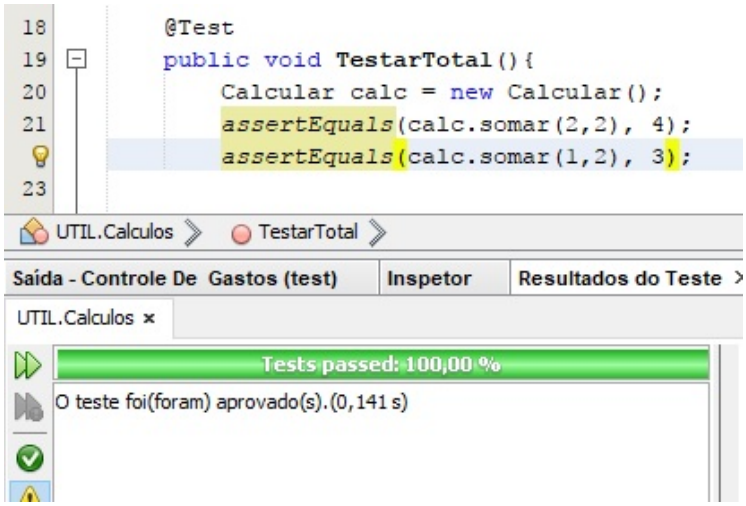

Figura 13: TDDFim

Os *baby steps* não precisam ser seguidos com precisão já que no TDD também tem a opção de pular algumas etapas como um processo de aprendizagem, ou seja, poderia pular o “return 4” (figura 9) para o “return a+b” (figura12), supostamente pulando algumas etapas podem ocorrer erros inesperados caso o programador não tenha total conhecimento do que precisava ser feito.

4. O PROGRAMA

Neste capítulo são mostrados os diagramas do software e alguns testes utilizados.

O principal objetivo deste trabalho não é o software (programa), mas sim o estudo e as técnicas que o TDD pode nos proporcionar.

Foi feito um programa de gerência de gastos pessoais, baseado no estilo de livro caixa. O programa tem cinco telas, sendo que uma dessas telas é a consulta dos gastos juntos com o valor total, nas outras três um CRUD (Create, Read/Retrive, Update e Delete ou Criar, Consultar, Atualizar e Remover) cada, e a última tela é o menu que na ordem de execução sempre é a primeira tela a ser exibida quando o programa é executado. Foi utilizado o banco de dados MySQL.

