EXPLICAÇÃO DETALHADA – TP2

ATIVIDADES DE REFATORAÇÃO

1. Extrair Método

Classe: VENDA

Método: calcularFrete(CLIENTE cliente)

Introdução: o método calcularFrete recebe como parâmetro o cliente que está realizando a compra e, a partir disso, cria uma variável temporária denominada **regiaoCliente**, responsável por armazenar a região em que o cliente se encontra. Essa informação é necessária para calcular o valor bruto do frete sobre a comprar. Logo em seguida, uma outra variável auxiliar, denominada **frete**, é declarada com valor inicial 0, a fim de receber, no futuro, o valor total do frete a ser pago. Em resumo, tem-se as seguintes **variáveis locais**:

- regiãoCliente
- frete

Por fim, sobre o método em questão, tem-se dois blocos de código em que a criação de comentários se fez necessário para explicá-los (**mal cheiro de código**). São eles:

- Bloco de código 1: verificando o valor do frete

```
// Verificando valor do frete
switch (regiaoCliente) {
    case Regiao.DISTRITO_FEDERAL:
        if (cliente.isCapital() == true) {
            frete = 5;
        }
        else {
            frete = 0;
        }
    case Regiao.REGIAO_CENTRO_OESTE:
        if (cliente.isCapital() == true) {
            frete = 10;
        }
        else {
            frete = 13;
        }
    case Regiao.REGIAO_NORDESTE:
        if (cliente.isCapital() == true) {
            frete = 15;
        }
        else {
            frete = 18;
        }
    case Regiao.REGIAO_NORTE:
        if (cliente.isCapital() == true) {
            frete = 20;
        }
        else {
            frete = 25;
        }
    case Regiao.REGIAO_SUDESTE:
        if (cliente.isCapital() == true) {
            frete = 7;
        }
        else {
            frete = 10;
        }
    case Regiao.REGIAO_SUL:
        if (cliente.isCapital() == true) {
            frete = 10;
        }
        else {
            return 13;
        }
}
```

- Bloco de código 2: aplicando o desconto de frete para clientes especiais e prime

```
    // Aplicar desconto de frete para clientes especiais e prime
    if (cliente.getTipoCliente() == TipoCliente.PRIME) {
        frete = 0;
    } else if (cliente.getTipoCliente() == TipoCliente.ESPECIAL) {
        frete *= 0.7;
    }

    return frete;
}
```

]

Aplicando a técnica de refatoração: de acordo com as imagens evidenciadas acima, ambos os blocos de código realizam escrita sobre a variável local frete, com a diferença que o primeiro bloco também realiza uma leitura sobre a variável local regiaoCliente. Diante do exposto, considere as seguintes etapas de refatoração utilizando a técnica Extrair Método sobre o primeiro bloco de código:

- Etapa 1 - criando um método cujo nome retrata o que ele faz

```
public void verificaValorDoFrete() {

}
```

- Etapa 2 – copiando o código extraído do método de origem para o novo método

```
public void verificaValorDoFrete() {
    switch (regiaoCliente) {
    case Regiao.DISTRITO_FEDERAL:
        if (cliente.isCapital() == true) {
            frete = 5;
        }
        else {
            frete = 0;
        }
    case Regiao.REGIAO_CENTRO_OESTE:
        if (cliente.isCapital() == true) {
            frete = 10;
        }
        else {
            frete = 13;
        }
    case Regiao.REGIAO_NORDESTE:
        if (cliente.isCapital() == true) {
            frete = 15;
        }
        else {
            frete = 18;
        }
    case Regiao.REGIAO_NORTE:
        if (cliente.isCapital() == true) {
            frete = 20;
        }
        else {
            frete = 25;
        }
    case Regiao.REGIAO_SUDESTE:
        if (cliente.isCapital() == true) {
            frete = 7;
        }
        else {
            frete = 10;
        }
    case Regiao.REGIAO_SUL:
        if (cliente.isCapital() == true) {
            frete = 10;
        }
        else {
            return 13;
        }
    }
}
```

- Etapa 3 – procurando no método extraído por referências para variáveis locais declaradas no escopo do método de origem. Elas serão variáveis locais e parâmetros para o novo método

Conforme sublinhado em vermelho pela IDE Eclipse, utilizada para realização do projeto, observa-se duas referências: uma sobre a variável **frete** e outra sobre a variável **regiaoCliente**.

- Etapa 4 - verificando se as variáveis temporárias são usadas apenas dentro do código extraído. Se sim, declará-las como variáveis temporárias no novo método

  A variável temporária **regiaoCliente** é utilizada somente pelo primeiro bloco de código, logo ela será declarada como variável local no novo método.

```
public void verificaValorDoFrete() {
    Regiao regiaoCliente = CLIENTE.getVerificarRegiao(cliente.getEstado());
```

- Etapa 5 - procurando no código extraído se há alguma variável local que é modificada no código, pois se uma variável local é modificada, verifica-se se pode tratar o código como uma consulta e atribuir o resultado à variável em questão.

  A variável local **frete** do método de origem é modificada no código extraído, logo ela deverá ser tratada como uma consulta (por meio do parâmetro do método) e seu valor atualizado definido como retorno do novo método.

- Etapa 6 – passando como parâmetro para o método-alvo (novo método) variáveis de escopo local que são lidas do código extraído

  Para o caso atual (bloco de código 1), será necessário passar como parâmetro a variável **frete** que será atualizada e o **objeto cliente** que está realizando a compra para verificar sua região.

```
public void verificaValorDoFrete(CLIENTE cliente, int frete) {
    Regiao regiaoCliente = CLIENTE.getVerificarRegiao(cliente.getEstado());
```

