Aridio Gomes da Silva - aridiosilva@aridiosilva.com - https://www.linkedin.com/in/aridio-silva-74997111/

INTRODUÇÃO

Este é o meu **Relatório da TAREFA FINAL do Curso de Introdução ao Teste de Software (Plataforma Coursera em parceria com a USP),** cujo prazo de entrega é 19 de outubro de 2020, contendo descrição e detalhamento solicitados, bem como recomendações e requisitos especificados no documento "*Tarefa Avaliada por Colega: Resolução de Casos Utilizando os Conceitos Estudados*".

As seções deste relatório seguem o formato descrito no "*Template de Apoio à Atividade de Teste de Software (Criado para o MOOC Introdução ao Teste de Software – Plataforma Coursera*", são respectivamente:

- ***Seção 1 – Cenário Geral do Algoritmo (O que será testado)*** - apresentado uma visão geral do algoritmo testado, descrevendo o que ele é, o que ele faz, quais são as suas principais funcionalidades, e quais as funcionalidades e módulos testados;
- ***Seção 2 – Estratégias de Testes (Como será testado)*** – apresenta de acordo com o cenário e escopo daquilo que foi testado, o detalhamento de quais atividades de teste foram conduzidas e das técnicas de testes usadas com os respectivos objetivos e finalidades de cada uma. Esta seção contém ainda o detalhamento dos critérios utilizados para cada técnica. Como complemento, tem-se o detalhamento da ferramenta utilizada descrita de forma geral.
- ***Seção 3 – Projeto dos Casos de Testes (Como será testado)*** - apresenta, para cada técnica e critério considerados, quais são os casos de testes a serem utilizados (roteiro e/ou dados de entrada, resultados esperados). Por exemplo, será usado o Teste Funcional e Critério Análise do Valor Limite, e para isso teremos a tabela final considerando as variáveis de entrada, as classes de equivalência válidas e inválidas, bem como os casos considerando os limites.
- ***Seção 4 – Execução (Quando e Como será testado)*** – apresenta os detalhes sobre a execução, explicando como o teste foi executado. Por serem utilizadas ferramentas de apoio (mesmo que não tenha sido apresentada no curso) para os casos de teste considerados contam especificados os resultados retornados (resultados dos testes) e a descrição dos erros (caso tenham ocorrido)..
- ***Seção 5 – Análise dos Resultados e Próximos Passos*** – apresenta a conclusão obtida após a realização dos testes, além de considerações sobre outras técnicas ou ferramentas adicionais necessárias, e fala sobre a situação onde, se tivesse mais tempo, para realizar outros testes, quais seriam os próximos passos que poderiam ser realizados.

Seção 1 – Cenário Geral do ALGORÍTMO Escolhido (O Que Será Testado)

Aqui está sendo apresentado uma visão geral do Algoritmo que será **testado**, e portanto, serão providas as descrições para os seguintes tópicos:

- *o que ele é,*
- *o que ele faz,*
- *quais são as suas principais funcionalidades,*
- *quais serão as funcionalidades partes dele serão testados*.

No Brasil todo o rendimento mensal do trabalho assalariado de pessoas físicas, tanto de pessoas sob o regimento CLT (com carteira assinada), quanto pessoas sob trabalho autônomo e, também, daquelas sob o regime de trabalho temporário, estão sujeitas ao recolhimento mensal (por meio de retenção na fonte) dos rendimentos pagos por pessoas físicas ou jurídicas. Assim, o algoritmo a ser usado ao longo deste Relatório será justo este utilizado numa grande gama de aplicações e diferentes sistemas, que consta o detalhamento deste Algoritmo nas seções abaixo:

1.1 – O Que é o Algoritmo Escolhido para os Testes-

A seguir a descrição do que é o Algoritmo escolhido para realizar toda a sequência obrigatória das técnicas de **Testes Funcionais, Estruturais e de Mutação** descritas neste relatório, respectivamente:

> ➢ *É uma **Classe**, de um **programa escrito em Java**, cuja **responsabilidade** é proceder o cálculo do Imposto de Renda sobre Rendimento recebido de Trabalho Assalariado de pessoas físicas.*

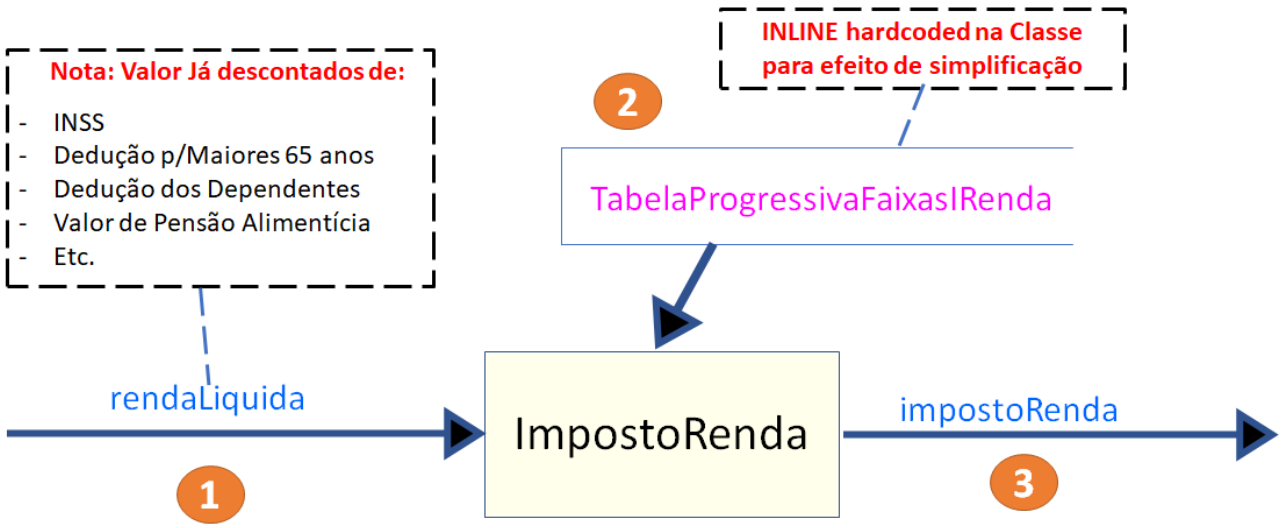
1.2 – O Que Faz a *Classe ImpostoRenda*

1.2.1 – Responsabilidades da Classe

A ***Classe ImpostoRenda*** possui a seguinte **funcionalidade única**:

> ✓ *Efetuar o cálculo do imposto de renda a ser retido na fonte sobre a remuneração de trabalho assalariado - é um procedimento único e válido para todos os diferentes regimes trabalhistas das pessoa físicas assalariadas.*

1.2.2 – Os Inputs e Outputs e Constraints da *Classe ImpostoRenda*:



Acima o **diagrama do Escopo** da ***Classe ImpostoRenda*** onde se visualiza o **input (1)** que **é renda líquida** da pessoa física assalariada para a qual se quer calcular o **output (3)**, que é o **imposto de renda** a ser retido na fonte. Para tanto a Classe utiliza INLINE e hard coded a Tabela Progressiva de Faixas do Imposto de Renda.

a) Constraints da *Classe ImpostoRenda*

Com vistas a simplificação das regras de negócio da ***Classe ImpostoRenda***, define-se as seguintes restrições:

1ª Restrição – O valor da renda a ser calculado o imposto pela ***Classe ImpostoRenda*** deve vir já descontado de todas as deduções previstas em lei e enumeradas no diagrama acima, assim o valor da renda deve ser líquida contendo descontado do valor do INSS, do valor da dedução para pessoas com mais de 65 anos, do valor de dedução por dependentes, da dedução de pensão alimentícia, e quaisquer outras deduções legais aplicáveis. Assim, a responsabilidade da ***Classe ImpostoRenda*** é simplificada apenas para cálculo do valor do imposto a ser retido.

2ª Restrição – Com vistas a *eliminar o acoplamento e dependências externas e assim excluir a necessidade do uso de mocks* nos *testes unitários (unit tests)* durante o processo de *desenvolvimento orientado a teste (TDD)* do softwarea, as faixas da

tabela progressiva das faixas do imposto de renda estarão codificadas *inline* no código fonte da ***Classe ImpostoRenda***, em vez de ser implementada por meio de acesso à fonte de dados externa ao algoritmo (*database*).

b) Inputs da *Classe ImpostoRenda*

Para que a ***Classe ImpostoRenda*** consiga realizar o cálculo do valor a ser retido de imposto sobre a renda ela se utiliza dos seguintes elementos:

- ✓ *do* ***Valor da Renda Líquida*** – Valor de *input* da Classe e sobre o qual deve ser calculado o imposto a ser retido e descontado do rendimento do trabalho assalariado;

- ✓ *de uma* ***Tabela Progressiva de Faixas do Imposto de Renda*** *- onde estão identificadas várias faixas de valores a tributar, a alíquota do imposto que se aplica para cada faixa de valores, e o valor do desconto a abater da base de cálculo do imposto antes de aplicar o percentual de tributação do imposto de renda)* . Esta tabela é definida e atualizada de vez enquanto pela Secretaria da Receita Federal, órgão subordinado ao Ministério da Fazenda. Abaixo a estrutura da Tabela Progressiva do Imposto de Renda em vigência atualmente e válida a partir de abril de 2015 :

Tabela Imposto de Renda 2020

Base de cálculo (R$)	Alíquota (%)	Parcela a deduzir do IRPF (R$)
Menor do que 1.903,99	ISENTO	0,00
De 1.903,99 até 2.826,65	7,5	142,80
De 2.826,66 até 3.751,05	15	354,80
De 3.751,06 até 4.664,68	22,5	636,13
Acima de 4.664,68	27,5	869,36

c) Output da *Classe ImpostoRenda*

Como o objetivo funcional da ***Classe ImpostoRenda*** é devolver como resultado de seu processamento:

- ✓ *o* ***Valor do Imposto de Renda Calculado*** - Valor devolvido pelos métodos da Classe após ter realizado os procedimentos do cálculo requerido.

1.2.3 – Comportamento (Behaviour Rule) da *Classe ImpostoRenda*

Os passos do procedimento para cálculo do imposto de renda são respectivamente na sequência e os indicados abaixo:

Passo 1 - Se o Valor da Renda Líquida for menor do que zero devolver valor zero porque a renda liquida não pode ser negativa para calculo do imposto de renda na fonte -- pois a pessoa se enquadrará na condição de isento de imposto de renda

Passo 2 - Se o Valor da Renda Líquida for igual a zero deverá retornar o valor zero porque não há imposto a calcular com renda liquida igual a zero - pois a pessoa se enquadrará na condição de isento de imposto de renda;

Passo 3 - Para valores maiores do que Zero deve-se comparar o valor da renda liquida com cada valor das Faixas de Valores da Tabela Progressiva do Imposto de Renda de modo a identificar a que faixa se enquadra a renda líquida que se deseja calcular o imposto de renda;

Passo 5 – Ao identificar o valor da faixa de valores a que se enquadra a renda líquida, deve-se aplicar a alíquota do imposto associada à faixa, do valor do imposto calculado deve-se abater o valor da dedução correspondente àa faixa de valor previsto na lei;
Passo 6 – Deve-se retornar o valor do imposto de renda calculado;

1.2.4 – A Implementação do Algoritmo da *Classe ImpostoRenda* em JAVA

Abaixo o código fonte da ***Classe ImpostoRenda*** escrito na Versão 8 da Linguagem Orientada à Objetos JAVA, cujo código fonte será utilizado em toda a sua plenitude nos *testes Funcionais, Estruturais e de Mutação*:

```
public class ImpostoRenda {

	public float calculaImpostoRenda(float renda) {

		if (renda < 1903.99f) {
		// Faixa 1 - Menor do que 1.903.99 Isento de Imposto de Renda
			return 0.0f;
		}
		if (renda <= 2826.65f) {
		// Faixa 2 - de 1.903,99 até 2.826,65 Taxa 7,5% Dedução 142,80
			return (float)((Math.round(renda * 0.075)) - 142.80f);
		}
		if (renda <= 3751.05f) {
		// Faixa 3 - de 2.826,66 até 3.751,05 Taxa 15% Dedução 354,80
			return (float)((Math.round(renda * 0.15f)) - 354.80f);
		}
		if (renda <= 4664.68f) {
		// Faixa 4 - de 3.751,06 até 4.664,68 Taxa 22,5% Dedução 636,13
			return (float)((Math.round(renda * 0.225f)) - 636.13f);
		} else {
		// Faixa 5 - Acima de 4.664,68 Taxa    27,5% Dedução 869,36
			return (float)((Math.round(renda * 0.275f)) - 869.36f);
		}
	}
}
```

Seção 2 – ESTRATÉGIAS de TESTE (Como será Testado) da *Classe ImpostoRenda*

De acordo com o cenário/escopo daquilo que será testado, descreva quais atividades de teste serão conduzidas. Quais técnicas de teste serão usadas e com qual objetivo/finalidade? Qual ou quais critérios serão utilizados para cada técnica? Se alguma ferramenta for utilizada, descreva-a(s) aqui, de forma geral.

Em atendimento as exigências especificadas no documento "*Tarefa Avaliada por Colega: Resolução de Casos Utilizando os Conceitos Estudados*", há necessidade de aplicar na sequência os seguintes tipos de testes: Funcional, Estrutural e de Mutação. Assim sendo, para a *Classe ImpostoRenda* serão aplicados estes tipos de testes conforme explicações descritas a seguir:

Seção 3 - Teste Funcional da *Classe ImpostoRenda*

3.1 – Finalidade do Teste Funcional

Na realização do Teste Funcional será utilizada a técnica de Teste de Domínio, com a finalidade de analisar o comportamento da *Classe ImpostoRenda* sob o ponto de vista do usuário. Para isso, a *Classe ImpostaRenda* será vista como uma CAIXA PRETA - *não importando como foi feita a implementação interna das solução, mas o que vale é avaliar se os requisitos funcionais para os quais a Classe foi desenvolvida são atendidos em toda a plenitude – precisamos analisar se estará calculando corretamente o valor do imposto de renda para valores informados de renda líquida da pessoa física assalariada*.

