UNEB

UNIVERSIDADE DO
ESTADO DA BAHIA

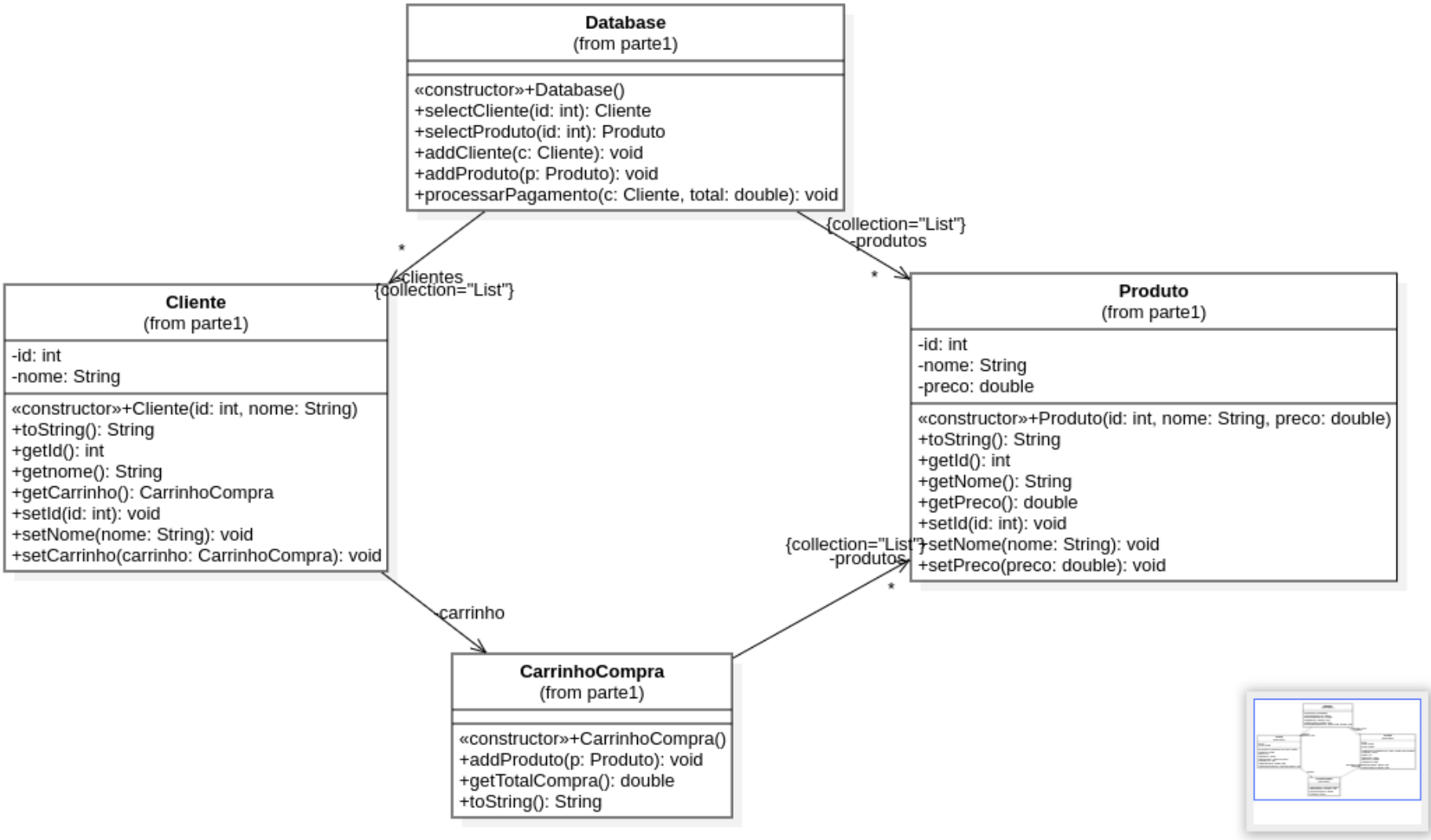
Disciplina: PAS

Docente: Helano

Discente: Adeonita dos Santos

Análise do roteiro 06

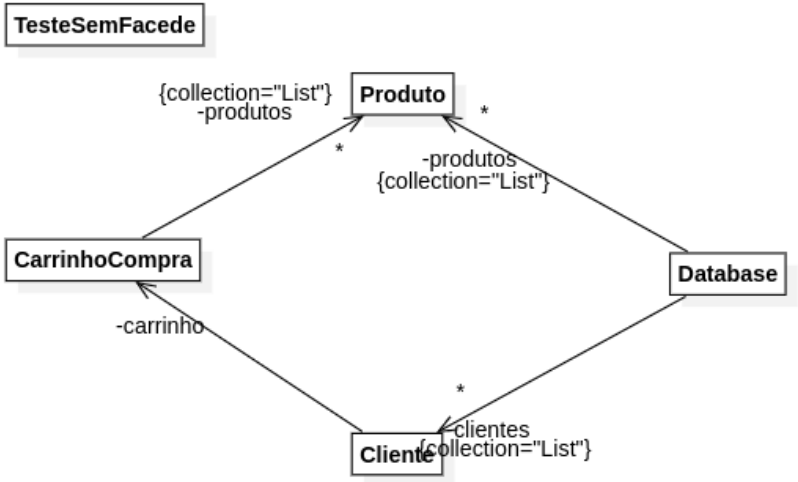
Parte 01:

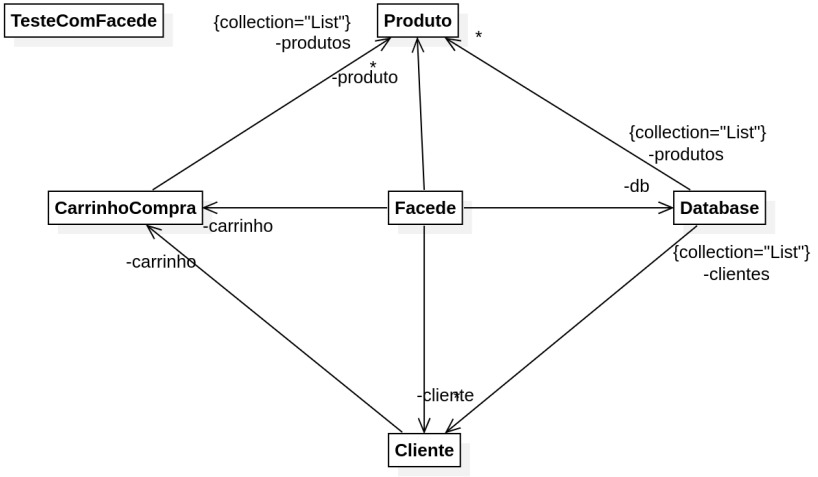

Database
(from parte1)
«constructor»+Database()
+selectCliente(id: int): Cliente
+selectProduto(id: int): Produto
+addCliente(c: Cliente): void
+addProduto(p: Produto): void
+processarPagamento(c: Cliente, total: double): void
{collection="List"}
-produtos
*
*
-clientes
{collection="List"}
Cliente
(from parte1)
-id: int
-nome: String
«constructor»+Cliente(id: int, nome: String)
+toString(): String
+getId(): int
+getnome(): String
+getCarrinho(): CarrinhoCompra
+setId(id: int): void
+setNome(nome: String): void
+setCarrinho(carrinho: CarrinhoCompra): void
Produto
(from parte1)
-id: int
-nome: String
-preco: double
«constructor»+Produto(id: int, nome: String, preco: double)
+toString(): String
+getId(): int
+getNome(): String
+getPreco(): double
+setId(id: int): void
+setNome(nome: String): void
+setPreco(preco: double): void
{collection="List"}
-produtos
*
carrinho
CarrinhoCompra
(from parte1)
«constructor»+CarrinhoCompra()
+addProduto(p: Produto): void
+getTotalCompra(): double
+toString(): String