- Etapa 7 – últimos ajustes: indicando o retorno do novo método

Como o objetivo do novo método é calcular o valor bruto do frete de acordo com a sua região, é necessário definir como valor de retorno um inteiro, que será indicado pelo valor que a variável **frete** possuir. Sendo assim, ao concatenar todas as etapas anteriores, tem-se o seguinte método-alvo resultante:

```
public int verificaValorDoFrete(CLIENTE cliente, int frete) {
    Regiao regiaoCliente = CLIENTE.getVerificarRegiao(cliente.getEstado());

    switch (regiaoCliente) {
    case Regiao.DISTRITO_FEDERAL:
        if (cliente.isCapital() == true) {
            frete = 5;
        }
        else {
            frete = 0;
        }
    case Regiao.REGIAO_CENTRO_OESTE:
        if (cliente.isCapital() == true) {
            frete = 10;
        }
        else {
            frete = 13;
        }
    case Regiao.REGIAO_NORDESTE:
        if (cliente.isCapital() == true) {
            frete = 15;
        }
        else {
            frete = 18;
        }
    case Regiao.REGIAO_NORTE:
        if (cliente.isCapital() == true) {
            frete = 20;
        }
        else {
            frete = 25;
        }
    case Regiao.REGIAO_SUDESTE:
        if (cliente.isCapital() == true) {
            frete = 7;
        }
        else {
            frete = 10;
        }
    case Regiao.REGIAO_SUL:
        if (cliente.isCapital() == true) {
            frete = 10;
        }
        else {
            return 13;
        }
    }
    return frete;
}
```

- Etapa 8 – substituindo o código extraído no método fonte por uma chamada ao método-alvo,

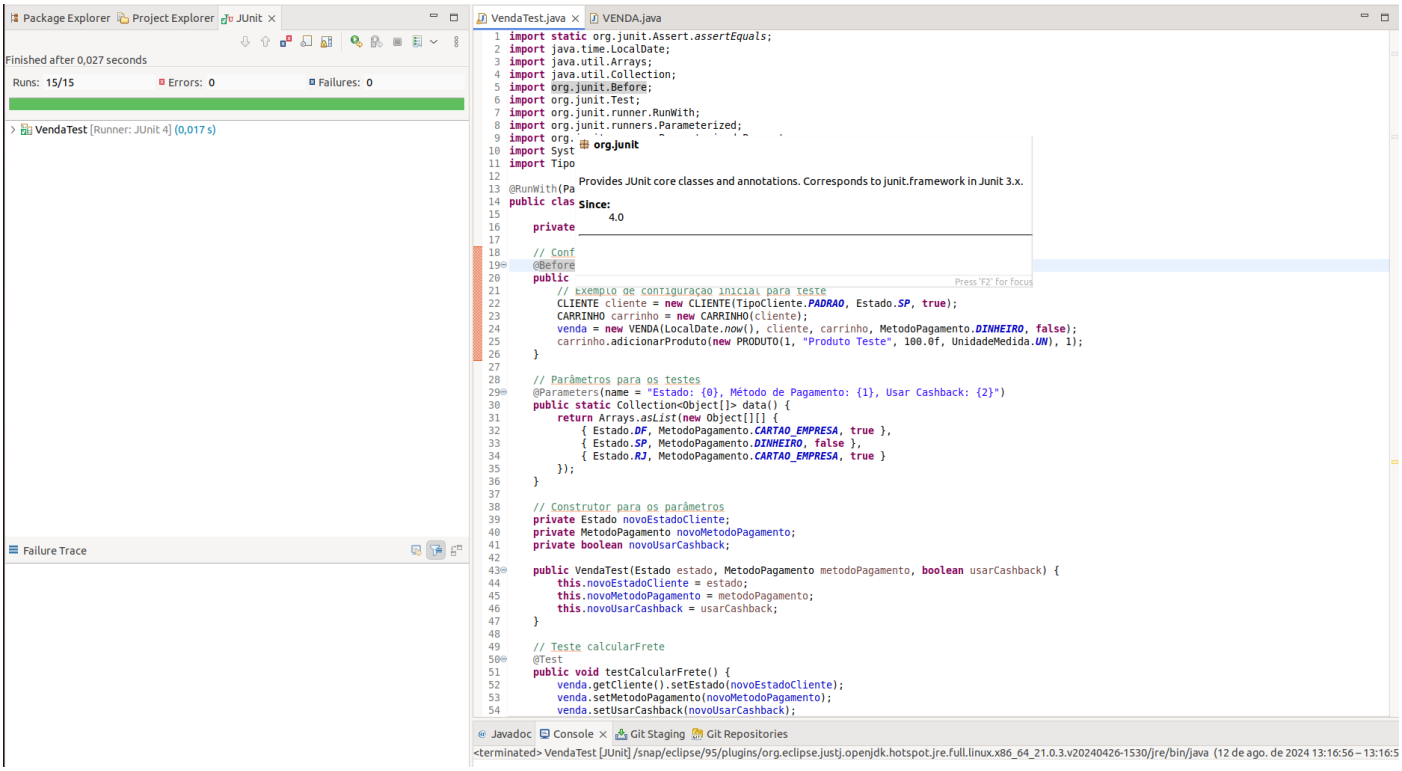
```java
// Calcular Frete
public int calcularFrete (CLIENTE cliente) {
    // Variaveis auxiliares
    int frete = 0;

    frete = verificaValorDoFrete(cliente, frete);

    // Aplicar desconto de frete para clientes especiais e prime
    if (cliente.getTipoCliente() == TipoCliente.PRIME) {
        frete = 0;
    } else if (cliente.getTipoCliente() == TipoCliente.ESPECIAL) {
        frete *= 0.7;
    }

    return frete;
}
```

- Etapa 9 – verificando compilação

```
Problems  Javadoc  Declaration  Console ×
<terminated> App [Java Application] /snap/eclipse/95/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.linux.x86_64_21.0.
Total da Venda 1: 102.79999

Detalhes da Venda 2: Venda{dataVenda=2024-08-12, cliente=Cliente{tipoCliente=ESPECIAL, estado=MG, capi
Total da Venda 2: 43.8

Detalhes da Venda 3: Venda{dataVenda=2024-08-12, cliente=Cliente{tipoCliente=PRIME, estado=DF, capital
Total da Venda 3: 94.4

Cashback do Cliente 1 após venda: 0.0
Cashback do Cliente 2 após venda: 0.0
Cashback do Cliente 3 após venda: 4.7200003

Detalhes da Venda 4: Venda{dataVenda=2024-08-12, cliente=Cliente{tipoCliente=PRIME, estado=DF, capital
Total da Venda 4: 113.28
Cashback do Cliente 3 após venda: 0.0
```