Desta forma, o foco do teste funcional é avaliar a responsabilidade específica da *Classe ImpostoRenda*, e para isso, será submetido um **Conjunto de Testes** com vistas a validar os resultados produzidos pela classe para cada um dos **Casos de Testes planejados**.

Como a *Classe ImpostoRenda* recebe como **input** o **valor da renda líquida**, e que este valor possui um ***domínio contínuo***, isto é, podendo variar de **valores negativos** a **valores positivos** dentro de todo o **range** permitido para a **variável renda líquida.**

Não faria sentido utilizar valores de renda liquidas arbitrários e sem critério e em número qualquer para testar as funcionalidades da *Classe ImpostoRenda*. Além de não ser prático esta forma arbitrária, não teríamos certeza de afirmar que o cálculo do imposto de Renda estaria correto para todas as situações. Assim sendo, há necessidade de delimitar um subconjunto de valores de testes adequados que nos garanta de que tenhamos testado o comportamento da Classe Imposto de Renda em todas as suas especificidades.

Logo, para restringir o número de **casos de testes** a aplicar no mínimo indispensável, porém, com garantia de que iremos testar os subconjuntos de valores e assim exercitar o cálculo de valores de todas as ***Faixas de Valores da Tabela Progressiva do Imposto de Renda,*** adotaremos os **Critérios** de **Classes de Equivalência** *combinado com* **Valores Limites** para poder estabelecer os **casos de teste** adequados e em número mínimo necessário (que constam descritos na seção de Estratégia dos Testes).

Adotei **testes automatizados** na realização dos **Testes Funcionais**, e para isso, a ***ferramenta*** usada é o específico para a ***Linguagem de Programação Java***, cujo nome é ***JUnit Versão 4***, que permite realizar **Teste Unitários** sobre a *Classe ImpostoRenda*. O **Conjunto de Testes planejados** foram assim implementados na ***Linguagem Java 8*** com bases nos recursos da ***ferramenta JUNIT 4***. Como utilizei o ***Eclipse IDE*** para desenvolvimento em ***Java*** da *Classe ImpostoRenda e dos* ***Testes Unitários***., foi instalado o plugin do ***JUnit Versão 4***.

O **Conjunto de Testes Unitários** escritos na ***linguagem Java*** e utilizando as ***features da ferramenta JUnit Versão 5*** serão posteriormente usados nos **Testes Estruturais** e no **Testes de Mutação**.

3.2 – Projeto dos Casos de Testes Funcionais (Como será Testado) da *Classe ImpostoRenda*

3.2.1 - Análise das Classes de Equivalência e dos Limites de Valores

No projeto do Conjunto de Testes para realização do Teste Funcional , utilizando a técnica de Teste de Domínio, adotei os Critérios de Classes de Equivalência *combinado com* Valores Limites para poder estabelecer os **casos de teste** adequados e em número mínimo necessário.

Devido ao fato de que **para cálculo do imposto de renda** o Ministério da Fazenda por meio da Secretaria da Receita Federal **estabeleceu** *faixas de valores de renda, e dependendo da faixa de valor será aplicada uma alíquota progressiva para cálculo do imposto a reter da renda da pessoa física assalariada.* Portanto, a **natureza desta aplicação** já estabelece particionamento do valor da renda em faixas, e portanto, enquadra-se como uma luva no **Critério de Classes de Equivalência** na definição dos **Testes de Domínio**, e embasando a adoção deste critério como o adequado para realizar os testes funcionais e de responsabilidade da *Classe ImpostoRenda*.

Outro aspecto é que diante de particionamento de valores, a literatura técnica mostra que as falhas do programador ocorre próximo as limites de valores das classes de equivalência. Assim, devido ao uso inadequado dos operadores de maior, menor, maior ou igual, menor ou igual, e assim por diante, os erros no software ocorre nestes limites de valores entre as classes de equivalência, portanto, a adoção do critério de limites de valores combinado com as classes de equivalência é a **estratégia adequada** para planejar os Casos de Testes Funcionais para a *Classe ImpostoRenda*.

Assim, analisando a Tabela do Imposto de Renda de 202º com as Faixas Progressivas de Valores e identificando as Classes de Equivalência e os Valores Limites a testar da variável Renda Líquida nos faz produzir a seguinte tabela:

Análise das Classes de Equivalência e dos Limites de Valores

VARIÁVEL	CLASSE DE EQUIVALÊNCIA	LIMITES DE VALORES	OBS1	OBS2
renda liquida	[0, 1903,98]	-1 0 1903,98	Nº Negativo Valor Nulo	Faixa 1
	[1903,99, 2.826,65]	1903,99 2.826,65		Faixa 2
	[2.826,66, 3.751,05]	2.826,66 3.751,05		Faixa 3
	[3.751,06, 4.664,68]	3.751,06 4.664,68		Faixa 4
	[4.664,69, infinito]	4.664,69		Faixa 5

Notação de Intervalo: [a, b] significa que os valores extremos a e b estão incluídos no intervalo

Como o valor da renda líquida é um domínio contínuo, não temos classes inválidas, isto é, os valores da renda estarão sempre enquadrados numa classe de equivalência válida. Mesmo no caso de valor negativo, o resultado do imposto de renda calculado será o mesmo para a situação do valor nulo, onde a pessoa é isenta da retenção na fonte do imposto de renda. Logo, não existem classes inválidas a serem testadas nos testes unitários da *Classe ImpostoRenda*.

3.21.2 – Conjunto de Casos de Testes para o Teste Funcional da *Classe ImpostoRenda*

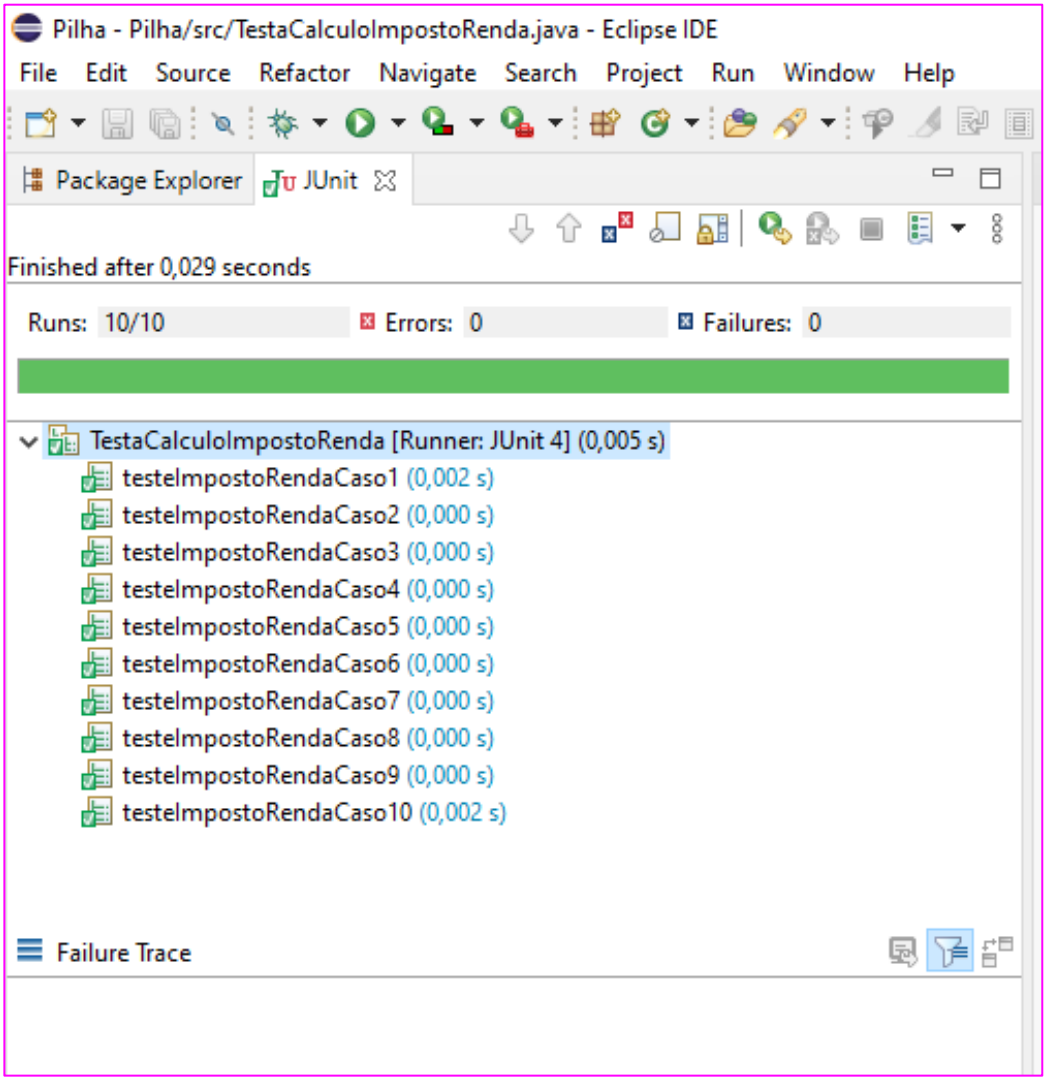
A seguir a Tabela contendo o Conjunto de Casos de Testes Unitários Funcionais da *Classe ImpostoRenda* com base na estratégia e critérios definidos na seção anterior:

Conjunto de Casos de Teste Unitários – Teste Funcional		
Caso de Teste	Renda Líquida	Imposto de Renda
T1	-1	0,00
T2	0	0,00
T3	1903,98	0,00
T4	1903,99	0,00
T5	2.826,65	69,20
T6	2.826,66	69,20
T7	3.751,05	207,86
T8	3.751,06	207,86
T9	4.664,67	413,42
T10	4.664,69	413,43
Nota: Casos de Testes Planejados sem duplicidades		

3.2.3 – Execução dos Casos de Testes para o Teste Funcional (Quando e Como será Testado) da *Classe ImpostoRenda*

Adotei testes automatizados na realização dos Testes Funcionais, e para isso, a *ferramenta* usada é o especifico para a *Linguagem de Programação Java*, cujo nome é *JUnit Versão 5*, que permite realizar Teste Unitários sobre a *Classe ImpostoRenda* . O Conjunto de Testes planejados foram assim implementados na *Linguagem Java 8* com bases nos recursos da *ferramenta JUNIT 4*. Como utilizei o *Eclipse IDE* para desenvolvimento em *Java* da *Classe ImpostoRenda* *e dos Testes Unitários*., foi instalado o plugin do *JUnit Versão 5*.

Na imagem a direita, vemos o ECLIPSE IDE e o log pós processamento no JUnit plugin da execução do Conjunto dos 10 Casos de Testes – como pode-se ver a cor verde é indicativa de que os testes passaram indicando que o código fonte da *Classe ImpostoRenda* está se comportando como deveria. Não há nenhum registro de falha e para cada caso de teste aparece o tempo do processamento.



A Classe que contém o Conjunto de Casos de Testes denomina-se: *TesteCalculoImpostoRenda* . A seguir o **código fonte** do Conjunto de Testes Unitários Funcionais implementado em Java é, respectivamente:

```
import static org.junit.Assert.*;
import org.junit.Test;
public class TestaCalculoImpostoRenda {
	@Test
	public void testeImpostoRendaCaso1() {
		// Teste 1 - Valor da Renda Líquida Negativa
		ImpostoRenda ir = new ImpostoRenda();
		assertEquals(0.0f,(float)(ir.calculaImpostoRenda(-1)),2);
	}
	@Test
	public void testeImpostoRendaCaso2() {
		// Teste Case 2 - Valor da Renda igual a Zero
		ImpostoRenda ir = new ImpostoRenda();
		assertEquals(0.f,(float)(ir.calculaImpostoRenda(0)),2);
	}
	@Test
	public void testeImpostoRendaCaso3() {
		// Teste Case 3 - Valor da Renda menor 1.903,99
		ImpostoRenda ir = new ImpostoRenda();
		assertEquals(0.f,(float)(ir.calculaImpostoRenda(1903.98f)),2);
	}
	@Test
	public void testeImpostoRendaCaso4() {
		// Teste Case 4 - LImite Inferior Faixa 2
		ImpostoRenda ir = new ImpostoRenda();
		assertEquals(0.f,(float)(ir.calculaImpostoRenda(1903.99f)),2);
	}
	@Test
	public void testeImpostoRendaCaso5() {
		// Teste Case 5 - LImite Superior Faixa 2
		ImpostoRenda ir = new ImpostoRenda();
		assertEquals(69.20f,(float)(ir.calculaImpostoRenda(2826.65f)),2);
	}
	@Test
	public void testeImpostoRendaCaso6() {
		// Teste Case 6 - LImite Inferior Faixa 3
		ImpostoRenda ir = new ImpostoRenda();
		assertEquals(69.20f,(float)(ir.calculaImpostoRenda(2826.66f)),2);
	}
	@Test
	public void testeImpostoRendaCaso7() {
		// Teste Case 7 - LImite Superior Faixa 3
		ImpostoRenda ir = new ImpostoRenda();
		assertEquals(207.86f,(float)(ir.calculaImpostoRenda(3751.05f)),2);
	}
	@Test
	public void testeImpostoRendaCaso8() {
		// Teste Case 8 - LImite Inferior Faixa 4
		ImpostoRenda ir = new ImpostoRenda();
		assertEquals(207.86f,(float)(ir.calculaImpostoRenda(3751.06f)),2);
	}
	@Test
	public void testeImpostoRendaCaso9() {
		// Teste Case 9 - LImite Superior Faixa 4
		ImpostoRenda ir = new ImpostoRenda();
		assertEquals(413.42f,(float)(ir.calculaImpostoRenda(4664.67f)),2);
	}
	@Test
	public void testeImpostoRendaCaso10() {
		// Teste Case 10 - LImite Inferior Faixa 5
		ImpostoRenda ir = new ImpostoRenda();
		assertEquals(413.43f,(float)(ir.calculaImpostoRenda(4664.68f)),2);
	}
}
```

Importante deixar registrado que, este mesmo Conjunto de Testes Unitários acima planejados e escritos na ***linguagem Java*** , utilizando as ***features da ferramenta JUnit Versão 4*** serão usados nos Testes Estruturais e no Testes de Mutação.