Parte 02 :


Produto
(from parte2)
-id: int
-nome: String
-preco: double
«constructor»+Produto(id: int, nome: String, preco: double)
+toString(): String
+getId(): int
+getNome(): String
+getPreco(): double
+setId(id: int): void
+setNome(nome: String): void
+setPreco(preco: double): void
{collection="List"}
-produtos
*
-produto
-produtos
{collection="List"}
Database
(from parte2)
«constructor»+Database()
+selectCliente(id: int): Cliente
+selectProduto(id: int): Produto
+addCliente(c: Cliente): void
+addProduto(p: Produto): void
+processarPagamento(c: Cliente, total: double): void
-db
Facede
(from parte2)
«constructor»+Facede()
+registrarCliente(id: int, nome: String): void
+comprar(productId: int, clientId: int): void
+finalizarCompra(clientId: int): void
-carrinho
CarrinhoCompra
(from parte2)
«constructor»+CarrinhoCompra()
+addProduto(p: Produto): void
+getTotalCompra(): double
+toString(): String
-carrinho
{collection="List"}
-clientes
-cliente
Cliente
(from parte2)
-id: int
-nome: String
«constructor»+Cliente(id: int, nome: String)
+toString(): String
+getId(): int
+getnome(): String
+getCarrinho(): CarrinhoCompra
+setId(id: int): void
+setNome(nome: String): void
+setCarrinho(carrinho: CarrinhoCompra): void

Faça uma análise comparativa entre as classes TesteSemFacade e TesteComFacade. Avalie vantagens e desvantagens. Avalie possíveis impactos no caso de uma mudança na regra de negócio do sistema.