Código compilando com sucesso

- Etapa 10 – testando por meio da suíte de teste definida no trabalho 1 – TDD



A suíte de testes continua sem erros (verde)

Após o término com sucesso do primeiro bloco de código, foi aplicado a mesma técnica sobre o segundo. Somado a isso, é importante ressaltar que, ao comprá-lo com o bloco de código anterior, este também é caracterizado por modificar a variável **frete**, com a diferença de não necessitar consultar a variável **regiaoCliente** do método pai. Por fim, as etapas da técnica extrair método podem ser verificadas a seguir:

- Etapa 1 - criando um método cujo nome retrata o que ele faz

```
public void aplicarDesconto() {

}
```

- Etapa 2 – copiando o código extraído do método de origem para o novo método

```
public void aplicarDesconto() {
    if (cliente.getTipoCliente() == TipoCliente.PRIME) {
        frete = 0;
    } else if (cliente.getTipoCliente() == TipoCliente.ESPECIAL) {
        frete *= 0.7;
    }
}
```

- Etapa 3 – procurando no método extraído por referências para variáveis locais declaradas no escopo do método de origem. Elas serão variáveis locais e parâmetros para o novo método

  Conforme sublinhado em vermelho pela IDE Eclipse, utilizada para realização do projeto, observa-se uma única referência sobre a variável **frete**.

- Etapa 4 - verificando se as variáveis temporárias são usadas apenas dentro do código extraído. Se sim, declará-las como variáveis temporárias no novo método

  A variável frete também é utilizada no bloco de código 1, logo não poderá ser declarada como variável temporária no método atual, sendo necessário recebê-la como um parâmetro.

- Etapa 5 - procurando no código extraído se há alguma variável local que é modificada no código, pois se uma variável local é modificada, verifica-se se pode tratar o código como uma consulta e atribuir o resultado à variável em questão.

  Assim como o bloco de código anterior, no bloco de código atual a variável local **frete** do método de origem é modificada no código extraído, logo ela deverá ser tratada como uma consulta e seu valor atualizado definido como retorno do novo método.

- Etapa 6 – passando como parâmetro para o método-alvo (novo método) variáveis de escopo local que são lidas do código extraído

  Para o caso atual (bloco de código 2), será necessário passar como parâmetro a variável **frete** que será atualizada e o **objeto cliente** que está realizando a compra para verificar seu tipo (normal, especial ou prime).

```
public void aplicarDesconto(CLIENTE cliente, int frete) {
    if (cliente.getTipoCliente() == TipoCliente.PRIME) {
        frete = 0;
    } else if (cliente.getTipoCliente() == TipoCliente.ESPECIAL) {
        frete *= 0.7;
    }
}
```

- Etapa 7 – últimos ajustes: indicando o retorno do novo método

  Como o objetivo do novo método é calcular o valor do desconto a ser aplicado sobre o frete de acordo com o tipo do cliente, é necessário definir como valor de retorno um inteiro, que será indicado pelo valor que a variável frete possuir. Sendo assim, ao concatenar todas as etapas anteriores, tem-se o seguinte método-alvo resultante:

```
public int aplicarDesconto(CLIENTE cliente, int frete) {
    if (cliente.getTipoCliente() == TipoCliente.PRIME) {
        frete = 0;
    } else if (cliente.getTipoCliente() == TipoCliente.ESPECIAL) {
        frete *= 0.7;
    }
    return frete;
}
```

- Etapa 8 – substituindo o código extraído no método fonte por uma chamada ao método-alvo,

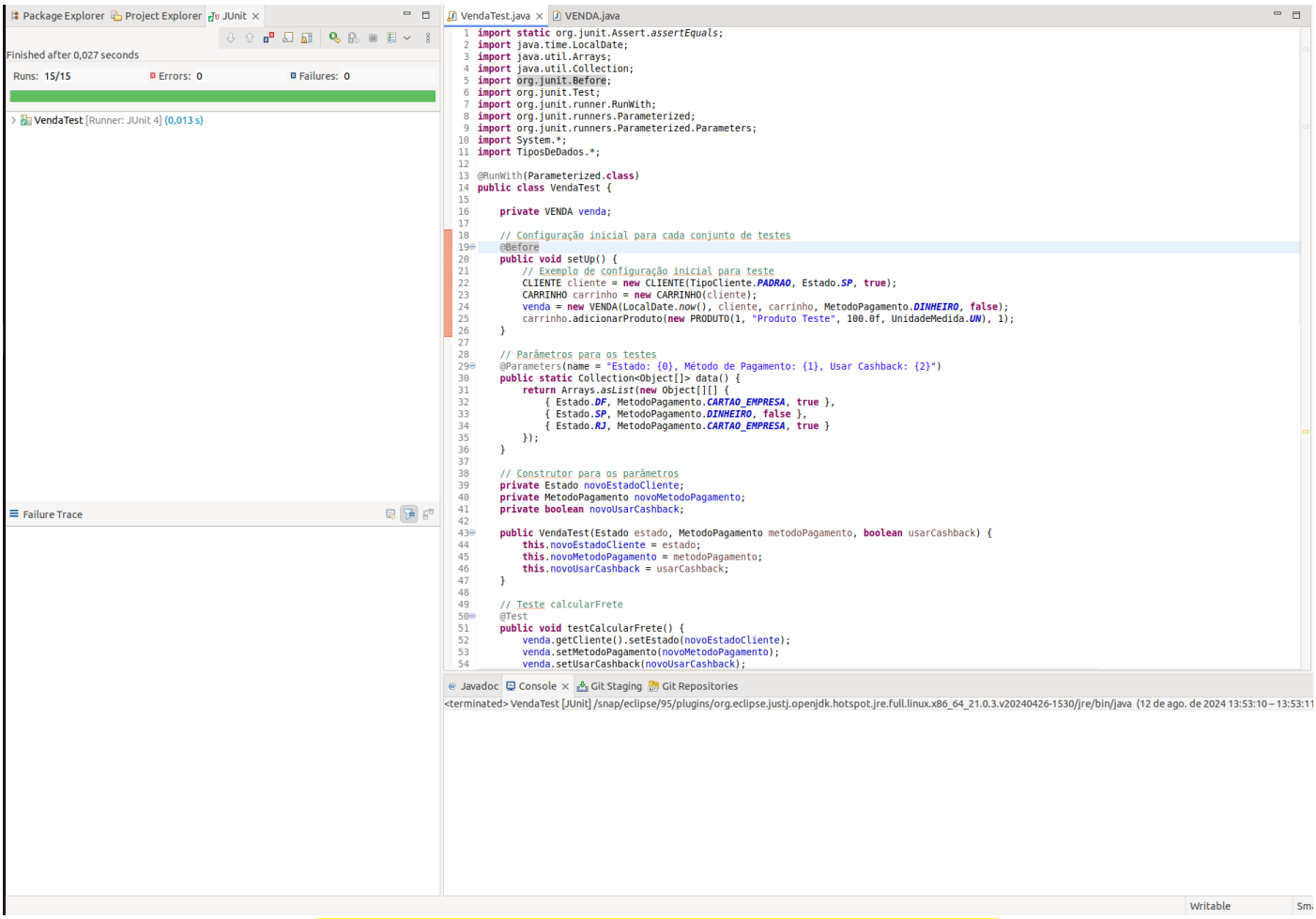
```
// Calcular Frete
public int calcularFrete (CLIENTE cliente) {
    // Variaveis auxiliares
    int frete = 0;

    frete = verificaValorDoFrete(cliente, frete);

    frete = aplicarDesconto(cliente, frete);

    return frete;
}
```

