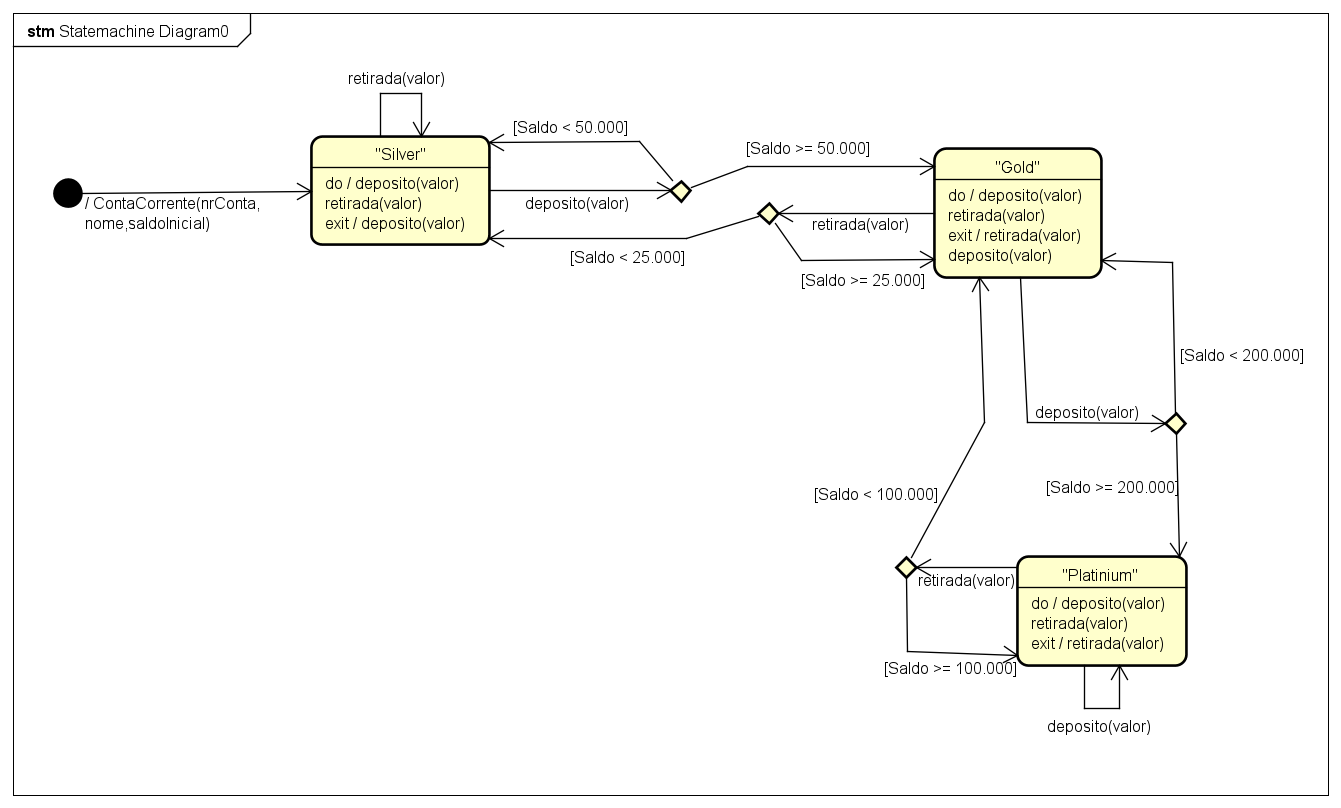
**Tarefa Geração Casos de Teste**

**Bruno Simm Alves**

**Turma 168 – Sistemas de Informação**

1. **Diagrama de Estados**

O primeiro passo para desenvolver a solução do projeto foi a geração do Diagrama de Estados, que resultou no diagrama apresentado abaixo:   


Este diagrama é dividido em **três estados básicos** (nós), Silver, Gold e Platinium. O estado “Silver” pode ser alcançado sempre que uma nova conta é criada, enquanto o saldo for inferior a 50.000 (aresta) ou quando o saldo de uma conta Gold se torna menor que 25.000 (aresta). Já o estado “Gold” pode ser alcançado de duas formas, a primeira é através do “upgrade” de uma conta “Silver” mediante depósitos que façam o saldo desta conta ser maior ou igual a 50.000 (aresta), a segunda é através de “downgrade” de uma conta “Platinium” mediante retiradas que façam o saldo desta conta ser menor que 100.000 (aresta).

O terceiro estado deste diagrama, o “Platinium”, pode também ser alcançado através do “upgrade” de uma conta “Gold” mediante depósitos que façam o saldo desta conta ser maior ou igual a 200.000 (aresta). Conforme citado anteriormente, uma conta “Platinium” só irá sofrer “downgrade” quando seu saldo for menor que 100.000 (aresta).