Seção 4 – Teste Estrutural da *Classe ImpostoRenda*

4.1 – O Porquê do Teste Estrutural da *Classe ImpostoRenda*

Com base na premissa de que nenhum software está livre de falhas, e de que, portanto, durante o processo de implementação do código fonte do meu algoritmo da *Classe ImpostoRenda*, posso ter cometido falhas de diversas naturezas em qualquer dos passos (um ou mais, e/ou muitos) da construção da solução do software.

Mesmo que, os *Testes Funcionais* da minha *classe ImpostoRenda* tenham sido feitos com sucesso, eles não consideraram a *implementação do código fonte*, pois olharam a minha *classe ImpostoRenda* como uma ***caixa preta*** – *apenas testaram os requisitos, responsabilidades e funcionalidades da minha classe* – consideraram apenas sob *o ponto de vista "do que deveria ter sido feito"*. Não consideram como feito feita a implementação do algoritmo de cálculo do imposto de renda. Assim, os *testes Unitários Funcionais* já aplicados na minha *Classe ImpostoRenda* podem ter exercitado apenas parte do meu código, e outros não. Como poderei saber disso? Se houver qualquer parte do código que não tenha sido testada, poderá implicar na existência de erros latentes, e por conseguinte, falhas susceptíveis de acontecer a qualquer momento, bastando apenas que, algum input implique com que a parte não testada possa ser exercitada e assim produzir erro(s).

Logo, a *única maneira que tenho, é me certificar de que, todo o meu código de produção da classe ImpostoRenda, tenha sido executado todos os seus statementes ao menos uma vez por um caso de teste*. Para isso, terei que produzir **casos de testes** que me garantam que *todo fluxo de controle, desvios e blocos de código sejam exercitados*. Fazendo isso, terei certeza de que, justamente as partes do programa, que não tenham ficado sem ser validadas pelo meu *Conjunto de Casos de Testes Unitários* que usei no *Teste Funcional* (aplicado e descrito na seção 3 deste documento) sejam exercitadas. Diante deste contexto, este enfoque de olhar a minha *Classe ImpostoRenda* como uma **caixa branca**, isto é, *sob o posto de vista de como feito feita a implementação da minha solução* para a Classe ImpostoRenda e, assim, ***usar a estrutura de todo o código fonte como base para construir os testes é denominado de Teste Estrutural***. Concluo assim que, os testes estruturais são necessários e complementares aos testes funcionais já realizados na minha *Classe ImpostoRenda*.

4.2 – Grafo do Fluxo de Controle a Usar no Teste Estrutural na *Classe ImpostoRenda*

4.2.1 – Que Critério de Cobertura de Código Adotar no Teste Estrutural

Diversas *técnicas* que *usam o código do programa como meio de guiar o planejamento e a aplicação de casos de testes unitários*, são denominadas de *técnicas de testes estruturais*. A parte do meu código a ser exercitada durante os *testes* diz respeito ao conceito de *cobertura do teste*. Cada uma das diferentes *técnicas de teste estrutural* entende de maneira diferente o *critério de cobertura do código* (*coverage criteria*). O *critério de cobertura do código* está diretamente relacionado com a *cobertura do teste* (*test covergqe*). Assim, a ***cobertura do teste*** significa *a quantidade (percentagem) do código de produção que é exercitado pelo conjunto de casos de testes*. *Logo, a minha meta é ter 100% do código coberto pelos testes de modo a eliminar a presença de erros (apesar de que nunca garantirá a ausência plena de erros).*

Os *critérios de cobertura de código* inclurem: *Line coverage, statement coverage , bloc coverage, branch/decision coverage, condition coverage (basic + condition + branch), path coverage e Modified Condition/Decision coverage.* Cada *critério de cobertura de código* tem as *suas vantagens e desvantagens*, além de que dependendo do critério escolhido o *número de testes a desenvolver irá variar*, alguns critérios precisam de mais casos de testes ao passo que outras menos. Segundo a literatura técnica, *alguns critérios estão implicitamente enquadrados em outros*. Por exemplo, *100% de Branch Converage sempre implica em 100% de Line Coverage. Contudo, 100% de line coverage não implica em 100% de Branch Coverage. Além disso, 100% de Branch + Condition Coverage sempre implica em 100% de Branch Coverage e 100% de Line Coverage*. Esta relação de alguns critérios englobados em outros implica em que o englobado gerar uma cobertura menor do que o englobante. O diagrama ao lado direito, demonstra esta *hierarquia de abrangência da cobertura de código entre estes diversos critérios*. Assim, *usei esta relação como guia para saber que critério adotar para a realização dos Testes Estruturais* da minha *Classe ImpostoRenda*.

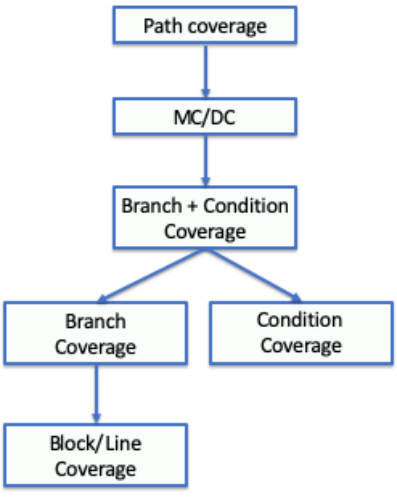


Figura: Relação entre Critérios de Cobertura de Código

4.2 – O Grafo de Fluxo de Controle (GFC) da *Classe ImpostoRenda*

O *Grafo de Fluxo de Controle (CFG)* é uma representação de todos os possíveis caminhos que poderão ser transversos durante a execução do código de produção da minha *Classe ImpostoRenda*. O GFC consiste de *blocos básicos*, *blocos de decisão* e *setas(arestas)* que conectam estes *blocos*. O bloco básico é composto do número máximo de statementes que são executados juntos não importa o que aconteça. Blocos Simples são representados por retângulos. Um bloco de decisão, por outro lado, representa todos os statements do código fonte que podem criar diferentes ramificações (branches) no caminho do fluxo de execução do programa. Os blocos de decisão são representados por losangos. A conexão entre um bloco simples e um bloco de decisão é denominado de aresta (edge). Um bloco básico tem sempre uma única aresta de saída. O bloco de decisão, por outro lado, tem sempre duas arestas de saída (indicando onde ir no caso decisão evoluir para verdadeiro e para onde ir no caso da decisão evoluir para falso). Quando o fluxo evolui para o fim do programa então o Gráfico de fluxo de controle encerra. Abaixo o *GFC* do código que implementa a minha c*lasse ImpostoRenda*.

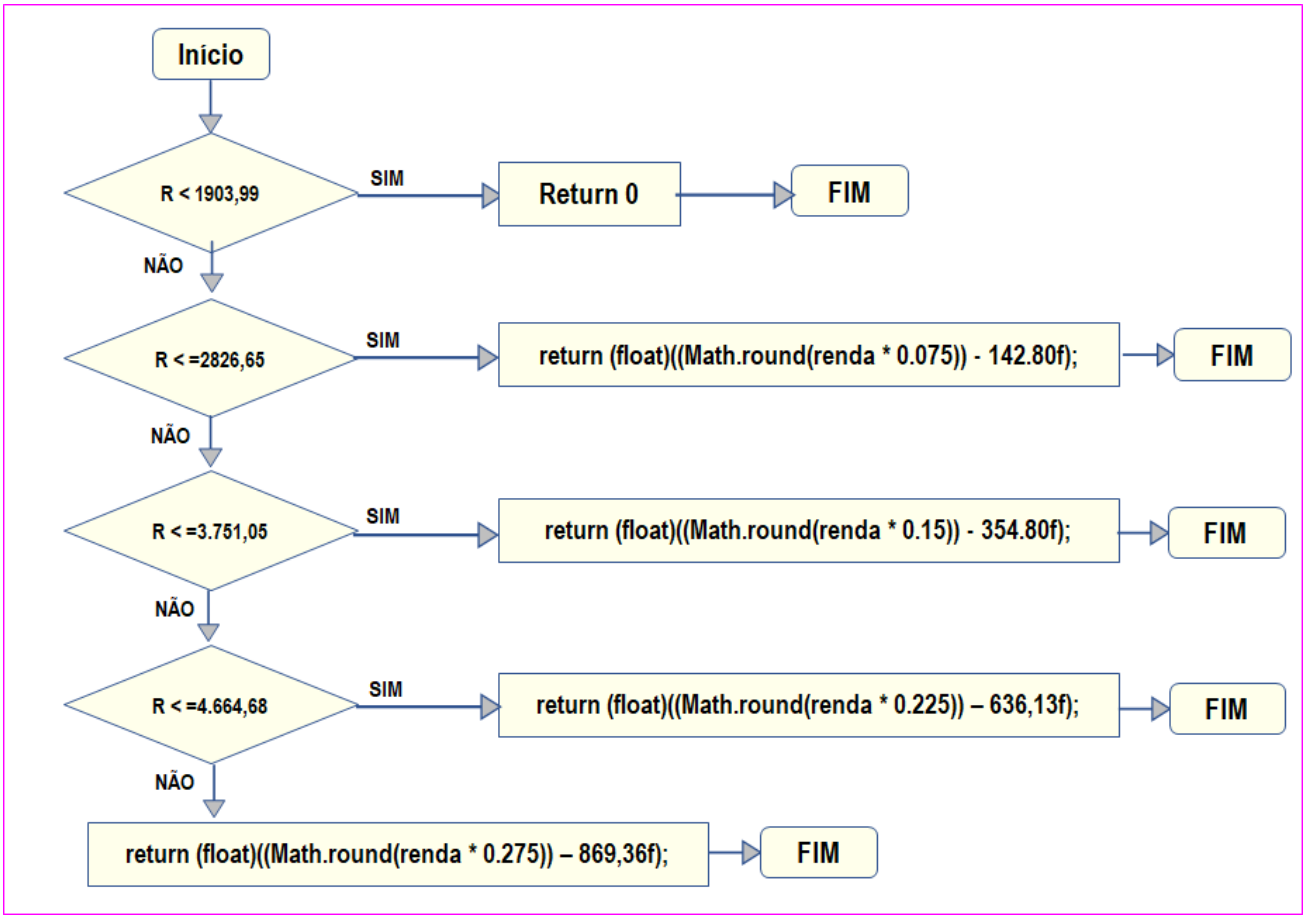


Figura – Grafo de Fluxo do Controle do Algoritmo da Classe ImpostoRenda

O grafo de fluxo de controle (CFG) é um elemento de apoio e auxílio na escolha do critério de cobertura de código a adotar e, também, no planejamento do *conjunto de casos de testes* para realização do Teste Estrutural. Com base no *código fonte escrito* na linguagem Java Versão 8 da minha *Classe ImpostoRenda* o CFG é construído. Importante enfatizar que o *CFG* usa a abstração da estrutura interna da minha *classe ImpostoRenda* representada por um **grafo** para apresentação do código implementado da minha classe..

Diante disso, a minha *Classe ImpostoRenda* é decomposta em um *conjunto de blocos de declarações do código*, de modo que a execução do primeiro elemento dentro do ***bloco*** implica na execução de todos os comandos naquele bloco e na ordem em que aparecem. Todas as ***declarações de código*** (*statements*) no ***bloco***, exceto a primeira, tem um *único predecessor* e exatamente um *sucessor* (exceto a última *declaração de código*). Isto significa que, *não há desvio no fluxo de controle de ou para as declarações de bloco no código no bloco*.

Assim, o *Grafo de Fluxo de Controle* mostrado na figura da página anterior, que representa o algoritmo implementado da minha ***Classe ImpostoRenda,*** estabelece a correspondência entre os NÓS e BLOCOS e indica os possíveis ***desvios no fluxo de controle entre os blocos*** usando as **ARESTAS** do grafo. Por esta razão, *o CFG é um grafo direcionado com uma única entrada e vários NÓS de saída.*

Do CFG *diferentes elementos podem ser selecionados para execução*, caracterizando o *teste estrutural*. Do CFG pude estabelecer os requisitos (como demonstro na seção 4.3.3 abaixo) a serem atendidos pelo meu *Conjunto de Casos de Testes Unitários*, de modo que garanta a execução ao menos uma vez de cada *VERTICE* do grafo de fluxo de controle.

4.3 – Planejamento dos Casos de Teste de Cobertura do Código (GFC) da *Classe ImpostoRenda*

4.3.1 – Que Critério Adotei para o Teste Estrutural

Considerando as especificidades da *implementação do meu algoritmo de cálculo do imposto de renda*, em particular de que o *Grafo de Fluxo de Controle* dele, demonstra a existência da combinação entre *blocos simples* e de *blocos de decisão não compostas*, e *a inexistência de loops*, optei por *Branch + Codition Coverage*. A *definição do conjunto de casos de testes unitários que usarei para garantir 100% de cobertura do código de produção da minha Classe ImpostoRenda.*

4.3.2 – Critério Branch + Condition Converage para Teste Estrutural da *Classe ImpostoRenda*

A *implementação do algoritmo* da minha *classe ImpostoRenda* tem *blocos de condição simples*, onde são testados as *faixa de valores (limite inferior e superior) de modo a tratar os limites superiores da faixa, e como são tratados os valores progressivos das faixas, pude tratar as condições compostas como condições simples, e assim, reduzir o tamanho da implementação, reduzindo o tamanho do código,* de modo a identificar a alíquota a aplicar e o abatimento a deduzir do imposto de renda calculado para cada faixa de valores..

Como *o critério de cobertura de código branch + condition* considera cada *condição e seu desvio individualmente*, se aplica adequadamente para o *planejamento dos testes estruturais* da implementação da minha *classe ImpostoRenda.* Isso foi possível porque:

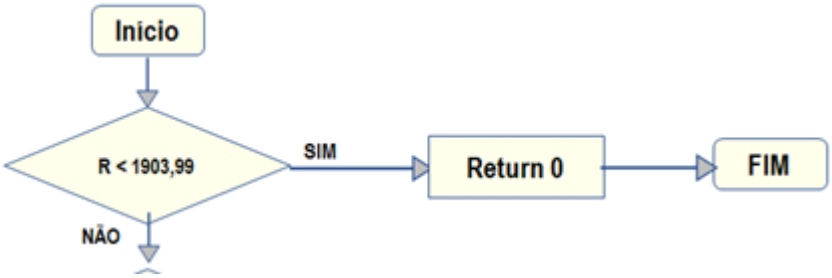
> substitui as ***condições compostas*** **(A >= x && A <= y)** *do meu algoritmo* por ***condições simples*** **(A <= x)**.