- Etapa 9 – verificando compilação

```
<terminated> New_configuration (1) [Java Application] /snap/eclipse/95/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.linux.x86_64_21.0.3.v20240426-1530/jre/bin/java (12 de ago. de 2024 13:52:01
Detalhes da Venda 1: Venda{dataVenda=2024-08-12, cliente=Cliente{tipoCliente=PADRAO, estado=SP, capital=true}, itensVendidos=CARRINHO{cliente=Cliente{tipoCliente=PADR
Total da Venda 1: 102.799995

Detalhes da Venda 2: Venda{dataVenda=2024-08-12, cliente=Cliente{tipoCliente=ESPECIAL, estado=MG, capital=false}, itensVendidos=CARRINHO{cliente=Cliente{tipoCliente=E
Total da Venda 2: 43.8

Detalhes da Venda 3: Venda{dataVenda=2024-08-12, cliente=Cliente{tipoCliente=PRIME, estado=DF, capital=true}, itensVendidos=CARRINHO{cliente=Cliente{tipoCliente=PRIME
Total da Venda 3: 94.4

Cashback do Cliente 1 após venda: 0.0
Cashback do Cliente 2 após venda: 0.0
Cashback do Cliente 3 após venda: 4.7200003

Detalhes da Venda 4: Venda{dataVenda=2024-08-12, cliente=Cliente{tipoCliente=PRIME, estado=DF, capital=true}, itensVendidos=CARRINHO{cliente=Cliente{tipoCliente=PRIME
Total da Venda 4: 113.28
```

Código compilando com sucesso

- Etapa 10 – testando por meio da suíte de teste definida no trabalho 1 – TDD



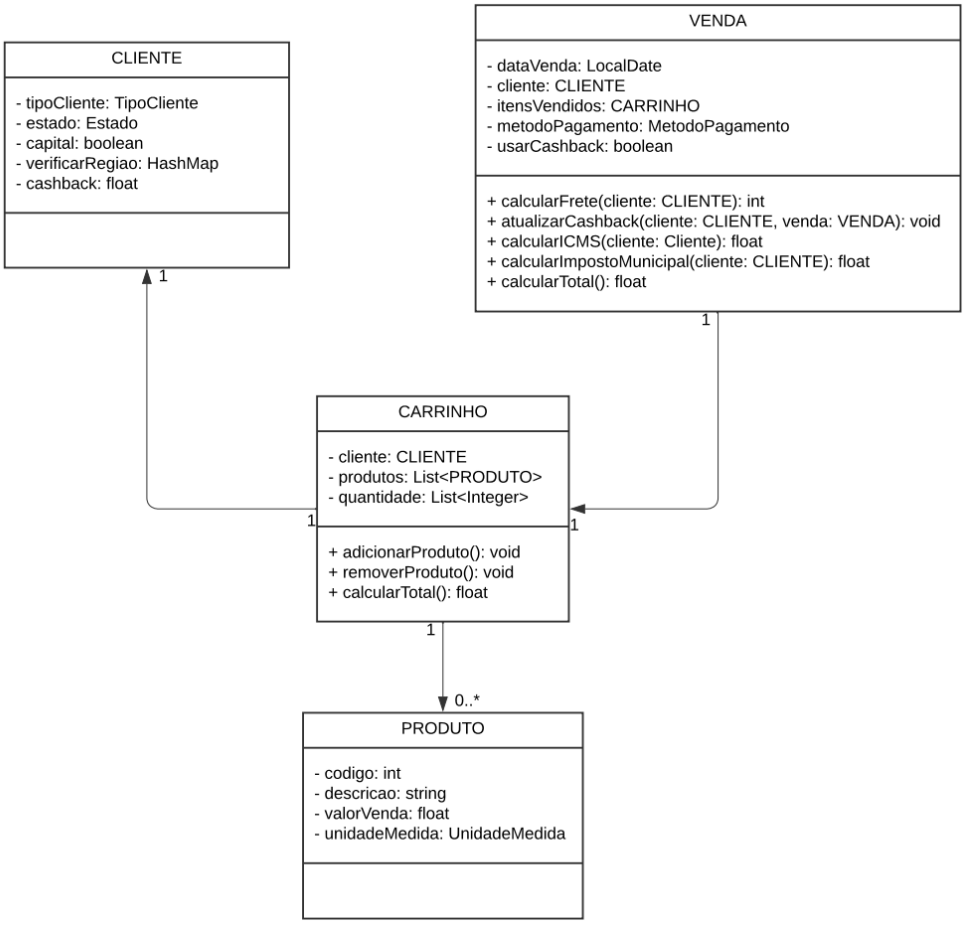
A suíte de testes também continua sem erros (verde)

Conclusão: apesar de ser um método de refatoração automatizado, para fins de aprendizado, esta técnica foi aplicada a mão. Somado a isso, percebe-se que o método calcularFrete, incialmente longo e com necessidade de comentários para ser entendido, se tornou mais intuitivo devido ao aumento de sua modularidade por meio da substituição de rotinas relativamente longas por chamadas de novos métodos criados a partir dessas rotinas.