A cobertura dos testes dos Nós e Arestas do diagrama de estados será implementado através do exercício dos seguintes casos e dos próximos casos de teste que se darão através de outras estratégias.

* 1. **Testes de upgrade através de depósito:**

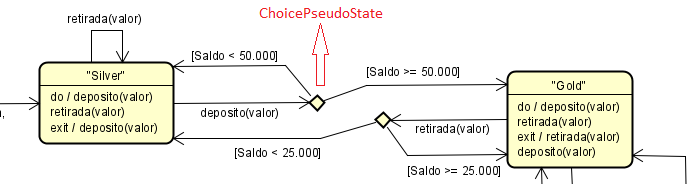
**1.1.1 Teste do Nó “Silver” visando upgrade via depósito:**

Teste para exercício da funcionalidade de upgrade para “Gold” através de depósitos em uma conta “Silver”. Este teste irá exercitar o caminho de criação de uma nova conta e então o depósito de um valor obtido através do “CsvSource”.

1. @ParameterizedTest
2. @CsvSource({
3. "1000,Silver", // in-point
4. "300000,Gold", // out-point
5. "50000,Gold",//Limite on-point
6. "49999,Silver"// Limite off-point
7. })
8. public *void* upgradeViaDeposito\_ContaSilver(*double* *valorDeposito*, String *tipoContaEsperado*){
9. //Teste de partição (C1) + Teste de limite para os upgrades da conta silver (49.99 e 50.00)
10. ContaCorrente c = **new** ContaCorrente("1", "Teste upgrade conta Silver");
11. c.deposito(*valorDeposito*);
12. assertEquals(*tipoContaEsperado*, c.getCategoria());
13. }

Utilizado como exemplo a entrada ”50.000,Gold” do teste parametrizado, que é o limite on-point (estratégia de limites) deste caso, o “**ChoicePseudoState**” utilizado no diagrama de estados, logo após a execução do método “deposito (valor) ” irá seguir o fluxo através do “Guard” [Saldo >= 50.000], alterando o estado da propriedade “categoria” para “Gold”.

A imagem que segue expõe o ChoicePseudoState e seus Guards: [Saldo < 50.000] e [Saldo >= 50.000]. Cada Guard irá definir o estado a ser adotado após o método deposito(valor), caso [Saldo >=50.000] então irá sofrer mudança de estado para “Gold” e caso Saldo < 50.000 irá permanecer no estado “Silver””. Além disso, a conta não poderá passar diretamente do estado “Silver” para “Platinium” em apenas um depósito, independentemente do valor, o teste com o valor de deposito 300.000 exercita esta regra.

  
”Diagrama da Máquina de estados”

O Nó “Gold” também irá seguir o mesmo padrão do “ChoicePseudoState”, mas com seu valor de “Guard” para upgrade como sendo [Saldo >= 200.000]. Já o Nó “Platinium” não existe estado disponível para upgrade através do depósito, logo seu estado não será alterado.

**1.1.2 Teste do Nó “Gold” visando upgrade via depósito:**

O teste irá seguir a mesma lógica apresentada anteriormente para o Nó Silver, mudando evidentemente os valores limite para upgrade (Guards). Assim como no Nó Silver e no nó Platinium, este **teste também implementa a estratégia de Particionamento e Valor Limite,** *uma vez que cada Nó pode ser tratado como uma partição de teste diferente e os dados de entrada utilizados exercitam os limites destas partições, apresentando os valores on-point e off-point de cada partição.*

@ParameterizedTest

    @CsvSource({

        "1000,Gold",

        "300000,Platinium",

        "50000,Gold",

        "150000,Platinium", //Limite on-point

        "149999,Gold" // Limite off-point

    })

    public *void* upgradeViaDeposito\_ContaGold(*double* *valorDeposito*, String *tipoContaEsperado*){

        //Teste de partição (C2) + Teste de limite para os upgrades da conta gold (>=200.000)

        ContaCorrente c = **new** ContaCorrente("2", "Teste upgrade conta Gold");

        c.deposito(50000);//Upgrade Gold

        c.deposito(*valorDeposito*);

        assertEquals(*tipoContaEsperado*, c.getCategoria());

    }

Este caso teste *teve que ser reescrito para a segunda versão do driver*, uma vez que o valor **“149999,Gold”** estava, na verdade, fazendo com que a conta tivesse um upgrade para “Platinium” devido ao depósito ser valorizado em 1% nas contas Gold (149999\*1,01 = 151498,99). O valor correto para o teste de limite off-point é “148513,Gold”, o CSVSource atualizado fica da seguinte forma:

@CsvSource({

        "1000,Gold",

        "300000,Platinium",

        "50000,Gold",

        "150000,Platinium", //Limite on-point

        "148513,Gold" // Limite off-point

    })

**1.1.3 Teste do Nó “Platinium” visando upgrade via depósito:**

O teste irá seguir a mesma lógica apresentada para o Nó Silver, mudando evidentemente os valores limite para upgrade (Guards). Assim como no Nó Silver e no nó Platinium, este **teste também implementa a estratégia de Particionamento e Valor Limite,** *uma vez que cada Nó pode ser tratado como uma partição de teste diferente e os dados de entrada utilizados exercitam os limites destas partições, apresentando os valores on-point e off-point de cada partição.* Uma conta “Platinium” não possui uma categoria superior e por isso permanecerá neste estado para qualquer valor depositado.