Analisando o CFG da implementação do código d minha *Classe Imposto de Renda,* presente na página 10 deste documento, podemos ver que, temos *quatro blocos de decisão com condições simples.* Assim, facilmente e com poucos casos de testes conseguirei obter 100% da *cobertura do código*. Como são *condições simples*, terei sempre 2 *possíveis caminhos para cada condição*. Portanto, oito casos de testes serão suficientes para testar todos os *possíveis caminhos no CFG como saída dos quatro blocos de decisão*, como demostro na seção a seguir.

4.3.3 – Desenvolvimento dos Casos de Testes do Teste Estrutural da *Classe ImpostoRenda*

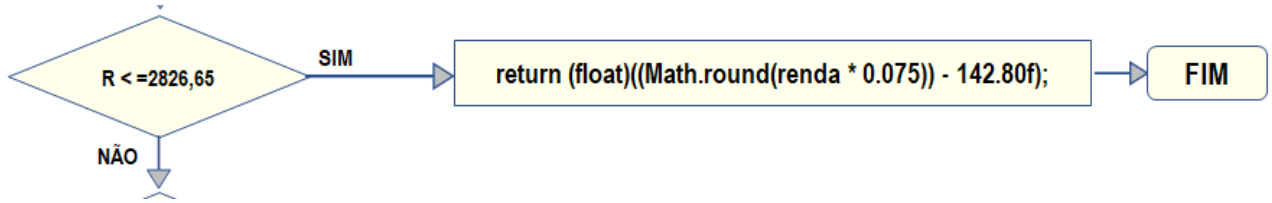
Analisando as partes do CFG da implementação da minha classe de cálculo do imposto de renda, e considerando a adoção do critério de *Branch + Codition Coverage* para montar as *tabelas de decisão*, e poder identificar os *diferentes caminhos no fluxo de controle,* considerando as combinações e os correspondentes *casos de testes* a utilizar:

Passo 1 – Bloco 1 - Decisão Simples



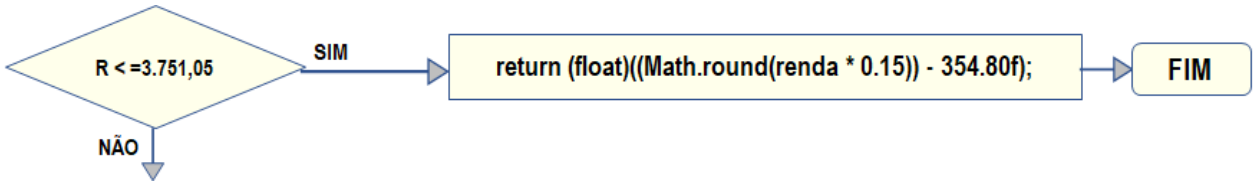
Bloco 1 - Condição Inicial - CFG da Classe ImpostoRenda			
	CONDIÇÃO		
Teste	Renda <= 1903,99	Valor de Input	Resultante
1	Verdadeiro	1903,98	0
2	Falso	1903,99	0

Passo 2 – Bloco 2 - Decisão Simples



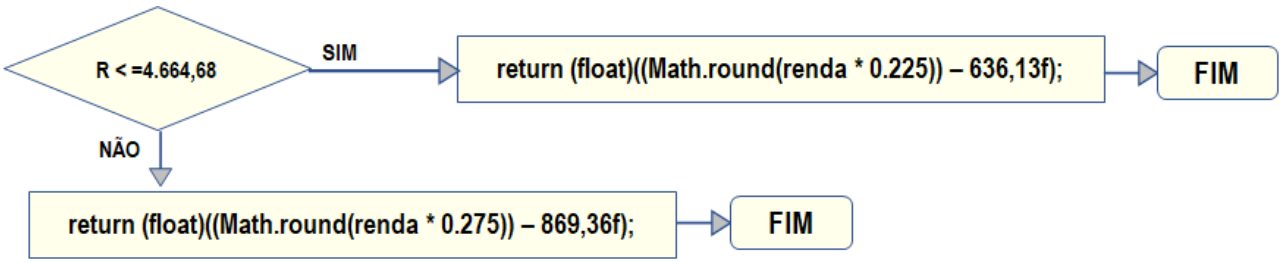
Bloco 2 - Segunda Codnição - CFG da Classe ImpostoRenda			
	CONDIÇÃO		
Teste	Renda <= 2826,65	Valor de Input	Resultante
3	Verdadeiro	2826,65	69,20
4	Falso	2826,66	69,20

Passo 3 – Bloco 3 - Decisão Simples



Bloco 3 - Terceira Condição - CFG da Classe ImpostoRenda			
	CONDIÇÃO		
Teste	Renda <= 3751,05	Valor de Input	Resultante
5	Verdadeiro	3571,05	180,86
6	Falso	3571,06	167,33

Passo 4 – Bloco 3 – Decisão Simples



Bloco 4 - Quarta Condição - CFG da Classe ImpostoRenda			
	CONDIÇÃO		
Teste	Renda <= 4.664,68	Valor de Input	Resultante
7	Verdadeiro	4664,68	413,42
8	Falso	4664,69	413,43

Diante do exposto, os oito casos de testes que precisam ser executados para exercitar todas as partes (100%), sem exceção, do código de produção implementado no meu algoritmo escrito em Java da minha ***Classe Imposto de Renda***, são respectivamente:

Conjunto de Casos de Teste Unitários - Teste Estrutural		
Caso de Teste	Renda Líquida	Imposto de Renda
T1	1903,98	0,00
T2	1903,99	0,00
T3	2.826,65	69,20
T4	2.826,66	69,20
T5	3.751,05	207,86
T6	3.751,06	207,86
T7	4.664,68	413,42
T8	4.664,69	413,43
Nota: Casos de Testes Planejados para Cobertura de 100% do Código		

Abaixo apresento o *Conjunto de Casos de Testes Unitários* Planejados e Executados para o *TESTE FUNCIONAL* da ***Classe ImpostoRenda:***

Conjunto de Casos de Teste Unitários – Teste Funcional		
Caso de Teste	Renda Líquida	Imposto de Renda
T1	-1	0,00
T2	0	0,00
T3	1903,98	0,00
T4	1903,99	0,00
T5	2.826,65	69,20
T6	2.826,66	69,20
T7	3.751,05	207,86
T8	3.751,06	207,86
T9	4.664,68	413,42
T10	4.664,69	413,43

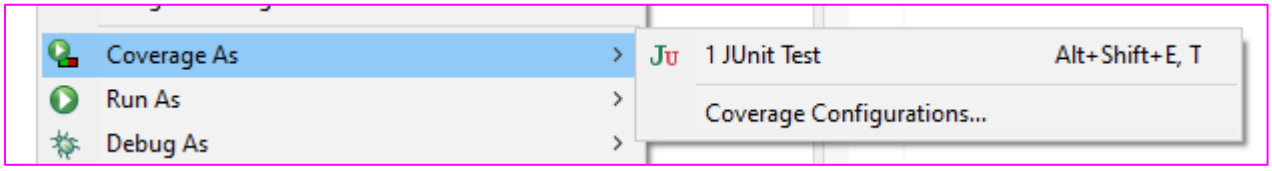
Comparando os *Casos de Teste* projetados para o *Teste Funcional* com aqueles do *Teste Estrutural*, dá para inferir que, os *Casos de Testes do Teste Estrutural* são 2 casos a menos do que os *Casos de Teste do Teste Funcional*. Porém, os *casos de testes são idênticos para os 8 casos de testes que são comuns entre ambas as técnicas de teste de software*. Desta forma, *é possível aplicar o mesmo Conjunto de Casos de Testes presentes na* ***Classe TestaImpostoRenda*** para realizar os *Testes Estruturais na* ***Classe ImpostoRenda***. A página 8 deste documento consta o *código fonte* escrito em Java da ***Classe TestaImpostoRenda*** contendo o *Conjunto de Casos de Testes Unitários* feitos para rodar na *ferramenta JUnit 4*.

4.4 – Automação do Teste Estrutural da *Classe ImpostoRenda*

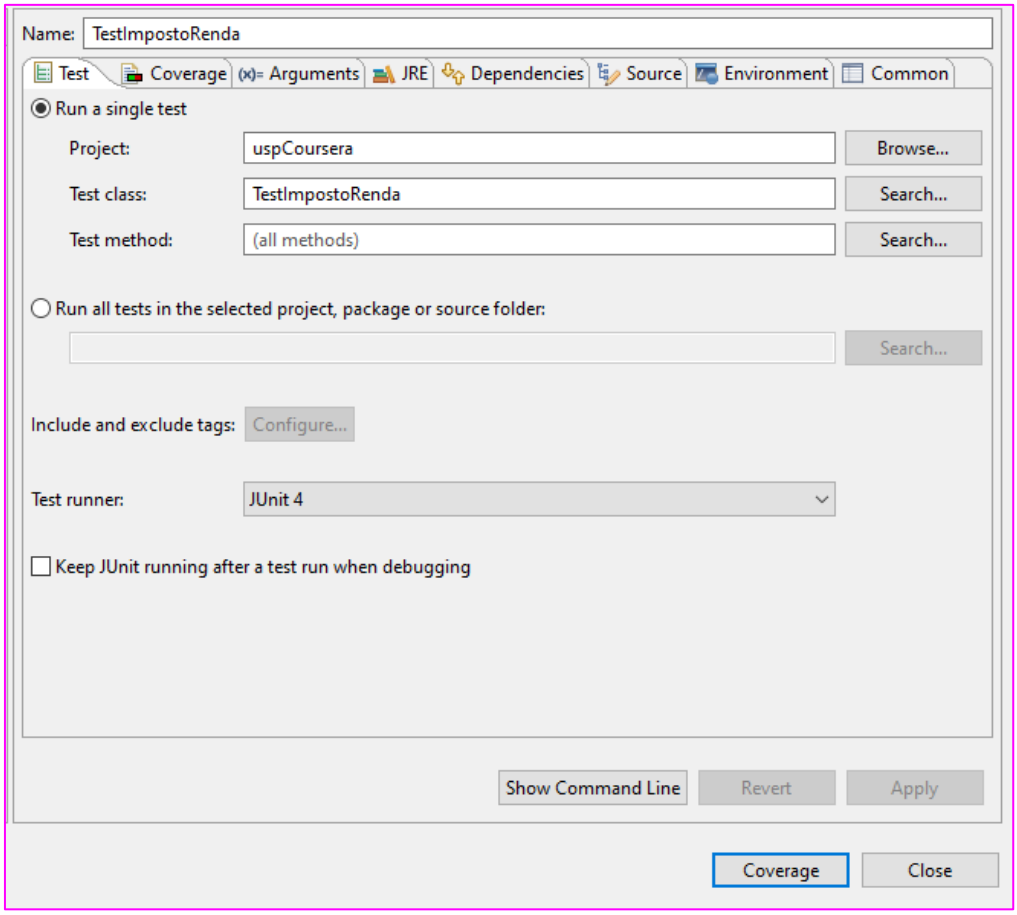
Para automação do *Teste Estrutural* da minha *Classe ImpostoRenda* adotei os recursos presentes na *ferramenta JUnit versão 4*, plugin que instalei no meu *Integrated Development Environment (IDE) ECLIPSE 2020-09*. Esta opçãopor esta ferramenta levou em que conta que, os meus testes unitários foram projetados com base nesta mesma ferramenta.

Fora isso, o *teste de cobertura do código* realizada pelo *JUnit* roda obre o *código de produção*, isto é, sobre *o bytecode compilado da minha classe ImpostoRenda*, para execução dos *Testes Estruturais*, usando o *Conjunto de Casos de Testes* projetados e presentes na *Classe TestaImpostoRenda*.

Para ativação do Teste de Cobertura do Código usando o Junit 4, basta selecionar a Classe de Teste e com o botão direito ativar o pull-down menu e selecionar a opção “Coverage as” e ao abrir o menu secundário selecionar o JUnit Test, como mostrado no printscreen a seguir:

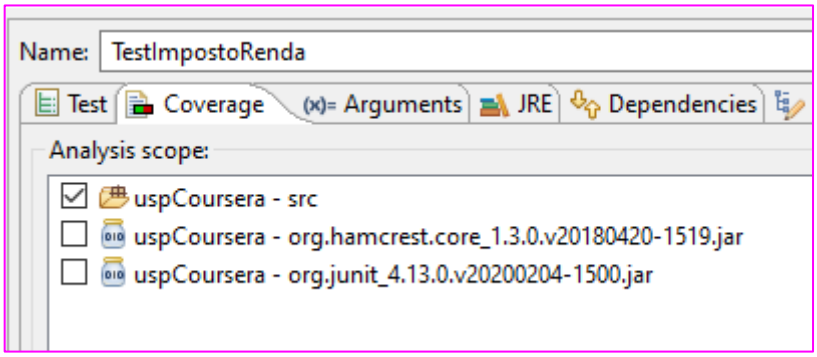


Antes da execução do Teste Estrutural de Cobertura de Código é necessário rever as configurações do Coverage Test.



A seguir a imagem abaixo mostra a parte de *Configuração do Coverage Test* da minha *classe de TestImpostoRenda* a ser utilizada no *Teste Estrutural de Cobertura do Código* da *Classe ImpostoRenda,* que está contida no

"*package default*" no *diretório* "*uspCoursera*" e *subdiretório* "*src*", onde o *JUnit 4* irá localizar a *classe Java* onde está o *Conjunto de Casos de Testes Unitários* a usar no *Teste Estrutural:*



Após rodar o *Teste Estrutural de Cobertura de Código* no *ECLIPSE 2020-09* usando o *JUnit 4*, é mostrada a *situação final do processamento*, e indicado por meio da **cor verde,** no *código fonte* da ***classe ImpostoRenda*** **as partes do código que foram exercitadas pelo *conjunto de Casos de Testes Unitários*** projetados e presente na ***Classe TestImpostoRenda***. A *parte do código não exercitado* é mostrado na **cor amarela**.