2. Extrair Classe

Classe: CLIENTE

Introdução: a classe **CLIENTE** está realizando trabalho que poderia ser feito por duas classes distintas, isto é, é possível separá-la em duas novas classes e atribuir um relacionamento entre elas. Isso fica mais claro quando se analisa os atributos **estado**, **capital** e **verificarRegiao**, os quais se referem ao **endereço do cliente** e podem originar uma nova classe denominada **ENDERECO**. Sendo assim, a fim de tornar a classe em questão em uma abstração pura, a qual lida com algumas poucas responsabilidades claras (em seu contexto), o método extrair classe se faz extremamente útil.



Artefato Diagrama de Classes UML do projeto

Aplicando a técnica de refatoração: de acordo com o diagrama de classes acima, confere-se que os atributos estado, capital e verificarRegiao são informações específicas do endereço do cliente, o que pode ser extraído, juntamente com eventuais métodos, para uma nova classe ENDERERCO. Essa observação servirá de base para à **decisão de divisão** da técnica **extrair classe**. Sendo assim, confira as etapas de aplicação da técnica em questão a seguir:

- Etapa 1: decidindo como dividir as responsabilidades da classe.

  Conforme evidenciado na introdução desse tópico, ficou decidido que os atributos estado, capital, e verificarRegiao, juntamente com eventuais métodos, originarão uma nova classe denominada ESTADO.

- Etapa 2: criando classe para receber as responsabilidades.

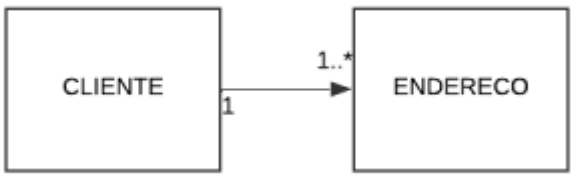
  Conforme discutido anteriormente, a nova classe deverá ter o nome ENDERECO. Somado a isso, é importante ressaltar que a responsabilidade da classe anterior (CLIENTE) ainda corresponde ao seu nome, não sendo necessário alterá-lo.

```
package System;

public class ENDERECO {

}
```

Nova classe denominada ENDERECO

- Etapa 3: criando associação da classe antiga para a nova classe.

  É intuitivo dizer que um cliente possa possuir diferentes endereços em uma situação real de comércio. Contudo, de acordo com o que foi definido no projeto anterior (TP1 – TDD), cada cliente só possui um único estado somado a informação de estar localizado na capital ou não, o que indica haver apenas um endereço (se houvesse múltiplos estados faria sentido, no novo contexto, haver múltiplos endereços, mas não é o caso). Diante do exposto, devido ao fato da atividade atual ser uma refatoração, não será modificado a estrutura do projeto, sendo a relação entre CLIENTE e ENDERERCO como 1 para N (CLIENTE possui um ENDERECO, enquanto um ENDERERCO pode ser atribuído a um ou mais CLIENTE).



- Etapa 4: usando a técnica **Mover Campo/Atributo** em cada atributo que será movido.

  4.1 criando três campos na classe alvo (ENDERECO) com o mesmo nome e regra de acesso.

```java
public class ENDERECO {

    private Estado estado;
    private boolean capital;
    private static final Map<Estado, Regiao> verificarRegiao = new HashMap<>();
    static {
        verificarRegiao.put(Estado.DF, Regiao.DISTRITO_FEDERAL);
        verificarRegiao.put(Estado.GO, Regiao.REGIAO_CENTRO_OESTE);
        verificarRegiao.put(Estado.MT, Regiao.REGIAO_CENTRO_OESTE);
        verificarRegiao.put(Estado.MS, Regiao.REGIAO_CENTRO_OESTE);
        verificarRegiao.put(Estado.AL, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.BA, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.CE, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.MA, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.PB, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.PN, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.PI, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.RN, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.SE, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.AM, Regiao.REGIAO_NORTE);
        verificarRegiao.put(Estado.RR, Regiao.REGIAO_NORTE);
        verificarRegiao.put(Estado.RO, Regiao.REGIAO_NORTE);
        verificarRegiao.put(Estado.AP, Regiao.REGIAO_NORTE);
        verificarRegiao.put(Estado.TO, Regiao.REGIAO_NORTE);
        verificarRegiao.put(Estado.ES, Regiao.REGIAO_SUDESTE);
        verificarRegiao.put(Estado.MG, Regiao.REGIAO_SUDESTE);
        verificarRegiao.put(Estado.RJ, Regiao.REGIAO_SUDESTE);
        verificarRegiao.put(Estado.SP, Regiao.REGIAO_SUDESTE);
        verificarRegiao.put(Estado.PR, Regiao.REGIAO_SUL);
        verificarRegiao.put(Estado.SC, Regiao.REGIAO_SUL);
        verificarRegiao.put(Estado.RS, Regiao.REGIAO_SUL);
    }
```

Observe que o atributo VerificarRegiao aparenta ser um método. Contudo, deve-se entendê-lo como um atributo do tipo HashMap, isto é, uma instância da classe HashMap. Somado a isso, devido ao prefixo **static**, ele é um atributo da classe, e não de suas instâncias, o que significa que todas as instâncias terão acesso a seu conteúdo previamente definido.

4.2 criando os **getters** e **setters** dos campos na classe-alvo.

```java
public Estado getEstado() {
    return estado;
}
public void setEstado(Estado estado) {
    this.estado = estado;
}
public boolean isCapital() {
    return capital;
}
public void setCapital(boolean capital) {
    this.capital = capital;
}

public static Regiao getVerificarRegiao(Estado estado) {
    return verificarRegiao.get(estado);
}
```

4.3 garantindo que a classe de origem possa acessar o objeto da classe-alvo.