@ParameterizedTest

    @CsvSource({

        "1,Platinium",

        "300000,Platinium",

        "50000,Platinium",

    })

    public *void* upgradeViaDeposito\_ContaPlatinium(*double* *valorDeposito*, String *tipoContaEsperado*){

        //Teste de partição (C3) + Teste para garantir permanencia na conta Platinium.

        ContaCorrente c = **new** ContaCorrente("3", "Teste upgrade conta Platinium");

        c.deposito(100000);//Upgrade Gold

        c.deposito(100000);//Upgrade Platinium

        c.deposito(*valorDeposito*);

        assertEquals(*tipoContaEsperado*, c.getCategoria());

    }

* 1. **Testes de “downgrade” através de retiradas:**

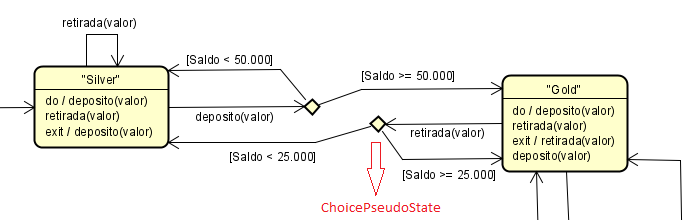
**1.2.1 Teste do Nó “Gold” visando downgrade via retirada:**

Teste de downgrade de categoria para conta do tipo “Gold” através do uso do método retirada(valor). Este teste irá exercitar a parte referente ao Nó “Gold” na execução do método “retirada(valor)” do diagrama de estados.

1. @ParameterizedTest
2. @CsvSource({
3. "1000,Gold",//out-point
4. "300000,Gold", //inpoint
5. "45000,Gold",//off-point
6. "45001,Silver"//on-point
7. })
8. public *void* downgradeViaRetirada\_ContaGold(*double* *valorRetirada*, String *tipoContaEsperado*){
9. //Teste de partição (C2) + Teste de limite para o downgrade da conta gold para silver (< 25.000).
10. ContaCorrente c = **new** ContaCorrente("4", "Teste downgrade conta Gold");
11. c.deposito(70000);//Upgrade gold
12. c.retirada(*valorRetirada*);
13. assertEquals(*tipoContaEsperado*, c.getCategoria());
14. }

Utilizado como exemplo a entrada ”45.001,Silver” do teste parametrizado, que é o limite *on-point (estratégia de limites)* deste caso, o “**ChoicePseudoState**” utilizado no diagrama de estados, logo após a execução do método “retirada(valor)” irá seguir o fluxo através do “Guard” [Saldo < 25.000], alterando o estado da propriedade “categoria” para “Silver”.

A imagem que segue expõe o ChoicePseudoState e seus Guards: [Saldo < 25.000] e [Saldo >= 25.000]. Cada Guard irá definir o estado a ser adotado após o método “retirada(valor)”, caso [Saldo < 25.000] então o estado de “Gold” irá sofrer mudança para “Silver”, do contrário irá permanecer no estado “Gold””.

  
”Diagrama da Máquina de estados”.

O Nó “Platinium” também irá seguir o mesmo padrão do “ChoicePseudoState”, mas com seu valor de “Guard” para downgrade como sendo [Saldo < 100.000]. Já o Nó “Silver” não existe estado disponível para downgrade através da retirada, logo seu estado não será alterado por este método.

**1.2.2 Teste do Nó “Platinium” visando downgrade via retirada:**

O teste irá seguir a mesma lógica apresentada para o Nó Gold, mudando evidentemente os valores limite para downgrade (Guards). Assim como no Nó Silver e no nó Platinium, este **teste também implementa a estratégia de Particionamento e Valor Limite,** *uma vez que cada Nó pode ser tratado como uma partição de teste diferente e os dados de entrada utilizados exercitam os limites destas partições, apresentando os valores on-point e off-point de cada partição.* Além disso, **a conta não poderá sofrer downgrade de mais de um estado por retirada**, o teste que exercita esta regra é o de valor de retirada 400.000, onde mesmo a conta permanecendo com apenas 20.000 de saldo o estado deverá ser atualizado para “Gold” (apenas 1 nível “inferior” a Platinium).