4.1 ANÁLISE DO SISTEMA

4.1.1 Diagrama caso de uso geral

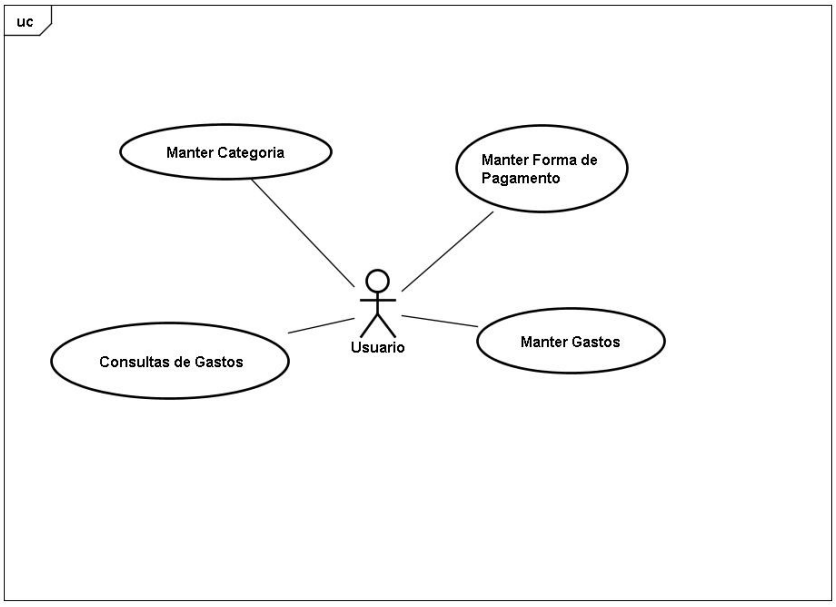

Figura 14: CasoGlobal

4.1.2 Diagrama de entidade e Relacionamento (DER)

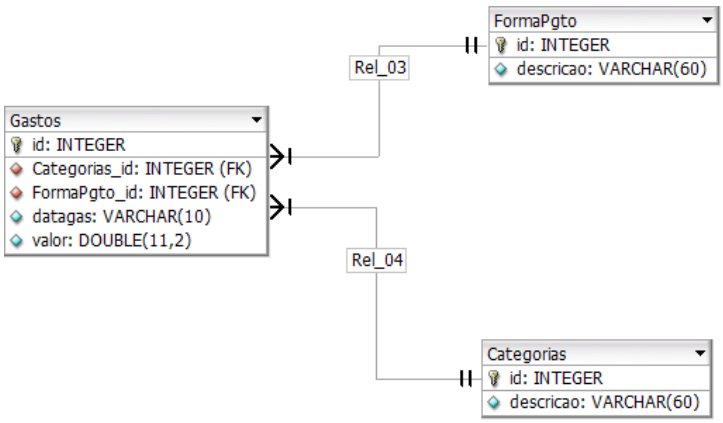

Figura 15: DER

4.1.3 Diagrama de classe

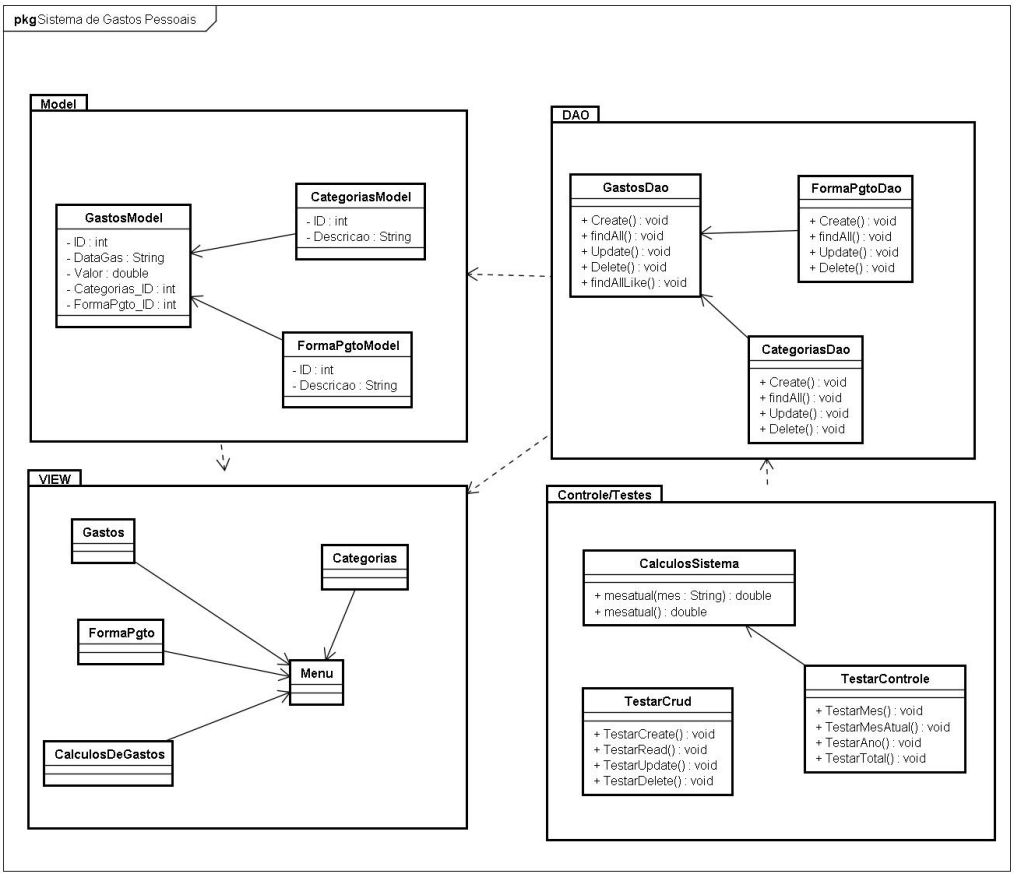

Figura 16: Diagrama de classe

4.1.4 Especificação dos casos de uso

4.1.4.1 Manter Categoria

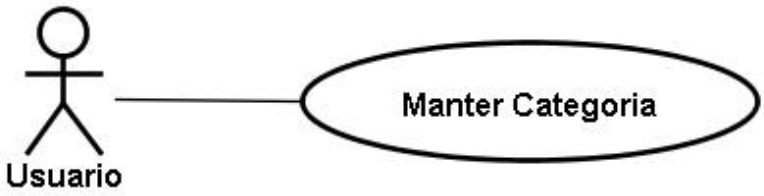

Figura 17: CasoManterCategoria

Finalidade/Objetivo:

Permitir que o usuário gerencie suas categorias

Atores:

Usuário

Evento Inicial:

É mostrado as categorias que o sistema possui.