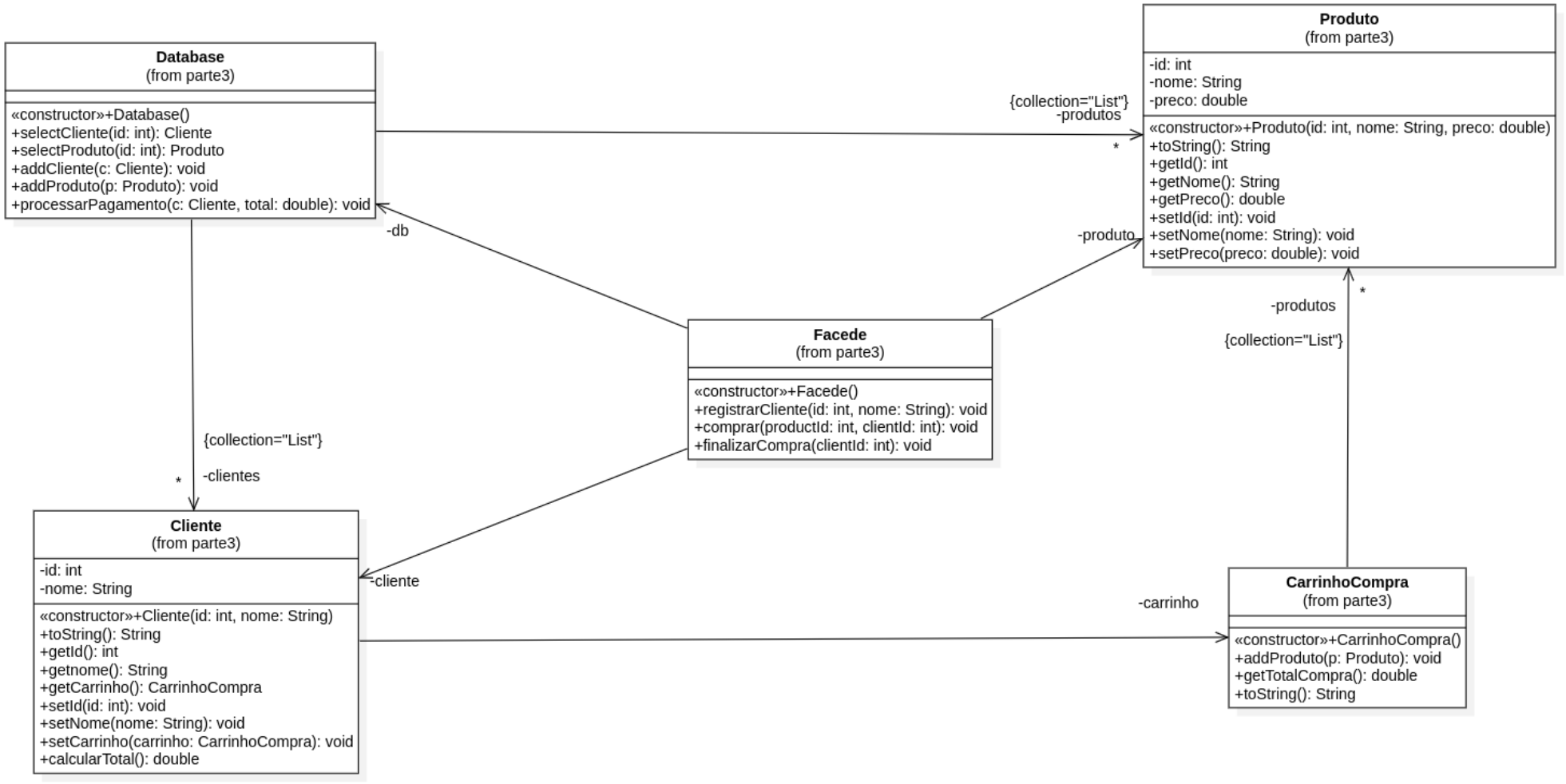

```java
public class TesteSemFacede
{
    public static void main(String args[])
    {
        Database db = new Database();

        Cliente cliente01 = new Cliente(1, "Jose");
        db.addCliente(cliente01);

        CarrinhoCompra carrinho = new CarrinhoCompra();
        cliente01.setCarrinho(carrinho);

        Produto p01 = db.selectProduto(1);
        Produto p02 = db.selectProduto(2);

        cliente01.getCarrinho().addProduto(p01);
        cliente01.getCarrinho().addProduto(p02);

        double total = cliente01.getCarrinho().getTotalCompra();
        db.processarPagamento(cliente01, total);
    }
}
```

```java
public class TesteComFacede
{
    public static void main(String[] args)
    {
        Facede facede = new Facede();

        facede.registrarCliente(222, "Jose");

        facede.comprar(1, 222);
        facede.comprar(2, 222);
        facede.finalizarCompra(222);
    }
}
```

Na classe de TesteSemFacede podemos observar que é necessário fazermos a utilização de diversas classes (Database, CarrinhoCompra, Cliente e Produto) para conseguir finalizar uma compra.

Já a classe de TesteComFacede podemos observar a utilização de uma única classe que é a Facede, pois a mesma encapsula todos os métodos das outras classes dentro dela, tornando possível acessá-los a partir da mesma.