@ParameterizedTest

    @CsvSource({

        "1000,Platinium",//out-point

        "400000,Gold", //inpoint

        "500000,Platinium", //Saldo insuficiente, não irá retirar nada.

        "320000,Platinium",//off-point

        "320201,Gold"//on-point

    })

    public *void* downgradeViaRetirada\_ContaPlatinium(*double* *valorRetirada*, String *tipoContaEsperado*){

        //Teste de partição (C2) + Teste de limite para o downgrade da conta gold para silver (< 25.000).

        ContaCorrente c = **new** ContaCorrente("5", "Teste downgrade conta Platinium");

        c.deposito(400000);//Upgrade Gold

        c.deposito(20000);//Upgrade Platinium

        c.retirada(*valorRetirada*);

        assertEquals(*tipoContaEsperado*, c.getCategoria());

    }

**1.2.3 Teste do Nó “Silver” visando downgrade via retirada:**

O Nó “Silver” não irá seguir o mesmo padrão dos Nós Gold e Platinium, uma vez que este Nó não possui estado anterior para downgrade. Todas as retiradas podem ser feitas enquanto houver saldo suficiente para este, sem modificar o estado do nó. Este teste **não foi inserido na primeira versão do driver de teste**, estando disponível apenas na segunda versão.

O teste busca validar a permanência do estado “Silver” do nó, independente das retiradas realizadas.

 @ParameterizedTest

    @CsvSource({

        "1000,Silver",

        "300,Silver",

        "45,Silver",

        "1259.50,Silver"

    })

    public *void* downgradeViaRetirada\_ContaSilver(*double* *valorRetirada*, String *tipoContaEsperado*){

        //Teste de partição (C1) + Teste de downgrade em conta Silver..

        ContaCorrente c = **new** ContaCorrente("15", "Teste downgrade conta Silver");

        c.deposito(1000);//Deposito inicial, para possibilitar retiradas.

        c.retirada(*valorRetirada*);

        assertEquals(*tipoContaEsperado*, c.getCategoria());

    }

* 1. **Estratégias de geração de casos de teste**

Além dos testes acima descritos para validação do funcionamento dos estados “Silver”, “Gold” e “Platinium”, feitos como base no Diagrama de Estados, que também já apresentarem alguns aspectos dos testes com estratégia de particionamento e de valor limite, os testes abaixo buscam exercitar os demais requisitos do projeto.

* 1. **Estratégia de particionamento e de valor limite**

Com estas estratégias busquei particionar o problema em duas partições de **parâmetros diferentes**, a primeira sendo o **tipo de conta** (C1 = Silver, C2 = Gold e C3 = Platinium) e a segunda sendo o valor de entrada para o **depósito (VD)**. Com esta estratégia busquei exercitar o código relativo ao depósito valorizado em cada um dos tipos de conta.

**2.1.2 Deposito Valorizado conta Silver (C1):**

Nota-se que neste caso de teste o valor depositado será exatamente igual ao valor original, sem valorização alguma. Portanto, o valor esperado é o depositado \* 1. **Este teste utiliza a estratégia de particionamento.**

@ParameterizedTest

    @CsvSource({

        "100",

        "120000",

        "500"

    })

    public *void* depositoValorizadoSilver(*double* *valorDeposito*){

        ContaCorrente c = **new** ContaCorrente("9", "Teste valorização de depósito conta Silver");

*double* saldoPreDeposito = c.getSaldo();

        c.deposito(*valorDeposito*);

        assertEquals(*valorDeposito* \* 1, (c.getSaldo() - saldoPreDeposito));

    }

**2.1.2 Deposito Valorizado conta Gold (C2):**

Nota-se que neste caso de teste parametrizado estamos trabalhando com uma conta do tipo “Gold”, o qual tem a valorização de 1% sobre todo depósito. O teste a seguir verifica se a **diferença** entre o saldo após o depósito e antes do depósito é igual ao valor esperado de valorização (valorDeposito \* 1.01). **O cálculo possui um delta aceitável de diferença de 0,01. Este teste utiliza a estratégia de particionamento.**

@ParameterizedTest

    @CsvSource({

        "100",

        "120000",

        "500.075"

    })

    public *void* depositoValorizadoGold(*double* *valorDeposito*){

        ContaCorrente c = **new** ContaCorrente("10", "Teste valorização de depósito conta Gold");

        c.deposito(100000);//Upgrade Gold

*double* saldoPreDeposito = c.getSaldo();

        c.deposito(*valorDeposito*);

        assertEquals(*valorDeposito* \* 1.01, (c.getSaldo() - saldoPreDeposito), 0.01);

    }

**2.1.3 Deposito Valorizado conta Platinium (C3):**

Nota-se que neste caso de teste parametrizado estamos trabalhando com uma conta do tipo “Platinium”, o qual tem a valorização de 2,5% sobre todo depósito. O teste a seguir verifica se a **diferença** entre o saldo após o depósito e antes do depósito é igual ao valor esperado de valorização (valorDeposito \* 1.025). **O cálculo possui um delta de diferença de 0,01. Este teste utiliza a estratégia de particionamento.**