Fluxo Principal:

O usuário deve acessar a área de gerenciamento de categorias;

Selecionar a opção;

Deve preencher os dados;

Selecionar o botão confirmar e é realizado o gerenciamento.

Fluxos Alternativos:

Pode cancelar o cadastro.

Os dados da categoria podem ser alterados pelo usuário.

O usuário pode excluir a categoria.

Fluxos de Exceção:

Se o usuário informar os dados incorretos a categoria não é cadastrada ou alterada, e é preciso fazer as correções.

Pós-Condições:

O usuário poderá gerenciar os itens que necessita.

Casos de Testes:

Validar os dados que vão ser cadastrados e alterados.

4.1.4.2 Manter Forma de pagamento

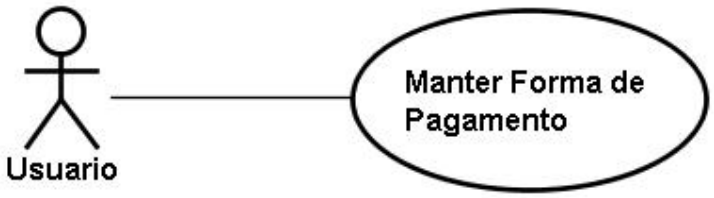

Figura 18: CasoManterForma

Finalidade/Objetivo:

Permitir que o usuário gerencie suas formas de pagamento

Atores:

Usuário

Evento Inicial:

É mostrado as forma de pagamento que o sistema possui.

Fluxo Principal:

O usuário deve acessar a área de gerenciamento de forma de pagamento;

Selecionar a opção;

Deve preencher os dados;

Selecionar o botão confirmar e é realizado o gerenciamento.

Fluxos Alternativos:

Pode cancelar o cadastro.

Os dados da forma de pagamento podem ser alterados pelo usuário.

O usuário pode excluir a forma de pagamento.

Fluxos de Exceção:

Se o usuário informar os dados incorretos a forma de pagamento não é cadastrada ou alterada, e é preciso fazer as correções.

Pós-Condições:

O usuário poderá gerenciar os itens que necessita.

Casos de Testes:

Validar os dados que vão ser cadastrados e alterados.

4.1.4.3 Manter Gastos

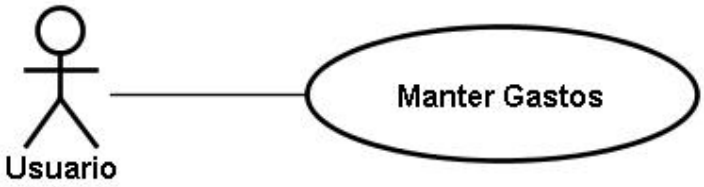

Figura 19: CasoManterGastos

Finalidade/Objetivo:

Permitir que o usuário gerencie seus gastos

Atores:

Usuário

Evento Inicial:

É mostrado os gastos que o sistema possui.

Fluxo Principal:

O usuário deve acessar a área de gerenciamento de gastos;

Selecionar a opção;

Deve preencher os dados;

Selecionar o botão confirmar e é realizado o gerenciamento.

Fluxos Alternativos:

Pode cancelar o cadastro.

Os dados dos gastos podem ser alterados pelo usuário.

O usuário pode excluir o gasto.

Fluxos de Exceção:

Se o usuário informar os dados incorretos o gasto não é cadastrado ou alterado, e é preciso fazer as correções.

Pós-Condições:

O usuário poderá gerenciar os itens que necessita.

Casos de Testes:

Validar os dados que vão ser cadastrados e alterados.

4.1.4.4 Consultar Gastos

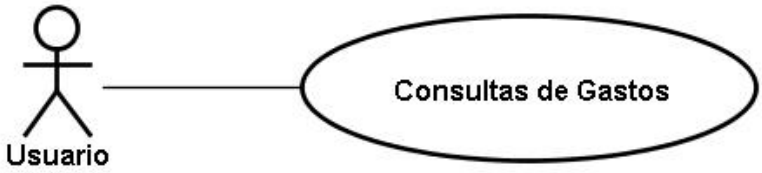

Figura 20: CasoCGastos

Finalidade/Objetivo:

Permitir que o usuário consulte os seus gastos

Atores:

Usuário

Evento Inicial:

É mostrado o valor dos gastos específicos que o sistema possui.

Fluxo Principal:

O usuário deve acessar a área de cálculos de gastos.

Selecionar a opção;

Deve preencher os dados;

Selecionar o botão buscar e é realizado a busca.

Fluxos Alternativos:

Pode ver o gasto total e do mês atual.