ImpostoRenda.java | TestImpostoRenda.java

```
1
2
3  public class ImpostoRenda {
4
5      public float calculaImpostoRenda(float renda) {
6
7          if (renda < 1903.99f) {
8          // Faixa 1 - Menor do que 1.903.99 Isento de Imposto de Renda
9              return 0.0f;
10         }
11         if (renda <= 2826.65f) {
12         // Faixa 2 - de 1.903,99 até 2.826,65 Taxa 7,5% Dedução 142,80
13             return (float)((Math.round(renda * 0.075)) - 142.80f);
14         }
15         if (renda <= 3751.05f) {
16         // Faixa 3 - de 2.826,66 até 3.751,05 Taxa 15% Dedução 354,80
17             return (float)((Math.round(renda * 0.15f)) - 354.80f);
18         }
19         if (renda <= 4664.68f) {
20         // Faixa 4 - de 3.751,06 até 4.664,68 Taxa 22,5% Dedução 636,13
21             return (float)((Math.round(renda * 0.225f)) - 636.13f);
22         }  else  {
23         // Faixa 5 - Acima de 4.664,68 Taxa 27,5% Dedução 869,36
24             return (float)((Math.round(renda * 0.275f)) - 869.36f);
25         }
26     }
27 }
```

Assim, pela indicação dos statementes do programa em cor verde (na imagem acima), dá para concluir que, o ***Teste Estrutural de Cobertura de Código*** *da minha* ***classe ImpostoRenda*** *foi de 100%.*

4.5 – Relatório da Automação do Teste Estrutural de Cobertura de Código da *Classe ImpostoRenda*

A ferramenta *JUnit 4,* instalada no *IDE ECLIPSE 2020-09,* fornece um *Relatório do Resultado da automação dos Testes Estruturais de Cobertura de Código* como mostrado abaixo, e indica explicitamente o *percentual da cobertura de código* conseguida após rodar o *Conjunto de Casos de Testes Unitários* sobre a *Classe ImpostoRenda*, respectivamente:

TestImpostoRenda (17 de out de 2020 08:19:11)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
uspCoursera	100,0 %	167	0	167
src	100,0 %	167	0	167
(default package)	100,0 %	167	0	167
ImpostoRenda.java	100,0 %	54	0	54
ImpostoRenda	100,0 %	54	0	54
TestImpostoRenda.java	100,0 %	113	0	113
TestImpostoRenda	100,0 %	113	0	113

No ***Relatório de Teste Unitário de Cobertura de Código*** da minha *Classe ImpostoRenda* acima temos:

a) O lado esquerdo mostra a *estrutura do folder do projeto uspCoursera* , o package "default", e as classes Java Imposto-Renda e TestImpostoRenda, e o código compilado de cada uma delas (tem uma letra C precedendo o nome da classe);

b) A segunda coluna consta o *indicador de Cobertura de Código*, resultado do cálculo do ***número total* de *bytecodes* exercitados** do ***código de produção*** dividido pelo ***número total de bytecodes do código compilado*** da classe – assim, temos que 100% do código foi exercitado no teste estrutural;

c) A terceira coluna de Covered Instructions – *total de instruções (bytecode* da classe compilada) *exercitadas no processamento do Teste Estrutural de Cobertura de Código* – assim **no meu caso 100%;**

d) A quarta coluna de Missed Instructions – *total de instruções (bytecode* da classe compilada) *não exercitadas no processamento do Teste Estrutural de Cobertura de Código* – isto é, o quanto do código da classe não teve cobertura de código e, portanto, não foi executado durante o testes – assim **no meu caso 0%;**

e) A quinta coluna de Total Instructions – *total de instruções (bytecode* da classe compilada) *que forma o a classe* – isto é, para a cobertura de código completa este número deve ser o mesmo contido na 3ª coluna – assim **no meu caso 100%;**

TestImpostoRenda (17 de out de 2020 08:19:11)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
uspCoursera	100,0 %	167	0	167
src	100,0 %	167	0	167
(default package)	100,0 %	167	0	167
ImpostoRenda.java	100,0 %	54	0	54
ImpostoRenda	100,0 %	54	0	54
calculaImpostoRenda(float)	100,0 %	51	0	51
TestImpostoRenda.java	100,0 %	113	0	113
TestImpostoRenda	100,0 %	113	0	113
testImpostoRendaCaso1()	100,0 %	11	0	11
testImpostoRendaCaso10()	100,0 %	11	0	11
testImpostoRendaCaso2()	100,0 %	11	0	11
testImpostoRendaCaso3()	100,0 %	11	0	11
testImpostoRendaCaso4()	100,0 %	11	0	11
testImpostoRendaCaso5()	100,0 %	11	0	11
testImpostoRendaCaso6()	100,0 %	11	0	11
testImpostoRendaCaso7()	100,0 %	11	0	11
testImpostoRendaCaso8()	100,0 %	11	0	11
testImpostoRendaCaso9()	100,0 %	11	0	11

Acima o zoom do Relatório de Cobertura pelos Métodos executados das Classes no Teste Estrutural de Cobertura de Código – demonstrando 100% de cobertura de código com os testes planejados.

Abaixo o ***JUnit*** mostra 100% da cobertura do código da *Classe de TesteImpostoRenda*, mostrando que partes do código de testes foram realizados. Como pode ser visto todas os statements da Classe estão em verde:

ImpostoRenda.java | TestImpostoRenda.java

```
1  import static org.junit.Assert.*;
4
5  public class TestImpostoRenda {
6
7      @Test
8      public void testImpostoRendaCaso1() {
9          // Teste 1 - Valor da Renda Líquida Negativa
10         ImpostoRenda ir = new ImpostoRenda();
11         assertEquals(0.0f,(float)(ir.calculaImpostoRenda(-1)),2);
12     }
13     @Test
14     public void testImpostoRendaCaso2() {
15         // Teste Case 2 - Valor da Renda igual a Zero
16         ImpostoRenda ir = new ImpostoRenda();
17         assertEquals(0.f,(float)(ir.calculaImpostoRenda(0)),2);
18     }
19     @Test
20     public void testImpostoRendaCaso3() {
21         // Teste Case 3 - Valor da Renda menor 1.903,99
22         ImpostoRenda ir = new ImpostoRenda();
23         assertEquals(0.f,(float)(ir.calculaImpostoRenda(1903.98f)),2);
24     }
25     @Test
26     public void testImpostoRendaCaso4() {
27         // Teste Case 4 - LImite Inferior Faixa 2
28         ImpostoRenda ir = new ImpostoRenda();
29         assertEquals(0.f,(float)(ir.calculaImpostoRenda(1903.99f)),2);
30     }
31     @Test
32     public void testImpostoRendaCaso5() {
33         // Teste Case 5 - LImite Superior Faixa 2
34         ImpostoRenda ir = new ImpostoRenda();
35         assertEquals(69.20f,(float)(ir.calculaImpostoRenda(2826.65f)),2);
36     }
37     @Test
38     public void testImpostoRendaCaso6() {
39         // Teste Case 6 - LImite Inferior Faixa 3
40         ImpostoRenda ir = new ImpostoRenda();
41         assertEquals(69.20f,(float)(ir.calculaImpostoRenda(2826.66f)),2);
42     }
43     @Test
44     public void testImpostoRendaCaso7() {
45         // Teste Case 7 - LImite Superior Faixa 3
46         ImpostoRenda ir = new ImpostoRenda();
47         assertEquals(207.86f,(float)(ir.calculaImpostoRenda(3751.05f)),2);
48     }
49     @Test
50     public void testImpostoRendaCaso8() {
51         // Teste Case 8 - LImite Inferior Faixa 4
52         ImpostoRenda ir = new ImpostoRenda();
53         assertEquals(207.86f,(float)(ir.calculaImpostoRenda(3751.06f)),2);
54     }
55     @Test
56     public void testImpostoRendaCaso9() {
57         // Teste Case 9 - LImite Superior Faixa 4
58         ImpostoRenda ir = new ImpostoRenda();
59         assertEquals(413.42f,(float)(ir.calculaImpostoRenda(4664.67f)),2);
60     }
61     @Test
62     public void testImpostoRendaCaso10() {
63         // Teste Case 10 - LImite Inferior Faixa 5
64         ImpostoRenda ir = new ImpostoRenda();
65         assertEquals(413.43f,(float)(ir.calculaImpostoRenda(4664.69f)),2);
66     }
67 }
```

Seção 5 - Teste de Mutação (Como será Testado) da *Classe ImpostoRenda*

5.1 – Finalidade do Teste de Mutação

Como sei se o meu ***Conjunto de Casos de Testes Unitários*** planejados e implementos por meio da *Classe TestaCalculoImpostoRenda* tem qualidade? A resposta a esta pergunta será obtida por meio dos Testes de Mutação. O conceito do Teste de Mutação é muito simples. Inicia-se com **Falhas** (ou **mutações**) que são automaticamente inseridas no código do programa fonte da minha *Classse ImpostoRenda*. Após isso, cada um dos terstes do Conjunto de Casos de Testes Unitários da minha *Classe TestImpostoRenda* são aplicados sobre cada uma de todas as mutações geradas do meu código fonte da *Classe ImpostoRenda*. Se o um **teste falha** então a **mutação** é **morta**, se o um teste passa então a **mutação** está **viva**. A qualidade dos meus testes pode então ser medidos a partir do percentual de mutações mortas. A seguir o diagrama de contexto de Execução dos Testes de Mutação.

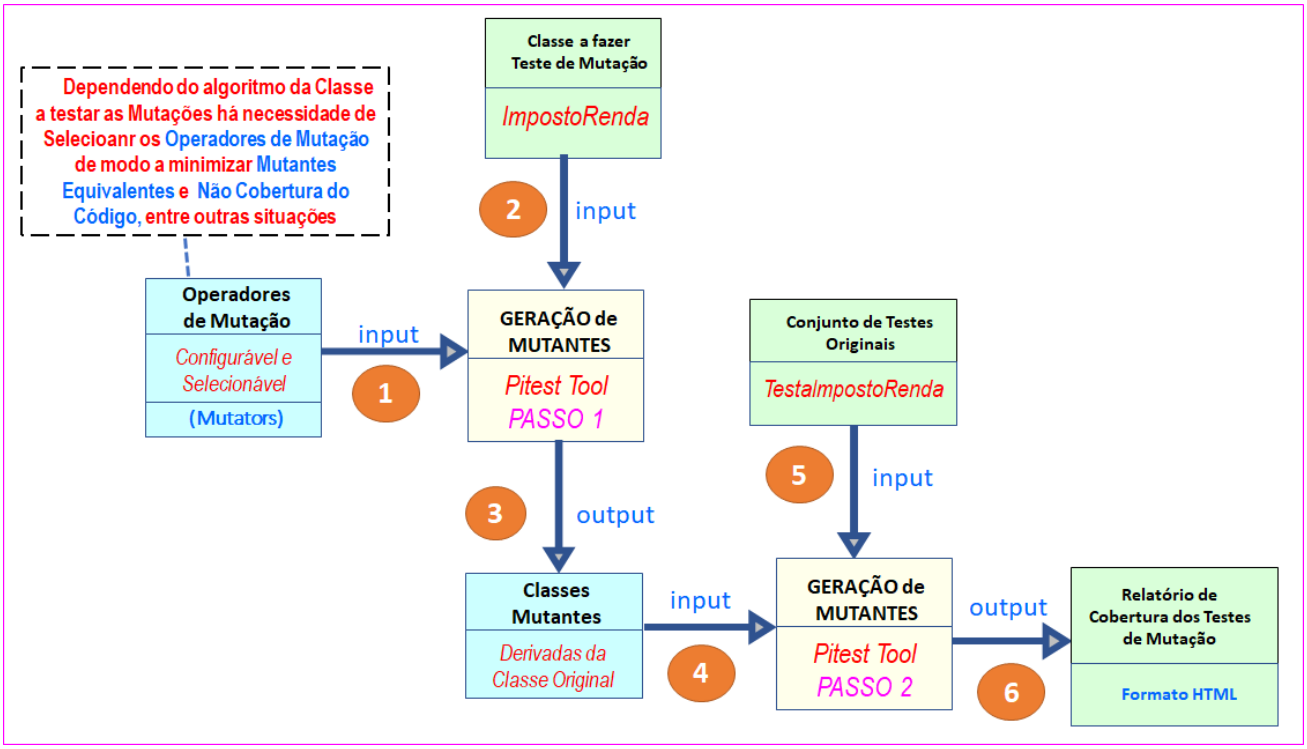


Figura: Fluxo do Ciclo de Execução dos Testes de Mutação

5.2 – Ferramenta Utilizada para Realizar o Teste de Mutação

Para realizar os testes de mutação da *Classe ImpostoRenda*, utilizei a **ferramenta PIT Teste** (*Parallel Isolated Test*). Esta ferramenta fornece um **teste de cobertura para Java** e a **Java Virtual Machine (JVM).** Segundo a literatura técnica é atualmente a melhor ferramenta para isso. É rápida, escalável e integra as plataformas modernas de teste e de build das aplicações em Java.

Esta **ferramenta PIT** executou os meus **casos de testes unitários** contra as **versões modificadas do meu código fonte** da *Classe ImpostoRenda.* Quando o **código fonte da aplicação** é modificado poderá produzir resultados diferentes e fazer com que os **casos de testes falhem**. Se um **teste unitário não falhar** é um indicativo de **problema** no meu **Conjunto de Testes**.

Os **Testes Estruturais** de **Cobertura de Código** (*variantes como: de linha, de desvios, de comandos, etc.)* medem somente que partes do meu código fonte são exercitados pelos meus testes. Ele não verifica, no entanto, se os meus casos de testes estão habilitados a detectar falhas no código executado. Sendo por esta razão que os **Testes Estruturais de Cobertura de Código** conseguem somente identificar a **parte do código fonte** que não foi definitivamente testado.

Existem outros sistemas de teste de mutação para Java, como Jester e Jumble, porém eles não são amplamente utilizados como o PIT Teste. Além disso, são lentos, difíceis de usar e foram escritos para o mundo de pesquisadores, em vez do mundo real das equipes de desenvolvimento de software. Fora isso, o PIT é fácil de usar, está ativamente em desenvolvimento e ativamente tendo suporte e manutenção para corrigir bugs e falhas nele.