A classe **CLIENTE** pode se referir à classe **ENDERERCO** por meio de um novo atributo, do tipo **ENDERECO**.

```java
public class CLIENTE {

    // Atributos
    private TipoCliente tipoCliente;
    private ENDERECO endereco;
    private Estado estado;
    private boolean capital;
    private float cashback;
    private static final Map<Estado, Regiao> verificarRegiao = new HashMap<>();
    static {
        verificarRegiao.put(Estado.DF, Regiao.DISTRITO_FEDERAL);
        verificarRegiao.put(Estado.GO, Regiao.REGIAO_CENTRO_OESTE);
        verificarRegiao.put(Estado.MT, Regiao.REGIAO_CENTRO_OESTE);
        verificarRegiao.put(Estado.MS, Regiao.REGIAO_CENTRO_OESTE);
        verificarRegiao.put(Estado.AL, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.BA, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.CE, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.MA, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.PB, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.PN, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.PI, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.RN, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.SE, Regiao.REGIAO_NORDESTE);
        verificarRegiao.put(Estado.AM, Regiao.REGIAO_NORTE);
        verificarRegiao.put(Estado.RR, Regiao.REGIAO_NORTE);
        verificarRegiao.put(Estado.RO, Regiao.REGIAO_NORTE);
        verificarRegiao.put(Estado.AP, Regiao.REGIAO_NORTE);
        verificarRegiao.put(Estado.TO, Regiao.REGIAO_NORTE);
        verificarRegiao.put(Estado.ES, Regiao.REGIAO_SUDESTE);
        verificarRegiao.put(Estado.MG, Regiao.REGIAO_SUDESTE);
        verificarRegiao.put(Estado.RJ, Regiao.REGIAO_SUDESTE);
        verificarRegiao.put(Estado.SP, Regiao.REGIAO_SUDESTE);
        verificarRegiao.put(Estado.PR, Regiao.REGIAO_SUL);
        verificarRegiao.put(Estado.SC, Regiao.REGIAO_SUL);
        verificarRegiao.put(Estado.RS, Regiao.REGIAO_SUL);
    }
```

4.3 removendo os atributos da classe fonte.

```java
public class CLIENTE {

    // Atributos
    private TipoCliente tipoCliente;
    private ENDERECO endereco;
    private float cashback;
```

4.4 atualizando o construtor da classe fonte.

```java
// Construtor
public CLIENTE(TipoCliente tipoCliente, ENDERECO endereco) {
    this.tipoCliente = tipoCliente;
    this.endereco = endereco;
    this.cashback = 0;
}
```

4.5 definindo os métodos **get** e **set** para o atributo endereco na classe fonte.

```java
public ENDERECO getEndereco() {
    return endereco;
}

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

4.6 apagando os getters e setters dos atributos antigos na classe fonte.

```java
// Getters e Setters
public TipoCliente getTipoCliente() {
    return tipoCliente;
}

public void setTipoCliente(TipoCliente tipoCliente) {
    this.tipoCliente = tipoCliente;
}

public ENDERECO getEndereco() {
    return endereco;
}

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

public float getCashback() {
    return cashback;
}

public void setCashback(float cashback) {
    this.cashback = cashback;
}
```

- Etapa 5: usando a técnica **Mover Método** para mover o método **verificarRegiao** da classe antiga para a nova classe.

  Não há métodos além de getters e setters para serem movidos, o que torna a aplicação da técnica extrair classe mais simples, sendo necessário apenas adiciona a chamada da **instância endereco** da classe ENDERECO por meio de uma **instancia cliente** da classe CLIENTE para que possa **acessar os recursos que anteriores eram exclusivos da classe fonte**.

- Etapa 6 – verificando compilação

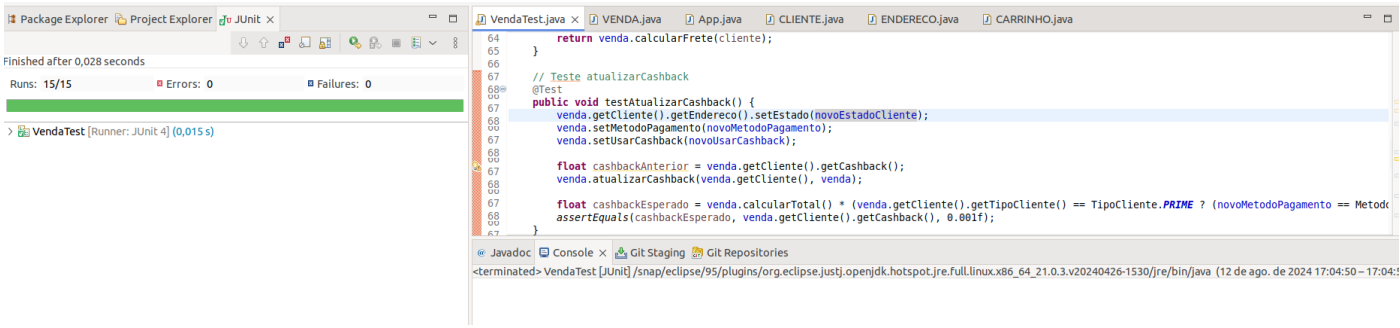
```java
package System;

import TiposDeDados.Estado;

public class App {
    public static void main(String[] args) {
        // Criação de endereços
        ENDERECO endereco1 = new ENDERECO(Estado.SP, true);
        ENDERECO endereco2 = new ENDERECO(Estado.MG, false);
        ENDERECO endereco3 = new ENDERECO(Estado.DF, true);

        // Criação de clientes
        CLIENTE cliente1 = new CLIENTE(TipoCliente.PADRAO, endereco1);
        CLIENTE cliente2 = new CLIENTE(TipoCliente.ESPECIAL, endereco2);
        CLIENTE cliente3 = new CLIENTE(TipoCliente.PRIME, endereco3);
```

Javadoc | Console | Git Staging | Git Repositories

```
<terminated> New_configuration (1) [Java Application] /snap/eclipse/95/plugins/org.eclipse.j
Detalhes da Venda 1: Venda{dataVenda=2024-08-12, cliente=CLIENTE [tipoCliente=
Total da Venda 1: 102.799995

Detalhes da Venda 2: Venda{dataVenda=2024-08-12, cliente=CLIENTE [tipoCliente=
Total da Venda 2: 43.8

Detalhes da Venda 3: Venda{dataVenda=2024-08-12, cliente=CLIENTE [tipoCliente=
Total da Venda 3: 94.4