Fluxos de Exceção:

Se o usuário informar os dados incorretos a busca não é realizada.

Pós-Condições:

O usuário poderá gerenciar os itens que necessitam dessa categoria.

Casos de Testes:

Validar os dados que vão ser consultados e calculados.

4.1.5 Diagrama de sequência

4.1.5.1 Diagrama de sequência manter categoria

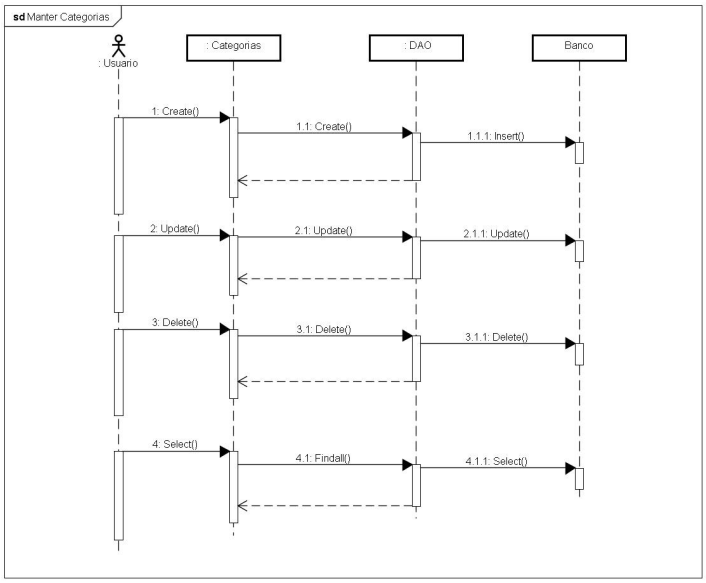

Figura 21: DiaSeqCat

4.1.5.2 Diagrama de sequência manter forma de pagamento

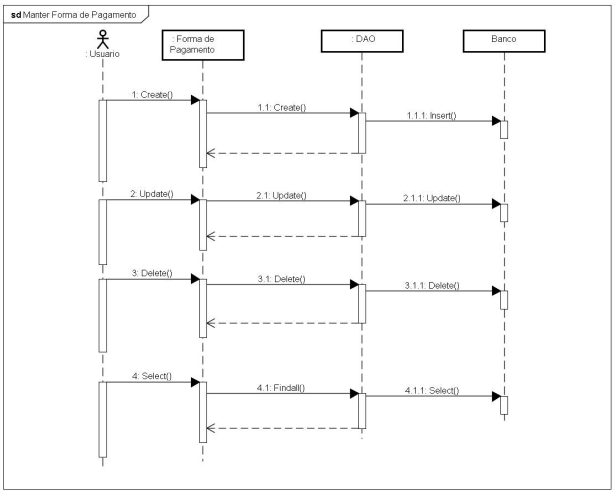

Figura 22: DiaSeqFor

4.1.5.3 Diagrama de sequência manter gastos

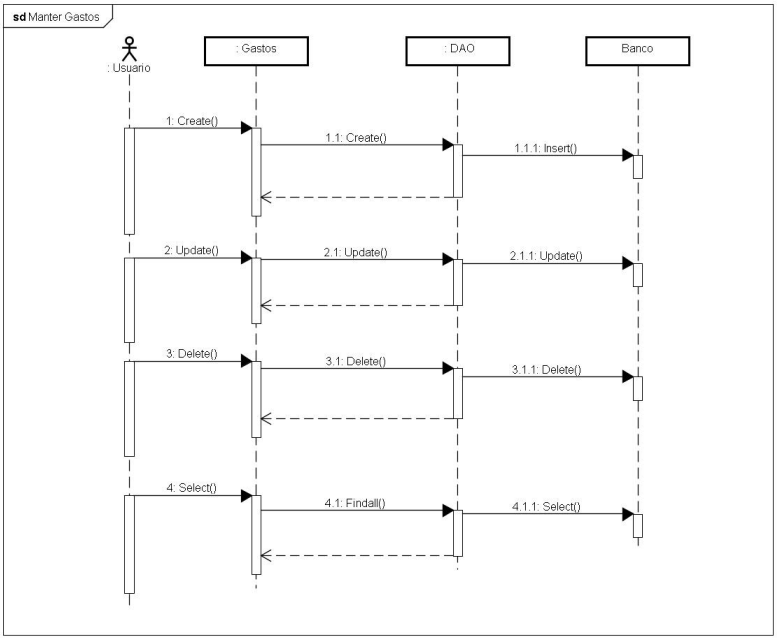

Figura 23: DiaSeqGast

4.1.5.4 Diagrama de sequência consultar gastos

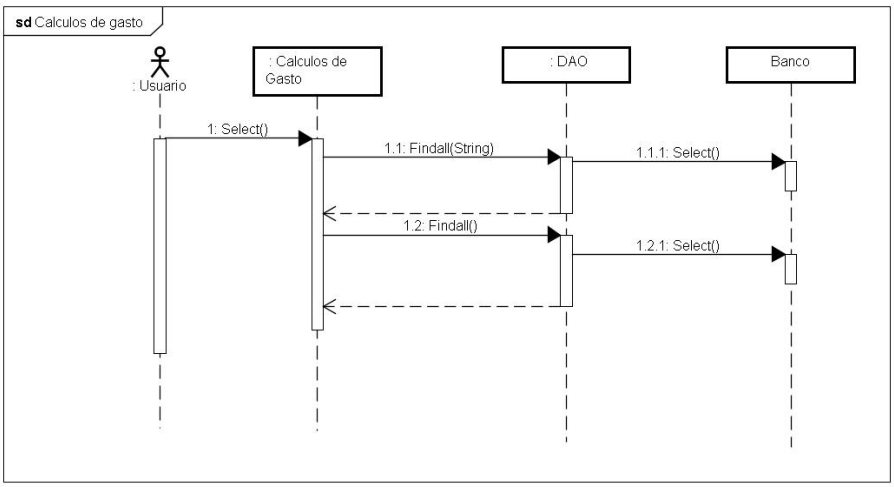

Figura 24: DiaSeqCG

4.2 TESTES DO SISTEMA