Os **relatórios** produzidos pelo PIT Teste de Mutação estão num formato fácil de ler e combina informação de **Cobertura de Código** e de **cobertura de mutação**. Ele utiliza ***quatro cores na Linha de Código do Programa*** para indicar:

- Cor Verde Clara = Mostra Cobertura de Linha;
- Cor Verde Escura = mostra Cobertura de Mutação;
- Cor Rosa Claro = falta cobertura de linha e
- Cor Rosa Escuro = mostra falta de Cobertura de Mutação.

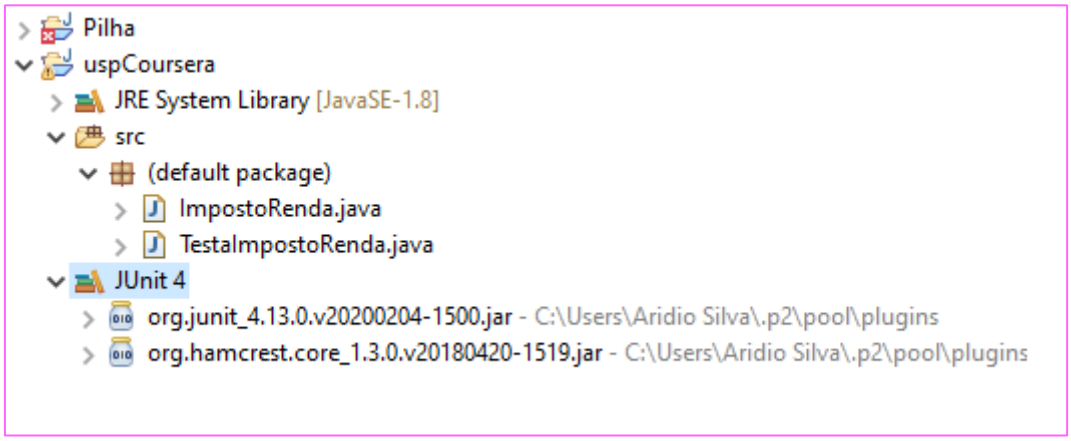
Para utilizar a **Ferramenta PIT Teste Versãoo 1.4.11** precisei instalar o **Plugin Pitclipse 2.1.0** no meu **ECLIPSE IDE** e assim habilitar que a **Ferramenta PIT teste** pudesse rodar as **mutações** contra o **Conjunto de Casos de Testes Unitários** que desenvolvi em **JUnit Versão 4.**

Para que seja possível rodar o PIT Test ele requer obrigatoriamente que esteja presente no bytecode da classe compilada a seguinte informação de depuração: Número das Linhas e Nome do Arquivo do Código Fonte. Assim se o seu Build System não tiver esta opção por "default" deve ativar – foi o que fiz no ECLIPSE IDE 2020-09. Fora isso o JUnit deve ter a versão 4.6 ou superior.

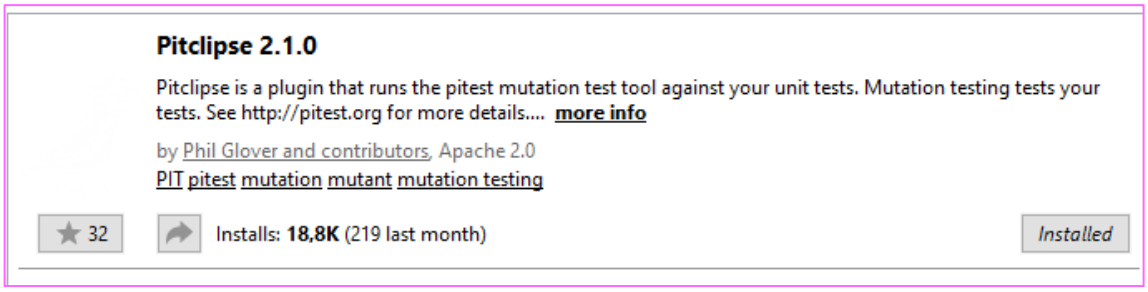
5.3 – Execução do Teste de Mutação na *Classe ImpostoRenda*

5.3.1 – Preparativos para Viabilizar a Execução dos Testes de Mutação

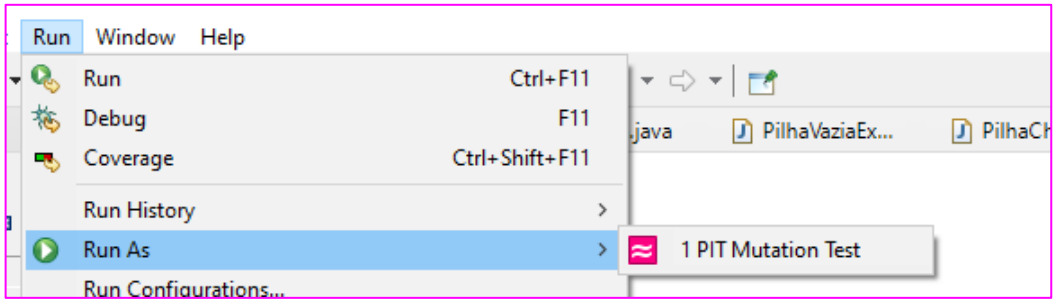
O projeto criado por mim para a realização desta Tarefa Final do Curso de Introdução ao Teste de Software da plataforma Coursera/USP é a mostrada na estrutura abaixo. Onde t podem ser visto dois programas fontes Java, sendo ao primeiro a minha *Classe ImpostoRenda* e o segundo o *Conjunto de Testes Unitários* representados pela *classe TestaImpostoRenda*. Pode ser visto que estão sendo compilados na versão 8 da linguagem Java e estou usando a ferramenta JUNit versão 4 para a realização dos testes Unitários. Esta estrutura será utilizada pela ferramenta PItest para a exceução dos Testes de Mutação.



Para utilizar a ***Ferramenta PITest Versão 1.4.11*** precisei instalar o ***Plugin Pitclipse 2.1.0*** na minha plataforma de desenvolvimento de projetos Java denominada ***ECLIPSE IDE 2020-09,*** e assim, habilitar para que a **Ferramenta PITest** pudesse rodar as **mutações** contra o **Conjunto de Casos de Testes Unitários** que desenvolvi em **JUnit Versão 4.**



Após instalado o PITeste no meu Eclipse 2020-09 IDE bastou selecionar a opção "run as" como mostrado na imagem abaixo:



No entanto, houve necessidade de configurar o arquivo de metadado do projeto de modo a reconhecer a Ferramenta PIT teste e para isso precisei inserir o seguinte fragmento em xml:

```
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>LATEST</version>
 </plugin>
```

Infelizmente a ferramenta PITest não consegue identificar automaticamente o plugin de testes sendo utilizado, e portanto, não reconhece as Classes de Testes, e acaba criando mutações até para o Conjunto de Casos de Testes. Diante disso, é necessário editar diretamente o arquivo de metadados no padrão xml do projeto ".project" para inserir a configuração do plugin JUnit, como indiquei abaixo:

```
<testPlugin>
    <value>junit5</value>
</testPlugin>
```

5.3.2 – Limitando os Testes de Mutação sobre a Minha Classe e ao meu Conjunto de Testes

Por padrão, o *Pitest* irá mutar toda a *base de código* que existir no meu *projeto*. Assim, para *limitar que código ele deverá produzir as mutações e que testes serão executados* precisei configurar o *arquivo xml do projeto* usando os parâmetros de configuração targetClasses e targetTests. Abaixo o *fragmento xml da configuração* necessária que precisei fazer para poder delimitar o *escopo do Pitest* a minha *Classe ImpostoRenda* e ao meu *Conjunto de Casos de Teste* da *classe TestaCalculoImpostoRenda:*

```
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>LATEST</version>
    <configuration>
```

```
            <targetClasses>
                <param>ImpostoRenda*</param>
            </targetClasses>
            <targetTests>
                <param>TestaImpostoRenda*</param>
            </targetTests>
        </configuration>
    </plugin>
```

Detalhe importante é que *a Ferramenta PIT Test quando ativado realiza os testes de mutação sobre todos os programas fontes e testes unitários* existentes no código base da aplicação inteiro. E gera um ***relatório completo*** sobre todas as ***Classes*** contra os respectivos ***Conjuntos de Casos de Testes*** que tenha encontrado.

Para cada ***mutação*** gerada pela ferramenta PIT Teste em cima da minha *Classe ImpstoRenda* ele indicará as seguintes informações após o processamento do *Teste de Mutação*:

- ***Killed*** *– São mutações que foram mortas porque foram identificadas pelos Casos de Testes que falharam;*
- ***Lived*** *ou* ***Survived*** *- São mutações que continuam vivas porque os casos de testes passaram, indicando assim problemas nos Testes planejados, pois não pegaram erros;*
- ***No Coverage*** *– O mesmo que* ***Lived*** *exceto que não encontrou testes que exercitaram a linha de código onde as mutações foram criadas;*

5.3.3 – Operadores de Mutação Utilizáveis e Utilizados do PIT Teste

A ferramenta PIT Teste aplica um *conjunto configurável* de *Operadores de Mutação* (ou *Mutators*) no bytecode gerado pelo meu código fonte compilado. A ferramenta de Mutação PIT Teste define um *número de operações* que mutará o *bytecode* de várias maneiras, inclusive removendo chamadas de métodos, inversão de declarações lógicas, alterando os valores de retorno, e muito mais. O total de *operadores de Mutação* disponibilizados pela *ferramenta PI Teste Versão 1.4.11 atual* são ***27 (vinte e sete)*** e classificados em três *grupos*, respectivamente:

I - Operadores de Mutação Default:

- ***Conditionals Boundary*** – Este operador de mutação substitui o operador relacional < por <= e o <= por < e p > por >= e o >= por >;
- **Increments** – Este operador de mutação modificará os incrementos, decrementos e atribuições de incrementos e decrementos por variáveis locais. Ele substituirá os incrementos por decrementos e vice e versa;
- ***Invert Negatives*** – Este operador de mutação inverterá a negação do inteiro por números de ponto flutuante.
- ***Math*** – Este operador de mutação substitui operações de aritmética por aritmética de ponto flutuante e aritmética de inteiro por outra operação. As substituições será de + por – e de – por _ e de * por / e de / por * e % por * e & por| e ^por & e << por >> e >> por << e >>> por <<.
- ***Negate Conditionals*** – Este operador de mutação modificará todas as condicionais encontradas de acordo com o seguinte critério: == por != e != por == e <= por > e >= por < e < por >= e > por <=;
- ***Return Values*** – Este operador de mutação foi substituído pelos operadores Empty returns, False returns, True returns, Null returns and Primitive returns;
- ***Void Method Calls*** – Este operador de mutação remove as chamadas de métodos void;
- ***Empty returns*** – Este operador de mutação retorna valores com valor 'empty' dependendo do tipo da variável. Assim, String por "", Optional por Optional.empty(), Collection por Collection, emptyList(), Set por Collections.emptySet(), integer por 0, Short por 0, Long por 0, Float por 0 e Double por 0;
- ***False Returns*** – Este operador de mutação substitui valores de retornos primitivos e booleanos por false.

- *True returns* – Este operador de mutação substitui valores de retornos primitivos e booleanos por True.
- *Null returns* – Este operador de mutação substitui valores de retornos com null.
- *Primitive returns* – Este operador de mutação substitui valores de retornos dos tipos int, short, long, char, float e double com 0.

II - Operadores de Mutação Opcionais:

- *Constructor Call* – Este operador de mutação substitui chamadas de construtores de classes por valores null;
- *Remove Conditionals* – Este operador de mutação remove todos os statements condicionais de tal forma que o bloco de código da condição seja sempre executado. Importante habilitar este operador se desejamos que o conjunto de testes tenha cobertura total de código dos statements condicionais. Há ainda disponíveis as seguintes especializações deste operador de mutação para :
    a. *REMOVE-CONDITIONALS_EQ_IF*
    b. *REMOVE-CONDITIONALS_EQ_ELSE*
    c. *REMOVE-CONDITIONALS_ORD_IF*
    d. *REMOVE-CONDITIONALS_ORD_ELSE*
- *Inline Constant* – Este operador de mutação
- *Non Void Method Calls* – Este operador de mutação remove chamadas de métodos de classes por métodos non void. O valor de retorno é substituído pelo Valor Default Java para o tipo específico. Para boolean retorna false, para int /byte/shory/long retorna 0, para float/double retorna 0.0, para char retorna '\u----', para object retorna null.
- *Remove Increments* – Este operador de mutação remove incrementos em variáveis locais;

III - Operadores de Mutação Experimentais:

- *Experimental Argument Propagation* – Este operador de mutação substitui chamadas de métodos com um dos seus parâmetros do tipo correspondente;
- *Experimental Big Integer* – Este operador de mutação troca grandes métodos que retornam inteiros;
- *Experimental Member Variable* – Este operador de mutação modifica classes pela remoção de atribuições variáveis membros. Este operador pode até mesmo remover atribuições por final members. Os membros podem ser inicializados com seus valores default Java para tipos específicos como: booleano com false, int/byte/short/long por 0, fload/double por 0.0, char por '\u0000', Object com null;
- *Experimental Naked Receiver* – Este operador de mutação substitui chamadas de métodos por um naked receiver;
- *Experimental Switch* – Este operador de mutação busca pelo primeiro lable dentro do statement switch que difere do label default, e em seguida modifica o statemente switch pela substituição do label default (sempre que ele for usado) com este label. Todos os outros labels são substituídos pelo default;
- *Negation* – Este operador de mutação substitui o uso de variável numérica (variável local, field, célula de array) com sua negação;
- *Arithmetic Operator Replacement* – Este operador de mutação tal como o operador de mutação math, substitui operações aritmética binária por outra operação de aritmética de ponto flutuante ou de aritmética de inteiros. Este operador de muta~~ao é composto de 4 suboperadores que modificam operadores de acordo com uma tabela; + por -, *, /, % e - por +,*,/,% e * por /,%,+,- e / por *,%,+,- e % por *, /,+,-;
- *Arithmetic Operator Deletion* – Este operador de mutação substitui uma operação aritmética por uma de um dos seus membros. Este operador de mutação é composto de dois suboperadores, que alteram a operação pra o seu primeiro e segundo membros;
- *Constant Replacement* – Este operador de mutação modifica constantes inline;

- *Unary Operator Insertion* – Este operador de mutação insere uma operação unária (incremento ou decremento) a uma variável de chamada. Isso afeta variáveis locais, array de variáveis, fields e parâmetros.