 @ParameterizedTest

    @CsvSource({

        "100",

        "120000",

        "500"

    })

    public *void* depositoValorizadoPlatinium(*double* *valorDeposito*){

        ContaCorrente c = **new** ContaCorrente("11", "Teste valorização de depósito conta Platinium");

        c.deposito(100000);//Upgrade Gold

        c.deposito(100000);//Upgrade Platinium

*double* saldoPreDeposito = c.getSaldo();

        c.deposito(*valorDeposito*);

        assertEquals(*valorDeposito* \* 1.025, (c.getSaldo() - saldoPreDeposito), 0.1);

    }

**2.1.4 Teste do método deposito com valores limite válidos/inválidos:**

Este teste parametrizado busca exercitar o método deposito() com os valores limite para este método, onde um **depósito não pode possuir valor menor ou igual a 0.** Sendo o valor “0.0” o ***on-point*** e todos os valores menores que ele sendo os valores ***out-points*** o método irá retornar “false”, pois, a operação não pode ser executada. Naturalmente, o valor “0.01” será o ***off-point*** e todos maiores que 0.0 serão ***in-points*** e por este motivo o método irá retornar “true”. Estas regras do método independem dos particionamentos, se comportando igualmente em todas as partições.

@ParameterizedTest

    @CsvSource({

        "0.0,false",//Invalid <= 0 - on-point

        "-1,false",//Invalid <= 0 - out-point

        "0.01,true", // off-point

        "253897,true" // in-point

    })

    public *void* depositoTest(*double* *valorDeposito*, String *statusDepositoEsperado*){

        ContaCorrente c = **new** ContaCorrente("6", "Teste downgrade conta Platinium");

        assertEquals(*statusDepositoEsperado*, c.deposito(*valorDeposito*));

    }

Este teste teve que ser reescrito após a primeira execução, visto que era necessário converter o valor textual “false” e “true” para um valor *booleano*. Para isso, o código foi atualizado com a função parseBoolean(string) da API de Booleanos do Java e ficou desta forma (O restante do código não foi modificado):

assertEquals(Boolean.parseBoolean(*statusDepositoEsperado*), c.deposito(*valorDeposito*));

**2.1.5 Teste do método retirada com valores limite:**

Estes testes foram separados em dois, o primeiro para verificar o limite de retirada quando o Saldo é insuficiente ou suficiente e o outro para testar a solicitação e retirada de valores inválidos.

**2.1.5.1 Testes relativos a saldo insuficiente/suficiente para retirada:**

Este teste parametrizado busca exercitar o método *retirada()* com os valores limite para este método, onde **o valor de uma retirada não pode ser maior que o saldo da conta.** Neste caso de teste foi utilizado o *valor base de saldo 1000*, sendo assim o valor limite ***off-point*** será 1000 e por consequência, valores menores que 1000 e maiores que 0 serão os ***in-points***. Valores de retirada maiores que 1000, como por exemplo 1001 (***on-point***), serão maiores que o saldo disponível e por este motivo retornarão “false” no método “retirada”. Estas regras do método independem dos particionamentos, se comportando igualmente em todas as partições.

@ParameterizedTest

    @CsvSource({

        "1000,true", //off-point - Saldo Suficiente.

        "845.25,true", //in-point - Saldo Suficiente.

        "1001,false",//on-point - Saldo Insuficiente.

        "5098,false"//out-point - Saldo Insuficiente.

    })

    public *void* retiradaSaldoInsuficienteTest(*double* *valorRetirada*, String *statusRetiradaEsperado*){

        ContaCorrente c = **new** ContaCorrente("7", "Teste downgrade conta Platinium");

        c.deposito(1000); //Valor base para simulações.

        assertEquals(*statusRetiradaEsperado*, c.retirada(*valorRetirada*));

    }

Este teste teve que ser reescrito após a primeira execução, visto que era necessário converter o valor textual “false” e “true” para um valor booleano. Para isso, o código foi atualizado com a função parseBoolean(string) da API de Booleanos do Java e ficou desta forma (O restante do código não foi modificado):

 assertEquals(Boolean.parseBoolean(*statusRetiradaEsperado*), c.retirada(*valorRetirada*));

**2.1.5.2 Testes relativos a valores inválidos para retirada:**

Este teste parametrizado busca exercitar o método *retirada()* com os valores limite para este método, onde **o valor de uma retirada não pode ser 0 ou menor que 0.** Sendo assim, o valor limite ***On-Point*** será “0.0” e os valores ***Out-Point*** serão todos aqueles menores que “0.0” e para estas entradas o método retirada deve retornar “false”. Juntamente a isso, valores maiores que “0.0” e que possuam saldo disponível na conta devem retornar “true” para a atividade de “retirada”. Estas regras do método independem dos particionamentos, se comportando igualmente em todas as partições.