O TDD não pode ser utilizado na inteira criação do software sendo que para cada etapa é necessário um tipo de teste diferente. O conceito de *baby steps* não foi o foco neste projeto, sendo que foi utilizado como material de estudo.

4.2.1 TDD no projeto

```
18
19      @Test
20      public void TestarTotal(){
21
22      }
23      @Test
24      public void TestarAno(){
25
26      }
27      @Test
28      public void TestarMes(){
29
30      }
31      @Test
32      public void TestarMesAtual(){
33
34      }
```

Figura 25: ListaTestes

Primeira coisa que foi feita nos testes utilizando o conceito de TDD é listar o que deve ser feito (mesmo que não seja realizado tudo) para ter uma base do que deve fazer (figura 25).

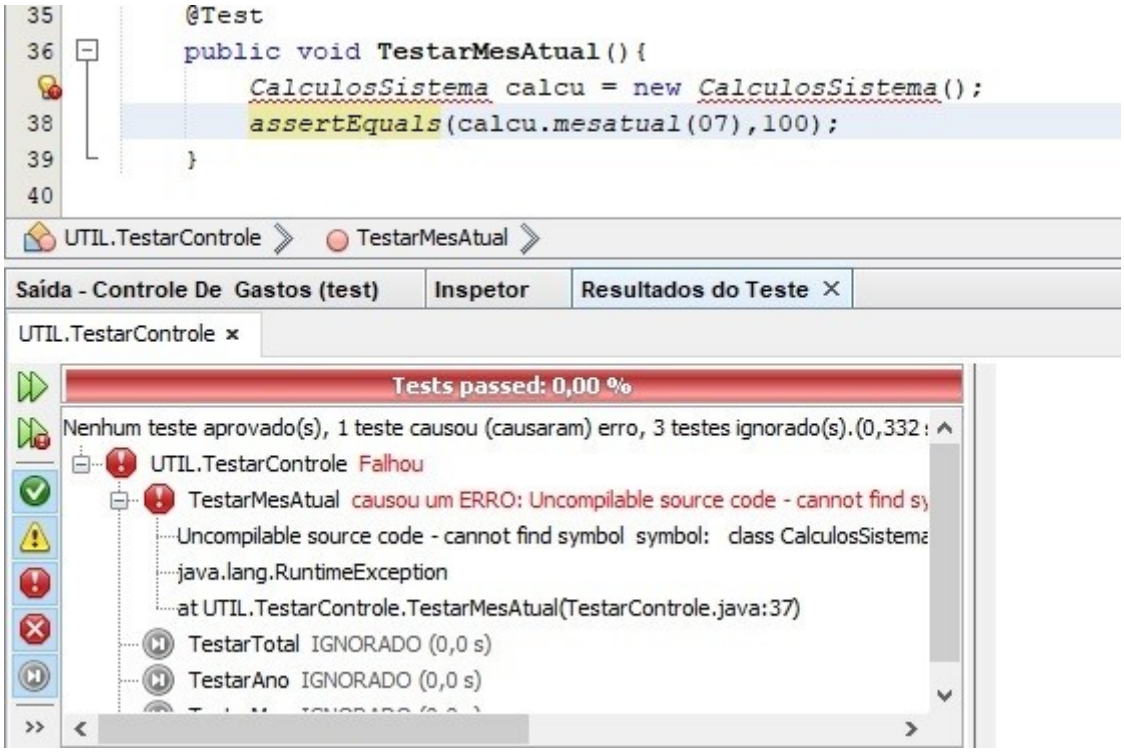

Figura 26: TDDSist

O segundo passo foi realizar um teste falhando seguindo assim os passos do TDD (figura 26), o TDD foi seguido utilizando o conceito de *baby steps* então a classe até então não existia.