A escolha que fiz inicial foi optar pela utilização das *configurações padrões do PIT Teste*. Assim, com o uso dos *operadores de mutação padrão* foram *produzidos 26 mutantes a partir do bytecode da minha Classe Imposto-Renda compilada, como mostrado no log do pós-processamento da ferramenta PITest* exibido na página seguinte e no Relatório de Cobertura do Teste de Mutação mostrado abaixo. Das *26 mutações produzidas, tivemos:*

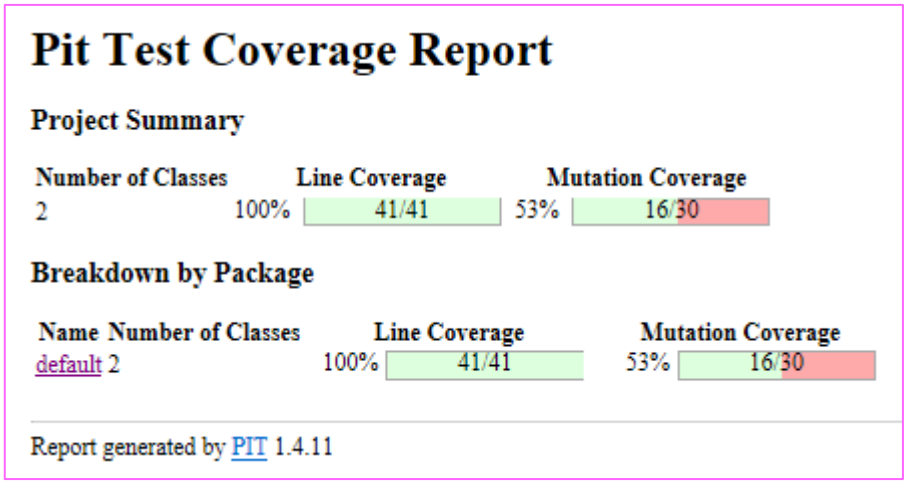
- *21 mutantes que foram **mortos**,*
- *2 mutantes vivos e*
- *3 mutantes que geraram nenhuma cobertura de código*

Na execução do ***Conjunto de Testes*** contidos na *classe TestaImpostoRenda que foram aplicados sobre os **26 programas mutantes** originados da minha classe original ImpstoRenda.*

5.3.4 – Relatório de Cobertura da Primeira Rodada dos Testes de Mutação

Depois de fazer os ajustes no arquivo metadado do projeto Java para configurar adequadamente o ambiente do ECLIPE IDE 2020-09, foi possível executar os testes de mutação usando a **ferramenta Pit Test** integrada com o **JUnit 4**.

A **primeira rodada dos testes de mutação** utilizando as mutações produzidas pelos operadores de mutação default, obtive, o Relatório de Teste de Cobertura com o resultado do processamento do conjunto de casos de testes sobre as Mutações cujo aspecto e constituição obtida está mostrada a seguir:



Pit Test Coverage Report

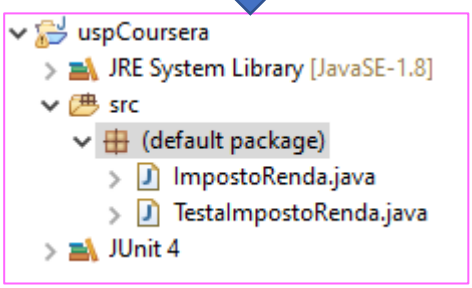
Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
2	100%	41/41	53%	16/30

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
default	2	100%	41/41	53%	16/30

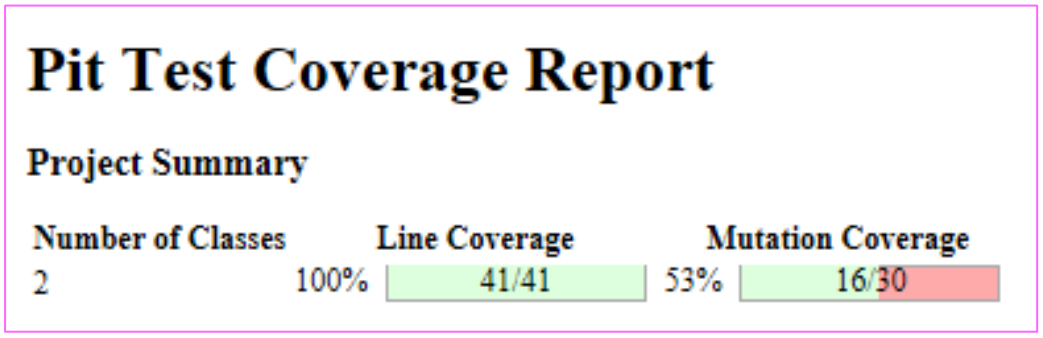
Report generated by PIT 1.4.11



Observe que, o Nome "**default 2**" no **PiT Test Coverage Report** refere-se ao "default package" e diz respeito as duas classes, respectivamente:

- *Classe ImpostoRenda.java*
- *Classe TestaImpostoRenda.java*

A1 - Análise do Desempenho Geral da Primeira Rodada dos Testes de Mutação Aplicados



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
2	100%	41/41	53%	16/30

Analisando a primeira parte do Relatório do Resultado da Execução das Mutações , denominado de Projec Summary, isto é, Resumo do Projeto, temos os seguintes aspectos:

i. Os *testes de Mutação* trabalharam com base em duas classes: A Classe *ImpostoRenda*, que contém o *algoritmo original objeto do teste,* e a Classe *TesteImpostoRenda* que contém o *Conjunto de Casos de teste;*

ii. A *Classe ImpostoRenda* possui 10 linhas de código e a Classe contendo o *Conjunto de Casos de Testes* denominado *TestaImpostoRenda* possui 31 linhas de código – o que totaliza 41 linhas de código;

iii. O *indicador de desempenho do processamento das mutações* denominado ***Line Coverage*** indica que 40 linhas de código das ***41 linhas de código*** *foram exercitadas (executadas) pelos 29 testes de mutação* –

iv. As ***36 variantes de código mutante com falhas*** introduzidas pela *ferramenta Pit teste* - ao serem aplicados os *29 operadores de mutação padrão*;

v. O valor d ***98% de cobertura de linha do código (Line Coverage)*** foi obtido dividindo-se o total de *41 linhas de código original* pelos *40 linhas de código executadas* – isto é, *apenas 1 linha de código não foi exercitada no processamento das mutações* – o que dá um índice ótimo, bem próximo a 100%.

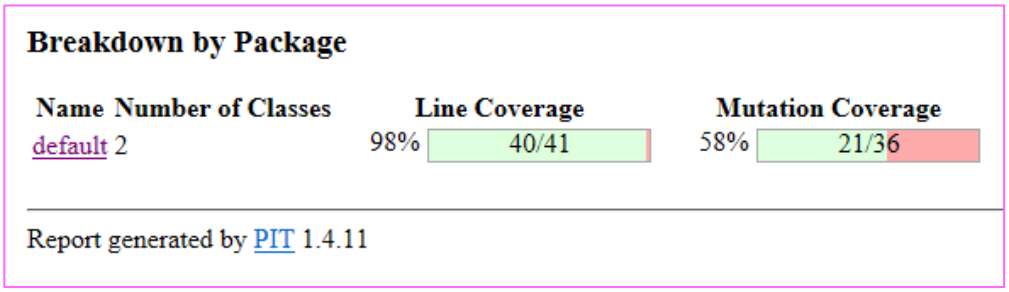
vi. Já o ***indicador de cobertura de mutação (mutation Coverage)*** é um medido de eficiência da qualidade dos Casos de Testes produzidos para testar falhas e defeitos na *Classe ImpostoRenda*. Este indicador é resultado da divisão entre o número total de mutações pelo número de mutantes mortos.

vii. Assim dos 36 mutantes tivemos 21 mortos, e portanto, 21 dividido por 36 nos gerou os 58% de desempenho de cobertura de mutação. Logo os 10 casos de testes não seriam adequados para pegar todas as situações de erros, e, portanto, precisaremos, segundo este indicado, de ter mais casos de testes unitários.

viii. No caso foram gerados ***36 mutantes***, dos quais ***21 foram mortos***. Dessa forma, tivemos ***15 situações*** que incluem ***mutantes vivos*** e ***não cobertura de código***.

ix. Porém, analisando os detalhes e os tipos de operadores aplicados, e as especificidades do nosso código da *Classe ImpostoRenda* e da *Classe de Casos de Testes TestaImpostoRenda*, dá para ver que, a estratégia de usar os operadores de mutação padrão não é adequada, porque, alguns operadores modificam os códigos de maneira que implica em anomalias do tipo parte do código sem cobertura por causa do tipo das falhas introduzidas nas duas classes – tanto na classe de teste quanto na classe principal objeto dos testes de mutação. Diante disso, estarei selecionado os operadores de mutação adequados e farei um segundo processamento das mutações para identificar a efetividade dos 10 casos de testes unitários como suficientes para testar a *classe ImpostoRenda*.

Analisando a segunda parte do Relatório do Resultado da Execução das Mutações , denominado de BREAK-DOWN BY PACKAGE, como mostrado abaixo:

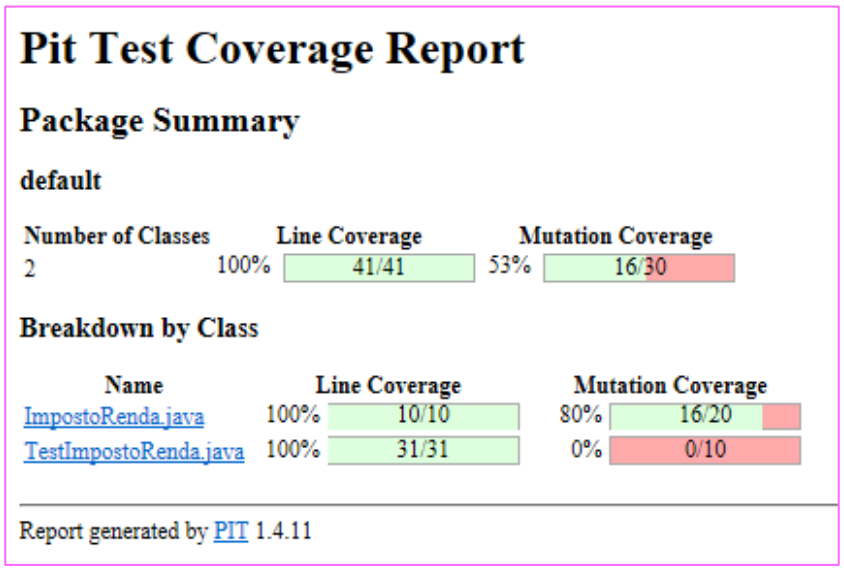


Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
default	2	98%	40/41	58%	21/36

Report generated by PIT 1.4.11

Fazendo zoom no package default (*isto é, clicando no link default do quando acima*) obtemos o seguinte:



Pit Test Coverage Report

Package Summary

default

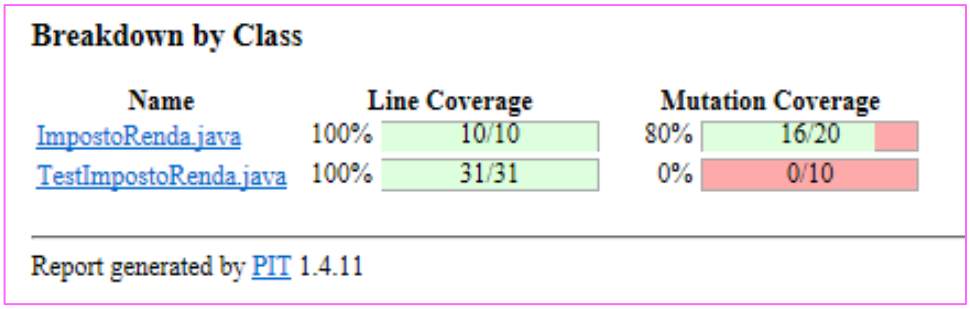
Number of Classes	Line Coverage		Mutation Coverage	
2	100%	41/41	53%	16/30

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
ImpostoRenda.java	100%	10/10	80%	16/20
TestImpostoRenda.java	100%	31/31	0%	0/10

Report generated by PIT 1.4.11

Isto é, obtemos o seguinte zoom e a decomposição por classe (*Breakdown by class*):



Breakdown by Class

Name	Line Coverage		Mutation Coverage	
ImpostoRenda.java	100%	10/10	80%	16/20
TestImpostoRenda.java	100%	31/31	0%	0/10