@ParameterizedTest

    @CsvSource({

        "0.0,false", //Invalid <= 0 - On-Point

        "-1,false", //Invalid <= 0 - Out-Point

        "0.01,true", //Valid > 0 - Off-point

        "50,true", // In-point

    })

    public *void* retiradaValorInvalidoTest(*double* *valorRetirada*, String *statusRetiradaEsperado*){

        ContaCorrente c = **new** ContaCorrente("8", "Teste downgrade conta Platinium");

        c.deposito(1000); //Valor base para simulações.

        assertEquals(*statusRetiradaEsperado*, c.retirada(*valorRetirada*));

    }

Este teste teve que ser reescrito após a primeira execução, visto que era necessário converter o valor textual “false” e “true” para um valor booleano. Para isso, o código foi atualizado com a função parseBoolean(string) da API de Booleanos do Java e ficou desta forma (O restante do código não foi modificado):

 assertEquals(Boolean.parseBoolean(*statusRetiradaEsperado*), c.retirada(*valorRetirada*));

**2.1.6 Testes adicionais para validação de métodos base e cobertura de código eficiente.**

Os testes a seguir têm por objetivo assegurar o funcionamento correto dos métodos base da classe ContaCorrente.java. Os métodos testados são os seguintes:

**2.1.6.1 Método Construtor:**

O construtor da classe Conta Corrente deve receber como parâmetro um número que deve ser maior ou igual a 0. Caso o número seja menor que o esperado será lançado uma exceção, conforme descrito no primeiro teste abaixo.

O segundo teste do construtor exercita o código em condições “normais”, onde o número informado é maior ou igual a 0.

/\* Numero da conta < 0 \*/

    @Test

    public *void* testaConstrutorNrInvalido(){

        assertThrows(IllegalArgumentException.class,() *->* {

            ContaCorrente c = **new** ContaCorrente("-1", "Teste numero inválido construtor.");

        });

    }

    /\* Numero da conta >= 0 \*/

    @Test

    public *void* testaConstrutorNrValido(){

        ContaCorrente c = **new** ContaCorrente("0", "Teste numero válido construtor.");

        assertEquals("0", c.getNumeroConta());

    }

**2.1.6.1 Métodos “get”:**

Todos os quatro métodos abaixo foram implementados para assegurar o correto funcionamento de métodos “get” para retornar os valores e formato/tipo esperados. Além disso, é importante o teste deste tipo de método para assegurar a máxima cobertura do código.

    @Test

    public *void* getCategoriaTest(){

        ContaCorrente c = **new** ContaCorrente("12", "Teste de método getCategoria");

        assertEquals("Silver", c.getCategoria());

    }

    @Test

    public *void* getNomeCorrentistaTest(){

        ContaCorrente c = **new** ContaCorrente("13", "Bruno");

        assertEquals("Bruno", c.getNomeCorrentista());

    }

    @Test

    public *void* getNumeroConta(){

        ContaCorrente c = **new** ContaCorrente("14", "Bruno");

        assertEquals("14", c.getNumeroConta());

    }

    @Test

    public *void* getSaldoTest(){

        ContaCorrente c = **new** ContaCorrente("9", "Teste método getSaldo");

        c.deposito(70000);

        assertEquals(70000, c.getSaldo());

    }

1. **Código da classe implementada**

Após a definição de *todos os casos acima citados que formaram a primeira versão do diver de teste*, exceto aqueles que foram informados explicitamente que foram inseridos na segunda versão do driver, a classe ContaCorrente foi implementada da seguinte forma:

**Primeira e *única* versão da classe “ContaCorrente”:**

package com.bsimm;

public class ContaCorrente{

    private String nroConta;

    private String nomeCorrentista;

    private *double* saldo;

    private String categoria;

    public ContaCorrente(String *nro*, String *nome*){

*double* nroDouble = Double.parseDouble(*nro*);

        //Floor irá arredondar "p/ baixo". Testa se o num é inteiro.

        if (((Math.floor(nroDouble)) - nroDouble) != 0){ //Número não inteiro.

            throw **new** IllegalArgumentException("O número da conta deve ser um numero inteiro.");

        }

        if(Long.parseLong(*nro*) < 0){

            throw **new** IllegalArgumentException("O número da conta deve ser maior ou igual a 0.");

        }

*this*.nroConta = *nro*;

*this*.nomeCorrentista = *nome*;

*this*.saldo = 0.0; //Inicializando.

*this*.categoria = "Silver"; //Inicializando.