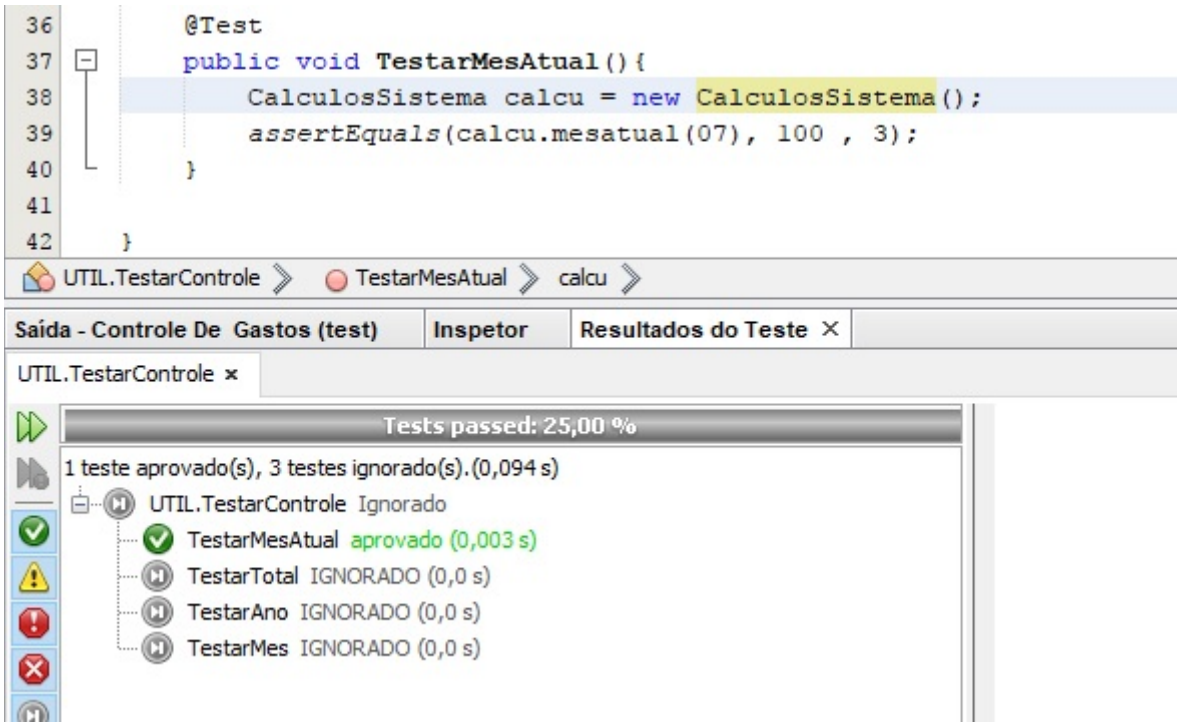

Figura 27: CalculosSist

O Teste foi moldado para o modelo de ponto flutuante (“[...](07), 100, 3”) para poder retornar a soma do mês (figura 27), no instante deste teste a classe testada já existia e foi moldada para seguir os conceitos dos *baby steps*.