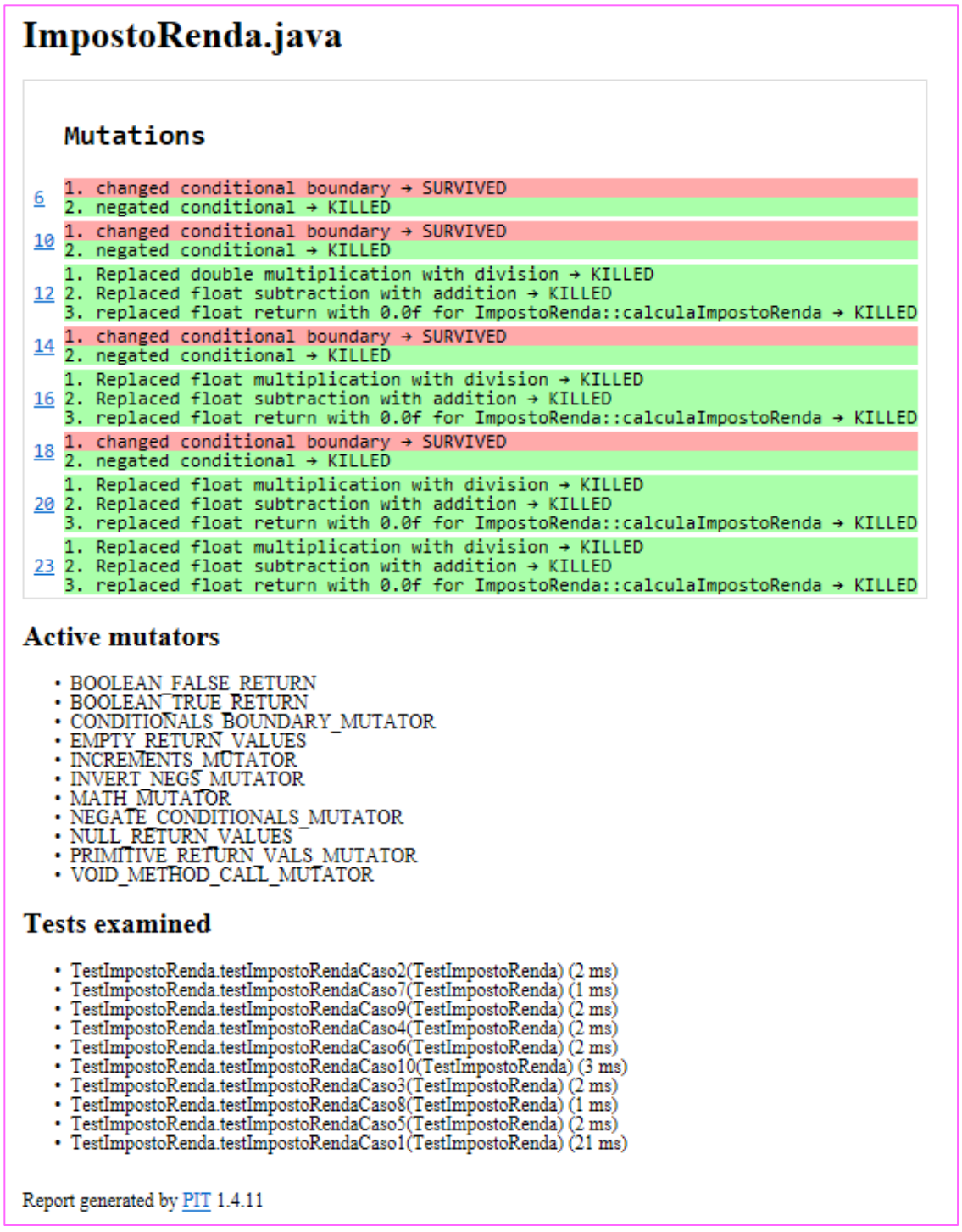
Report generated by PIT 1.4.11

Analisando o quadro acima:

i. A Classe com o *Conjunto de Casos de Teste* - Indicando que 31 linhas das 31 linhas da *Classe TesteImpostoRenda* foram executada (100% de cobertura do código) e que a Cobertura de Mutação foi 0% isto dos 10 testes nenhum deles foi testado. Este número distorce o *indicador geral de cobertura de mutação* e produziu apenas 53% de *cobertura de mutação*, em vez de 80% de mutação que é o correto. A *Classe ImpostoRenda* teve 9 das 10 linhas executadas (90% de cobertura de código) e dos 26 mutantes foram 21 mortos. Vamos olhas as falhas introduzidas pelo Pit Test em cada mutante gerado para as "duas" classes" a seguir:

5.3.4 –Primeira Rodada de Processamento dos Testes de Mutação sobre a ***Classe ImpostoRenda***

Abaixo o zoom que obtemos ao clicar no nome da *Classe ImpostoRenda* no Relatório Pit Test Coverage Report. O relatório apresenta quais foram os operadores de mutação incluídos no código e os efeitos decorrentes, para cada mutação se gerou mutantes vivos, mortos e não cobertura de código.



ImpostoRenda.java

Mutations

6	1. changed conditional boundary → SURVIVED 2. negated conditional → KILLED
10	1. changed conditional boundary → SURVIVED 2. negated conditional → KILLED
12	1. Replaced double multiplication with division → KILLED 2. Replaced float subtraction with addition → KILLED 3. replaced float return with 0.0f for ImpostoRenda::calculaImpostoRenda → KILLED
14	1. changed conditional boundary → SURVIVED 2. negated conditional → KILLED
16	1. Replaced float multiplication with division → KILLED 2. Replaced float subtraction with addition → KILLED 3. replaced float return with 0.0f for ImpostoRenda::calculaImpostoRenda → KILLED
18	1. changed conditional boundary → SURVIVED 2. negated conditional → KILLED
20	1. Replaced float multiplication with division → KILLED 2. Replaced float subtraction with addition → KILLED 3. replaced float return with 0.0f for ImpostoRenda::calculaImpostoRenda → KILLED
23	1. Replaced float multiplication with division → KILLED 2. Replaced float subtraction with addition → KILLED 3. replaced float return with 0.0f for ImpostoRenda::calculaImpostoRenda → KILLED

Active mutators

- BOOLEAN_FALSE_RETURN
- BOOLEAN_TRUE_RETURN
- CONDITIONALS_BOUNDARY_MUTATOR
- EMPTY_RETURN_VALUES
- INCREMENTS_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- NULL_RETURN_VALUES
- PRIMITIVE_RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR

Tests examined

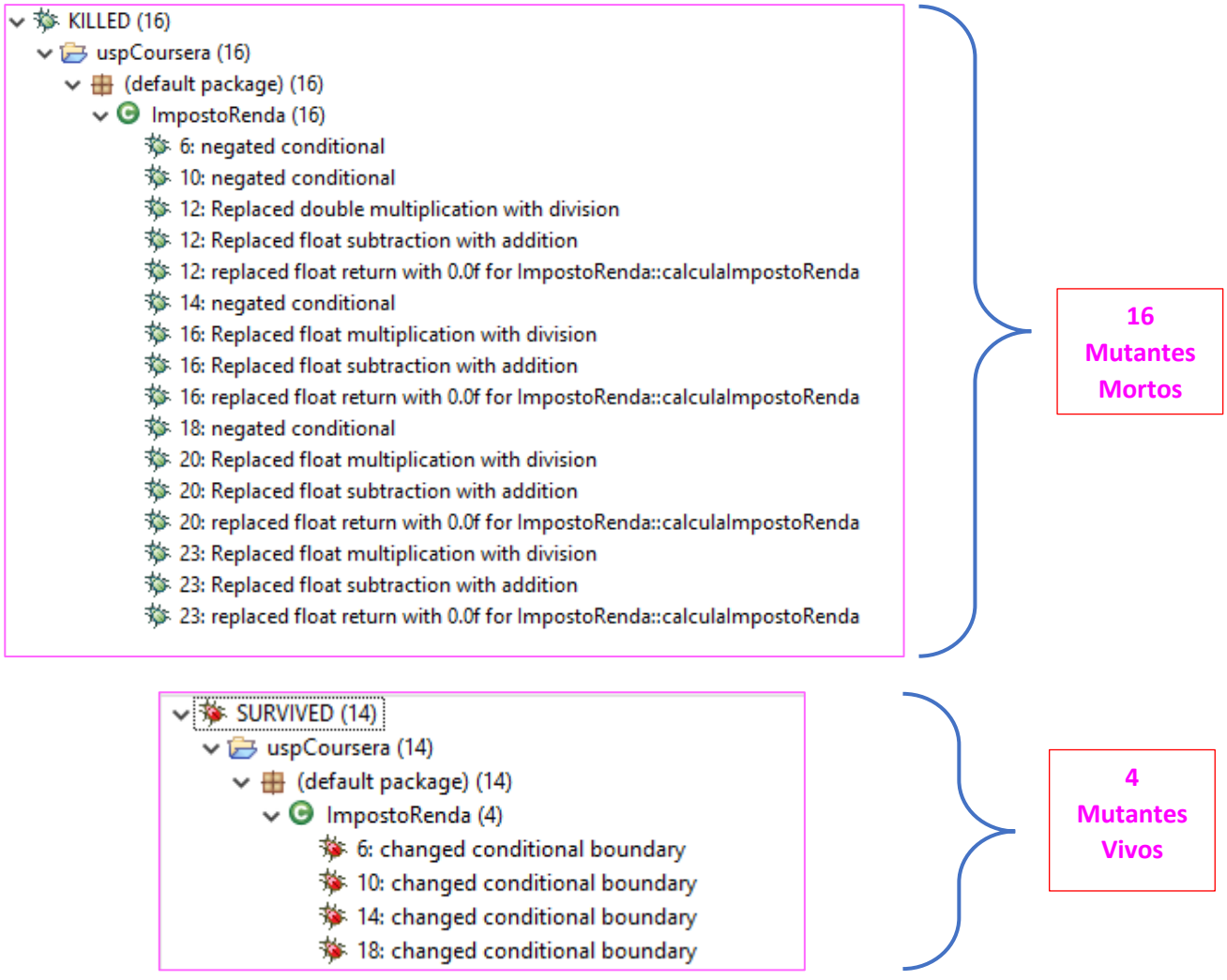
- TestImpostoRenda.testImpostoRendaCaso2(TestImpostoRenda) (2 ms)
- TestImpostoRenda.testImpostoRendaCaso7(TestImpostoRenda) (1 ms)
- TestImpostoRenda.testImpostoRendaCaso9(TestImpostoRenda) (2 ms)
- TestImpostoRenda.testImpostoRendaCaso4(TestImpostoRenda) (2 ms)
- TestImpostoRenda.testImpostoRendaCaso6(TestImpostoRenda) (2 ms)
- TestImpostoRenda.testImpostoRendaCaso10(TestImpostoRenda) (3 ms)
- TestImpostoRenda.testImpostoRendaCaso3(TestImpostoRenda) (2 ms)
- TestImpostoRenda.testImpostoRendaCaso8(TestImpostoRenda) (1 ms)
- TestImpostoRenda.testImpostoRendaCaso5(TestImpostoRenda) (2 ms)
- TestImpostoRenda.testImpostoRendaCaso1(TestImpostoRenda) (21 ms)

Report generated by PIT 1.4.11

Como pode ser visto no resultado da primeira rodada de processamento dos Testes de Mutação sobre a *Classe ImpostoRenda* utilizando os operadores de mutação padrão (default) da **ferramenta PiT teste**, foram aplicadas as seguintes tipos de mutações no código e resultou em:

Análise das Mutações Inseridas na Classe ImpostoRenda e os Efeitos Obtidos			
Operador de Mutação Usado	Mutantes Mortos	Mutantes Vivos	Código Sem Cobertura
1 - Changed Conditional Boundary		4	
2 - Negated Conditional	4		
3 – Replaced Float Subtaction with addition	4		
4 - Replaced Float Multiplication with division	4		
5 - Replaced Float Return with 0.0f	4		
Total de Mutações por Status	16	4	-
Total Geral de Mutações	20		

Assim o desempenho é excelente, indicando que os *Casos de Testes Unitários* tiveram uma adequação de 80% de cobertura de mutação – ***dos 20 mutantes gerados da Classe ImpostoRenda foram mortos 16 e sobraram vivos apenas 4.***



Infelizmente as **classes mutantes** geradas pela *ferramenta PiT Teste* são mantidas apena em memória, não sendo possível ter acesso ao conteúdo exato das falhas introduzidas no código. Esta estratégia de manter apenas as classes mutantes em memória tem por objetivo evitar que os códigos com falhas geradas pela automação não tenham o perigo de parar em um build de produção e no release liberado para o cliente final.

Como na minha implementação das faixas de imposto de renda, trato os valores negativos de qualquer natureza e valores menores do que 1.903,99 como viáveis e aceitos, e gero o valor zero para o imposto de renda. Dessa forma, como os operador de mutação que produziu **mutantes equivalentes** (vivos) é o operador Changed Conditional Boundary, *existe a possibilidade de que as mudanças nas condições de valor limite das classes que tratam as diferentes faixas do imposto de renda possam ocasionar em valor de input válido e portanto gerando o imposto calculado com valor zero, isto é, um valor correto e adequado. Logo, a falha introduzida acaba gerando valor válido de input e output valido, e o teste irá consequentemente passar e não falhar*.

Assim, para melhorar o SCORE dos TESTES de MUTAÇÃO eu teria a condição de configurar a exclusão deste tipo de operador de mutação no processamento do Teste de Mutação da minha classe ImpostoRenda. A inclusão dos seguintes parâmetros em formato XML, no arquivo de metadado MAVEN utilizado pela Ferramenta Pit Teste, para obter 100% de Cobertura de Mutação:

```
<configuration>
    <mutators>
        <mutator>NEGATE_CONDITIONALS_MUTATOR</mutator>
        <mutator>MATH_MUTATOR</mutator>
        <mutator>BOOLEAN_FALSE_RETURN</mutator>
        <mutator>EMPTY_RETURN_VALUES</mutator>
        <mutator>INCREMENTS_MUTATOR</mutator>
        <mutator>INVERT_NEGS_MUTATOR</mutator>
        <mutator>NULL_RETURN_VALUES</mutator>
        <mutator>PRIMITIVE_RETURN_VALS_MUTATOR</mutator>
        <mutator>VOID_METHOD_CALL_MUTATOR</mutator>
    </mutators>
</configuration>
```

Seção FINAL – ANÁLISE DOS RESULTADOS e PRÓXIMOS PASSOS

No meu ponto de vista, os resultados obtidos pelas técnicas de teste funcional, complementada pelo teste estrutural, e pela execução do teste de mutação (um indicador da qualidade dos casos de testes projetados). Constituem uma sequência necessárias de diferentes tipos de testes, que pretendo usar como prática válida aplicada no desenvolvimento de todas as minhas aplicações, independente da natureza delas. Estou muito satisfeito de ter tido a oportunidade de participar destes Curso de Introdução ao Teste de Software da Plataforma Coursera/USP e de ter podido ler diversos livros técnicos sobre o assunto, para me embasar e estender me conhecimento, de ter tido condições de concluir esta tarefa adequadamente, que me agregou muito na minha formação profissional. Como trabalho com diversas linguagens de programação, diversas plataformas e stacks tecnológicas, os meus próximos passos serão continuar as estudar e pesquisas as ferramentas de automação de testes unitários voltados aos testes funcionais, estruturais e de mutação. Como este universo de teste de software é complexo e amplo, preciso ainda estudar os testes de integração e de testes de aceitação, entre outros. Estou no momento cursando na plataforma Coursera o TDD (Test Driven Development). Sem a prática dos testes com uso adequado de técnicas e critérios será impossível produzir software de qualidade e com o mínimo de erros possíveis e com o menor custo.