    }

    public String getNumeroConta(){

        return *this*.nroConta;

    }

    public String getNomeCorrentista(){

        return *this*.nomeCorrentista;

    }

    public *double* getSaldo(){

        return *this*.saldo;

    }

    public String getCategoria(){

        return *this*.categoria;

    }

    public *boolean* deposito(*double* *valor*){

        if (*valor* <= 0){

            return false;

        }

        if (*this*.categoria.equals("Silver")){

*this*.saldo += *valor* \* 1;

            if (*this*.saldo >= 50000) {

*this*.categoria = "Gold";

            }

            return true;

        } else if (*this*.categoria.equals("Gold")){

*this*.saldo += *valor* \* 1.01;

            if (*this*.saldo >= 200000) {

*this*.categoria = "Platinium";

            }

            return true;

        } else if (*this*.categoria.equals("Platinium")){

*this*.saldo += *valor* \* 1.025;

            return true;

        }

        return false;

    }

    public *boolean* retirada(*double* *valorSaque*){

        if ((*valorSaque* > *this*.saldo) || (*valorSaque* <= 0)){

            return false;

        }

        if (*this*.categoria.equals("Silver")){

*this*.saldo -= *valorSaque*;

            return true;

        } else if (*this*.categoria.equals("Gold")){

*this*.saldo -= *valorSaque*;

            if (*this*.saldo < 25000){

*this*.categoria = "Silver";

            }

            return true;

        } else if (*this*.categoria.equals("Platinium")){

*this*.saldo -= *valorSaque*;

            if (*this*.saldo < 100000){

*this*.categoria = "Gold";

            }

            return true;

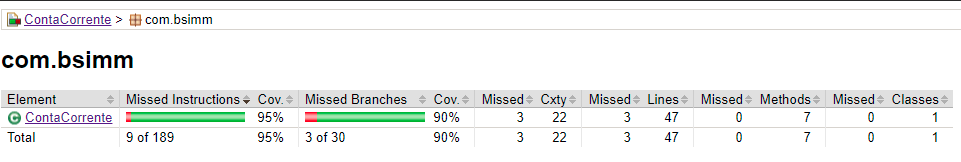
        }

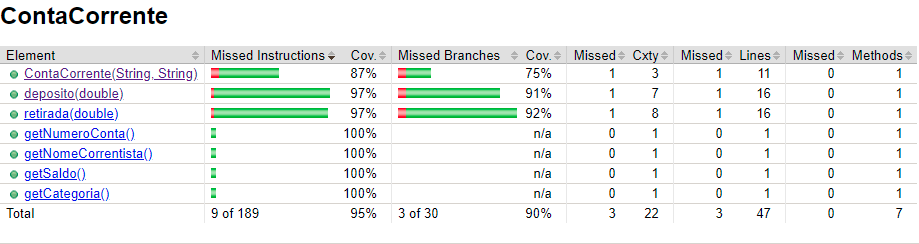
        return false;

    }

}

**3.1 Cobertura de teste do primeiro driver de teste juntamente com a primeira versão da classe:**

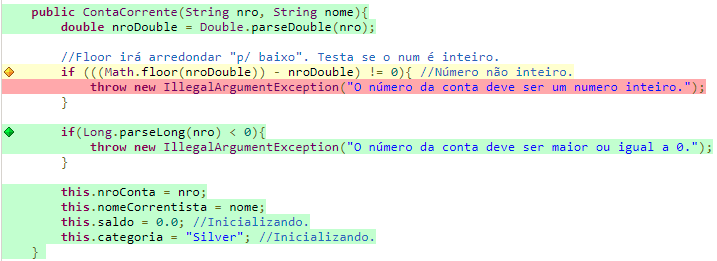
**Resultado da verificação feita pela ferramenta JaCoCo para cobertura de código:  
**

****

**3.2 Problemas apontados pela ferramenta JaCoCo:**

**3.2.1 Método Construtor:**

Na primeira versão do driver de testes a implementação do teste feito abaixo não esteve presente. Este teste tem por objetivo verificar se o número inserido para conta é decimal.

****“Erro apontado pelo JaCoCo, apontando a não cobertura da verificação”.

O teste implementado para exercitar esta verificação é o seguinte:

 /\* Numero da conta decimal - 2 versão driver teste\*/

     @Test

     public *void* testaConstrutorNrInvalidoDecimal(){

        assertThrows(IllegalArgumentException.class,() *->* {

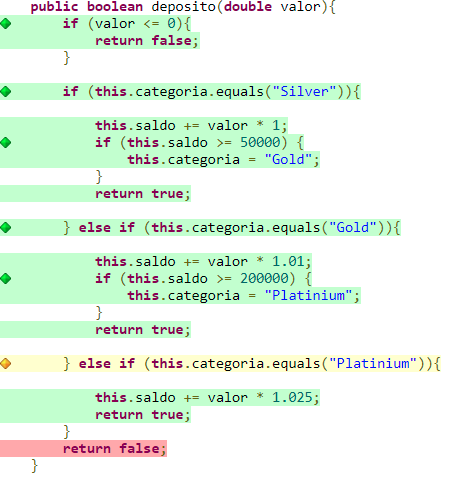
            ContaCorrente c = **new** ContaCorrente("12.5", "Teste numero decimal construtor.");