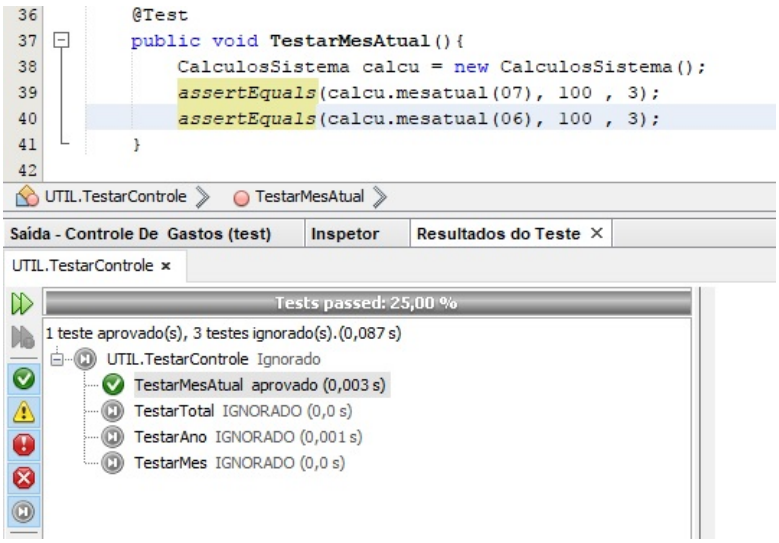

Figura 28: FalsoAfirmativo

Algumas vezes o teste pode passar mesmo precisando que falhe (figura 28), neste momento o programador deve realizar mais um teste para que ele falhe, note que mesmo passando um valor diferente ele retornou o mesmo resultado positivo, com isso em mente deve ser feito um teste mudando o resultado.

Foi utilizado o TDD para finalizar o programa (figura 29), alguns passos do TDD não foram registrados pois a leitura ficaria muito cansativa e repetitiva. Como o TDD tem o conceito de deixar a classe o mais simples possível então é notável que a classe ficou bem simples

```
15  public class CalculosSistema {
16      public double mesatual(String mes){
17
18          GastosDao gastd = new GastosDao();
19          double val=0;
20
21          for(GastosModel gastm : gastd.findAllLike(mes)){
22              val = val + gastm.getValor();
23          }
24
25
26          return val;
27      }
```

Figura 29: CalculosSistemaFeito

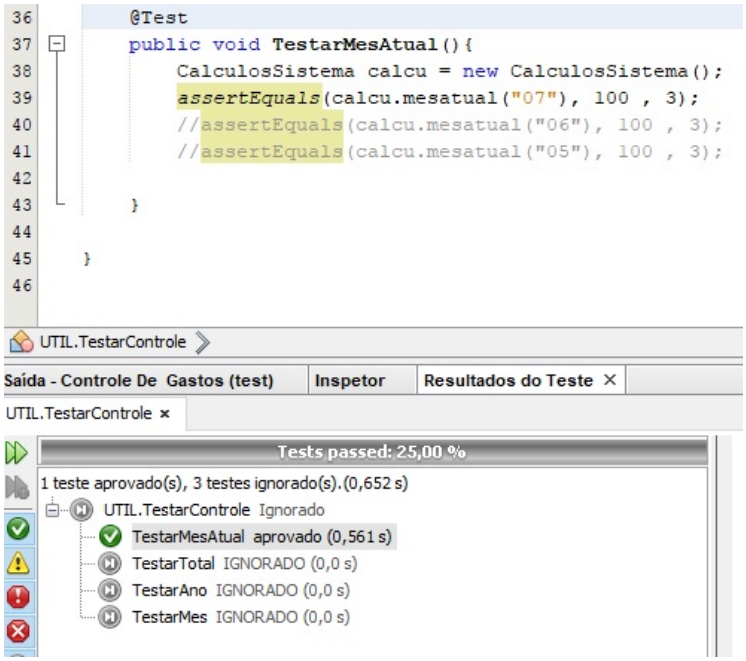

Figura 30: TesteCompleto

Foram feitas algumas mudanças no teste para que ele possa se adequar aos padrões realizados no projeto (figura 30)

Mesmo o registro do TDD do projeto neste trabalho ter terminado aquilo, vale ressaltar que o mesmo foi continuado até completar os requisitos necessários para o sistema.

4.2.2 Teste de Crud

O Teste de crud (figura 31 e 32) foi realizado com base no pacote DAO do projeto. Se reparar bem na figura 31 é notável que as classes model e dao são instanciadas e o assert do teste só é feito quando há alguma falha na String sql ou se não conecta no banco de dados.

```
18  public class TestarCrud {
19      //utilizar a annotation @Test para realizaar o teste
20      @Test
21      //utilizar a annotation @Ignore para não realizar o Teste
22      @Ignore
23      public void TestarCreate(){
24          //criar um objeto puxando o model
25          CategoriasModel ca = new CategoriasModel();
26          //criar um objeto puxando do dao
27          CategoriasDao cao = new CategoriasDao();
28
29          if(cao.Create(ca)){
30              System.out.println("Foi inserido com sucesso");
31          }else fail("Erro ao inserir");
32      }
33      @Test
34      @Ignore
35      public void TestarRead(){
36          CategoriasDao cao = new CategoriasDao();
37
            for(CategoriasModel ca: cao.findAll()){
39              System.out.println("Descrição: "+ ca.getDescricao());
40          }
41
42      }
```