Cashback do Cliente 1 após venda: 0.0
Cashback do Cliente 2 após venda: 0.0
Cashback do Cliente 3 após venda: 4.7200003

Detalhes da Venda 4: Venda{dataVenda=2024-08-12, cliente=CLIENTE [tipoCliente=
Total da Venda 4: 113.28
Cashback do Cliente 3 após venda: 0.0
```

  Como descrito na etapa 5, com a classe nova em mãos (ENDERECO), é necessário acessar os recursos anteriores por meio de sua instância em um objeto da classe fonte (FONTE). Sendo assim, foi necessário criar 3 novos objetos (endereco1, endererco2, endererco3) para poder construir um objeto do tipo CLIENTE. Fora isso, a compilação continua sem apontar erros.

- Etapa 7 – executando suíte de testes



Como evidenciado pela imagem acima, a suíte de teste continua verde. Somado a isso, é importante ressaltar mais uma vez que, conforme o tópico 5, a forma de acessar os recursos anteriormente exclusivos de CLIENTE se dá por meio do atributo endereco do tipo ENDERECO. Isso pode ser evidenciado na linha 67, onde o trecho de código ***venda.getCliente().setEstado(novoEstadoCliente)*** teve de ser adaptado por meio da chamada ***getEndereco()*** após o ***getCliente()***, para que possa assim alterar o valor do atributo estado, pois ele é declarado na classe **ENDERECO**, e não mais na classe **CLIENTE**.

3. Substituir método por objeto-método

Classe: VENDA

Método: calcularTotal()

Introdução: a técnica de refatoração "Substituir Método por Objeto-Método" é usada quando um método em uma classe se torna muito complexo ou extenso, a ponto de ser difícil de entender, manter ou modificar. Ao aplicar essa técnica, você transforma o método em uma classe separada, chamada de objeto-método. Essa nova classe encapsula o método original e quaisquer dados associados, permitindo que o comportamento seja dividido em vários métodos menores dentro da nova classe.

Aplicando a técnica de refatoração: diante do exposto na introdução desse tópico, o método **calcularTotal** da classe **VENDA** é um forte candidato à essa técnica de refatoração devido a sua complexidade, o qual torna difícil modificá-lo, por exemplo. Sendo assim, confira as etapas de aplicação da técnica em questão a seguir:

- Etapa 1 – criando uma classe (objeto-método) cujo nome reflete o propósito do método original: **CALCULADORADETOTAL**. (Por razões de segurança, relacionamento 1 para 1 com a classe de origem)

```
package System;

public class CALCULADORADETOTAL {

}
```

- Etapa 2 – trazendo os dados, como variáveis locais, utilizados pelo método na classe anterior para a nova classe

Os dados utilizados pelo método CalcularTotal são todos aqueles disponíveis por uma venda, logo basta adicionar um atributo do tipo VENDA. Observe que o construtor da classe já foi criado em seguida para tornar a refatoração mais dinâmica.

```java
package System;

public class CALCULADORADETOTAL {
    private VENDA venda;

    public CALCULADORADETOTAL(VENDA venda) {
        this.venda = venda;
    }
}
```

- Etapa 3 – movendo a lógica do método para a nova classe

Esta etapa consiste em trazer a lógica do método na classe original como um novo método na nova classe.

```java
public class CALCULADORADETOTAL {
    private VENDA venda;

    public CALCULADORADETOTAL(VENDA venda) {
        this.venda = venda;
    }

    public float calcular() {
        // Calcular o total dos itens no carrinho
        float totalCarrinho = venda.getItensVendidos().calcularTotal();

        // Calcular o valor do frete
        float frete = venda.calcularFrete(venda.getCliente());

        // Calcular o ICMS e o Imposto Municipal
        float icms = totalCarrinho * venda.calcularICMS(venda.getCliente());
        float impostoMunicipal = totalCarrinho * venda.calcularImpostoMunicipal(venda.getCliente());

        // Aplicar o cashback se necessário
        float cashbackAplicado = 0;
        if (venda.isUsarCashback() && venda.getCliente().getCashback() > 0) {
            cashbackAplicado = Math.min(venda.getCliente().getCashback(), totalCarrinho);
            venda.getCliente().setCashback(venda.getCliente().getCashback() - cashbackAplicado);
            totalCarrinho -= cashbackAplicado;
        }

        // Calcular o total da venda
        float totalVenda = totalCarrinho + frete + icms + impostoMunicipal;

        return totalVenda;
    }


}
```

- Etapa 4 – dividindo lógica se necessário

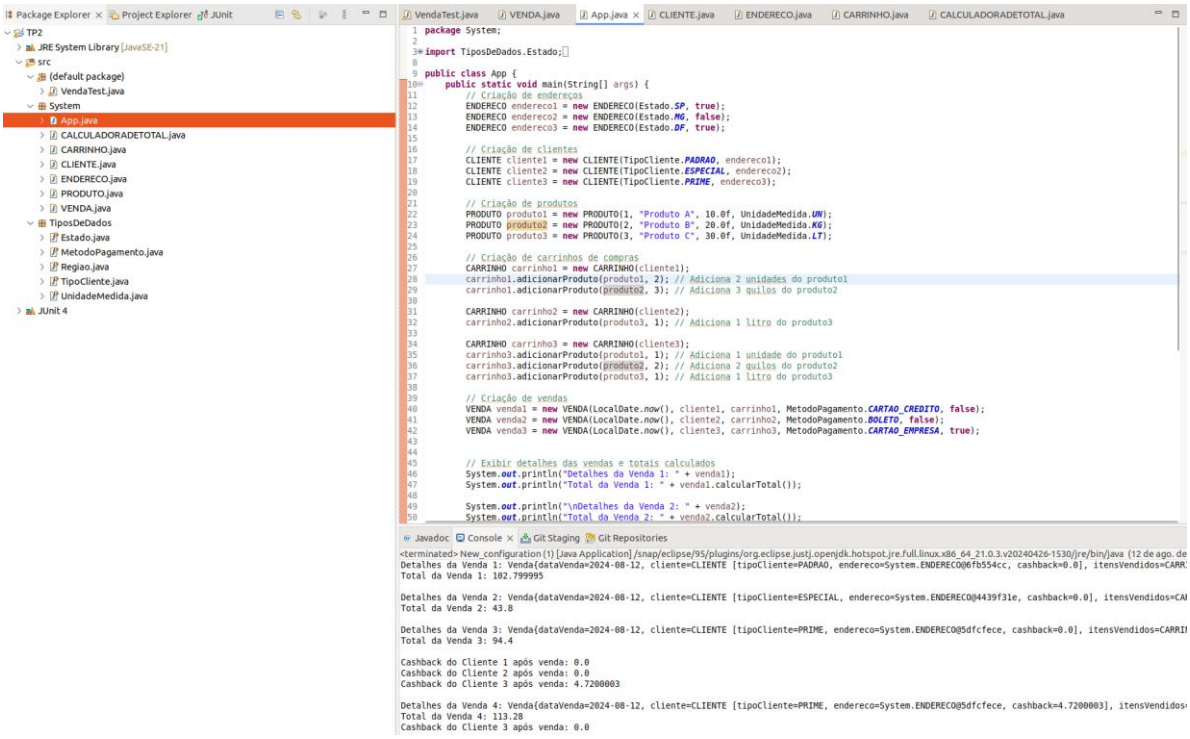
Ao analisar o método alvo mais a fundo, percebe-se que a lógica responsável por **aplicar o cashback de acordo com a necessidade** pode ser modularizada. Sendo assim, a primeira técnica de refatoração evidenciada nesse estudo (**Extrair Método**) pode ser realizada novamente. Por fim, após a aplicação dessa técnica, tem-se o seguinte resultado:

```java
public float calcular() {
    // Calcular o total dos itens no carrinho
    float totalCarrinho = venda.getItensVendidos().calcularTotal();

    // Calcular o valor do frete
    float frete = venda.calcularFrete(venda.getCliente());

    // Calcular o ICMS e o Imposto Municipal
    float icms = totalCarrinho * venda.calcularICMS(venda.getCliente());
    float impostoMunicipal = totalCarrinho * venda.calcularImpostoMunicipal(venda.getCliente());

 // Aplicar o cashback se necessário
    totalCarrinho = aplicarCashback(totalCarrinho);

    // Calcular o total da venda
    float totalVenda = totalCarrinho + frete + icms + impostoMunicipal;

    return totalVenda;
}

// Novo método extraído para aplicar o cashback
private float aplicarCashback(float totalCarrinho) {
    float cashbackAplicado = 0;
    if (venda.isUsarCashback() && venda.getCliente().getCashback() > 0) {
        cashbackAplicado = Math.min(venda.getCliente().getCashback(), totalCarrinho);
        venda.getCliente().setCashback(venda.getCliente().getCashback() - cashbackAplicado);
        totalCarrinho -= cashbackAplicado;
    }
    return totalCarrinho;
}
```

- Etapa 5 – transformando o método original na classe fonte em uma chamada do novo método correspondente na nova classe

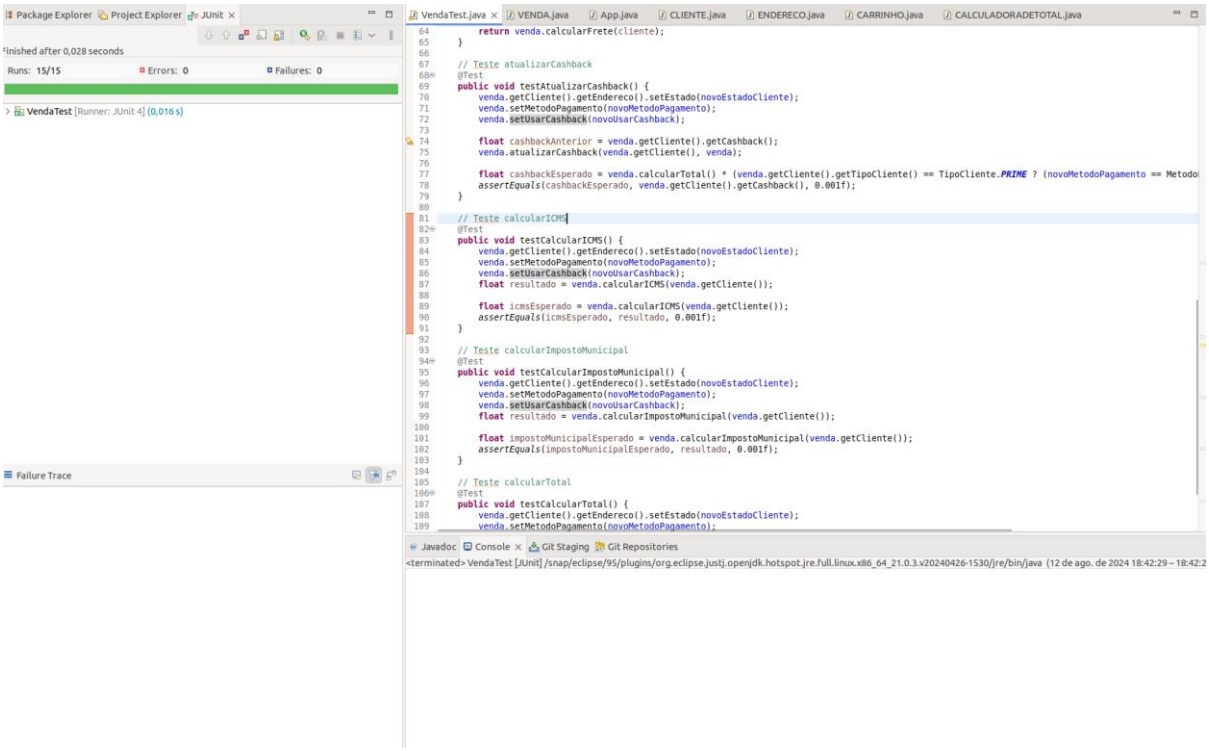
```java
public float calcularTotal() {
    return new CALCULADORADETOTAL(this).calcular();
}
```

- Etapa 6 – verificando compilação



Compilação executada sem problemas

- Etapa 7 – executando suíte de testes



Suíte de testes continua retornando verde, indicando que o resultado esperado se manteve