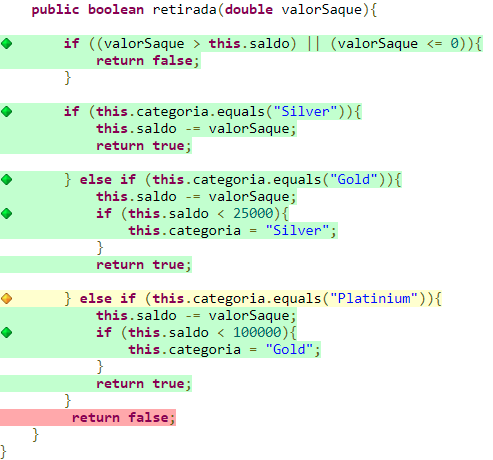
        });

    }

Depois desse teste restaram apenas os problemas no método deposito e retirada, que seguem.

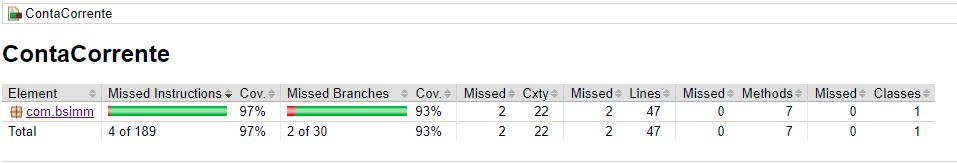
**3.2.2 Problemas com os métodos deposito e retirada:**

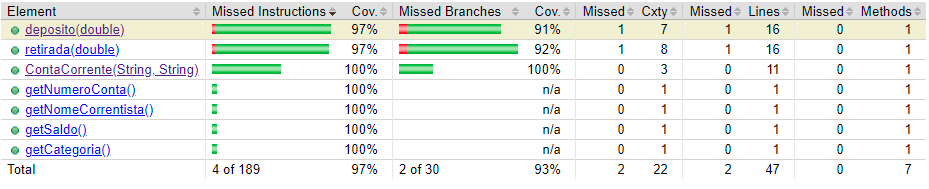
  
“Método deposito (2) erros”.

  
“Método retirada (2) erros”.

Note que em ambos os métodos os erros são iguais. O **alerta em vermelho** para a instrução “return false” no final do método, não foi exercitada por nenhum método de teste. Como a implementação não permite nenhuma outra categoria diferente de Silver, Gold e Platinium este ponto do código não será executado, exceto se por um motivo neste momento ainda desconhecido, uma categoria diferente destas 3 for definido erroneamente. Como o atributo “categoria” é privado e não disponibilizamos um método para “setCategoria”, a definição das categorias se dará exclusivamente através dos métodos “deposito” e “retirada” onde será **impossível definir uma categoria fora das 3 pré-estabelecidas.**

O **segundo alerta**, em amarelo, aponta a seguinte mensagem: *“1 of 2 branches missed”.* Busquei uma solução para este alerta nas documentações do JaCoCo mas infelizmente até o momento não encontrei uma solução para o problema em especifico. Entretanto, com as pesquisas pude compreender que o JaCoCo irá analisar o byte code gerado pelo compilador do Java e por este motivo ele irá analisar coberturas de ramificação diferentes das apontadas na codificação feita em “Java”.

Após a implementação do teste para o construtor da classe a análise de cobertura ficou em **97%,** conforme análise abaixo da ferramenta JaCoCo:  




**4 Conclusão**

A implementação dos casos de teste no início do processo de desenvolvimento se mostrou, em minha percepção, muito eficiente na atividade de compreender os requisitos e regras que deveriam ser implementados na codificação da classe alvo, fato que fez com que o desenvolvimento da classe fosse mais objetivo e eficiente. Além disso, uma futura manutenção desta classe estaria “assegurada” pelo driver de teste e reduziria o impacto de possíveis erros e também facilitaria a compreensão futura da lógica aplicada ao desenvolvimento da classe alvo e do driver de teste.

Somado a isso, foi possível garantir com segurança que o código desenvolvido estava de acordo com o esperado pela especificação do projeto proposto, apresentando “praticamente” (exceto a questão das branches e do return false, reduzindo para 97%) 100% de cobertura das situações possíveis para todos os métodos.

**Adicionalmente deixo os arquivos gerados pela análise da ferramenta JaCoCo disponíveis para consulta, caso for necessário:** https://drive.google.com/drive/folders/1NXxYkrcDZrLqbZva23zdzCYWNItBE1xC?usp=sharing