Figura 31: TesteCrudP1

Na imagem abaixo (figura 32) percebe-se que os testes são bem parecidos entre si, a diferença é a chamada do método.

```
43      @Test
44      @Ignore
45      public void TestarUpdate(){
46          CategoriasModel ca = new CategoriasModel("Comida");
47          ca.setId(1);
48          CategoriasDao cao = new CategoriasDao();
49
50          if(cao.Update(ca)){
51              System.out.println("Foi atualizado com sucesso");
52          }else fail("Erro ao atualizar");
53      }
54      @Test
55      @Ignore
56      public void TestarDelete(){
57          CategoriasModel ca = new CategoriasModel();
58          ca.setId(1);
59          CategoriasDao cao = new CategoriasDao();
60
61          if(cao.Delete(ca)){
62              System.out.println("Foi deletado com sucesso");
63          }else fail("Erro ao deletar");
64      }
65
66  }
```

Figura 32: TesteCrudP2

Foi utilizada a annotation @Ignore (figura 31 e 32) para não realizar o teste (update por exemplo) quando o outro estava sendo testado (delete por exemplo).

5. CONCLUSÃO

Os testes realizados utilizando o conceito de TDD são a princípio confusos, mas quando começa a ser executado o método do TDD, mostra que é de fácil utilização. TDD é uma técnica de programação que serve para diminuir os erros envolvendo o projeto, o tipo de teste utilizado é o teste de unidade, no software que foi desenvolvido utilizando essas técnicas foi notado que o TDD é utilizado em questões específicas, não se utiliza o TDD para fazer conexão com banco de dados ou para instanciar uma Classe padrão.

O TDD deveria ser utilizado para ensinar programação (utilizando os *baby steps*), poderia ser muito útil se for passado para os primeiros anos dos cursos de computação, já que ele obriga o programador a aprender.

Enfim o TDD é ótimo, quando for utilizado em grandes empresas o ideal é não utilizar os *baby steps*, pois poderia atrasar algo que não precisaria de tanto tempo. Sempre é importante ter o Software testado, então se tratando de testes e diminuição na quantidade de erros durante a programação a técnica de TDD se mostra extremamente útil e eficiente, porém o programador deve sempre levar em conta que a técnica não consegue resolver todos os problemas que podem surgir.

5.1 TRABALHOS FUTUROS

Apesar de os esforços ao fazer o trabalho de conclusão de curso, alguns pontos não foram estudados mais a fundo como: outras possibilidades com o JUnit (criar próprias annotations), maior profundidade em relação aos tipos de teste, maior foco no BDD (ATDD) com o TDD, outras técnicas que possam substituir o TDD, e outras ferramentas para teste. Essas questões não esclarecidas dariam boas pesquisas para o futuro.

6. REFERÊNCIAS

Ambler, Scott W. , **Introduction to Test Driven Development (TDD),** Disponível em: <http://agiledata.org/essays/tdd.html> Acesso em: 26 out. 2016.

Aniche, Maurício. **Test Driven Development Teste e Design no Mundo Real.** São Paulo: Casa do Código, 2012.

Aniche, Maurício. **Testes automatizados de software: guia pratico.** São Paulo: Casa do Código, 2015.

Aniche, Maurício, **Cuidado com seus baby steps!,** Disponivel em: <http://www.mauricioaniche.com/2010/11/cuidado-com-seus-baby-steps/> Acesso em: 28 jun. 2017

Aniche, Maurício, **TDD,** Disponivel em: <http://tdd.caelum.com.br/> Acesso em: 10 out. 2016.

Aniche, Maurício, **Unidade, integração ou sistema? Qual teste fazer?,** Disponivel em: <http://blog.caelum.com.br/unidade-integracao-ou-sistema-qual-teste-fazer/> Acesso em: 25 fev. 2017.

Cardoso, André, **TDD, Por que usar?,** Disponível em: <http://tableless.com.br/tdd-por-que-usar/> Acesso em: 25 out. 2016.

Gomes, Fabio, **Introdução ao desenvolvimento guiado por teste(TDD) com JUnit**, Disponivel em: <http://www.devmedia.com.br/introducao-ao-desenvolvimento-guiado-por-teste-tdd-com-junit/26559> Acesso em: 09 mar. 2017.

JUnit. **JUnit.** Disponivel em: <http://junit.org/junit4/> Acesso em: 18 jan. 2017

Molinari, Leonardo. **Testes Funcionais de Software.** Florianópolis: Visual Books, 2008.

Oracle. **NetBeans.** Disponivel em: <https://netbeans.org/> Acesso em: 09 mar. 2017.