Parte 3


Database
(from parte3)
«constructor»+Database()
+selectCliente(id: int): Cliente
+selectProduto(id: int): Produto
+addCliente(c: Cliente): void
+addProduto(p: Produto): void
+processarPagamento(c: Cliente, total: double): void
Produto
(from parte3)
-id: int
-nome: String
-preco: double
«constructor»+Produto(id: int, nome: String, preco: double)
+toString(): String
+getId(): int
+getNome(): String
+getPreco(): double
+setId(id: int): void
+setNome(nome: String): void
+setPreco(preco: double): void
{collection="List"}
-produtos
*
-db
-produto
Facede
(from parte3)
«constructor»+Facede()
+registrarCliente(id: int, nome: String): void
+comprar(productId: int, clientId: int): void
+finalizarCompra(clientId: int): void
-produtos
{collection="List"}
*
{collection="List"}
-clientes
*
Cliente
(from parte3)
-id: int
-nome: String
«constructor»+Cliente(id: int, nome: String)
+toString(): String
+getId(): int
+getnome(): String
+getCarrinho(): CarrinhoCompra
+setId(id: int): void
+setNome(nome: String): void
+setCarrinho(carrinho: CarrinhoCompra): void
+calcularTotal(): double
-cliente
-carrinho
CarrinhoCompra
(from parte3)
«constructor»+CarrinhoCompra()
+addProduto(p: Produto): void
+getTotalCompra(): double
+toString(): String

4 – Faça uma pequena pesquisa e registre aqui o seu entendimento sobre Coesão x Acoplamento

A coesão tem uma relação com os princípios da letra S(Single Responsability) do Solid definidos por Uncle Bob, ou seja, uma classe coesa é aquela possui apenas uma responsabilidade. Ou seja, ela não deve assumir responsabilidades que não são suas.

O acoplamento tem relação direta com a dependência. Quanto mais uma classe depende da outra, mais fortemente acoplada essa classe é.

5 – Faça uma pequena pesquisa sobre Lei de Demetrio (Lei de Demeter) ou Princípio do Conhecimento mínimo e registre aqui o seu entendimento.

A Lei de Demeter tem relação direta com o encapsulamento, ela define que apenas os donos da informação possam manipulá-la.
Ou seja, uma classe não deve manipular nem ter acesso a propriedades de outras classes.

6 – Qual associação você faz sobre a evolução do nosso projeto e as suas pesquisas feitas nas questões 4 e 5 ? Estes princípios e práticas de engenharia de software foram aplicados ? Como ?

Os princípios de coesão são aplicados a cada classe ser responsável apenas pelo que cabe a elas.

Acoplamento: Quando nós utilizamos uma classe Produto dentro da classe CarrinhoDeCompras por exemplo, nós poderíamos ter utilizado uma interface para não vincular a dependência a uma classe concreta.

Lei de Demeter: Na etapa 01 do roteiro 06 nós não utilizamos a lei de Demeter,

```
cliente01.getCarrinho().addProduto(p01);
cliente01.getCarrinho().addProduto(p02);
```

Na imagem acima é possível observar que o cliente01 conhece detalhes da implementação do carrinho de compras, o encapsulamento é quebrado pois é necessário chamar o getCarrinho e em seguida o addProduto.

Já com o uso da fachada nós conseguimos encapsular a chamada ao método addProduto dentro do método comprar, abstraindo detalhes da implementação.

```
public void comprar(int productId, int clientId)
{
    this.cliente = db.selectCliente(clientId);
    this.produto = db.selectProduto(productId);

    this.cliente.getCarrinho().addProduto(this.produto);
}
```

4- O resultado saiu como esperado conforme o Padrão Singleton descreve? Sim/Não e porquê?

Não, porque embora o método `InocenteSingleton` seja estático ao ser executado ele está sempre retornando uma nova instância da classe.

7- Qual é a sua análise sobre os resultados apresentados para as variáveis n1, n2, n3 e n4 ?

Bom, as instâncias das classes `InocenteSinleton` (n1, n2) ao realizarem a chamada pro método `getInstance` não fazem a checagem se há uma instância pré-existente então serem executados o mesmo sempre retornará uma nova instância, o que faz a comparação retornar o resultado **Instâncias diferentes.**

Já para as instâncias das classes `LazySingleton` (n3,n4) o retorno da comparação é **Instâncias Iguais** , pois antes de realizar a criação de uma nova instância o método `getInstance` checa se não há uma instância existente. Caso haja ele a retorna, caso não haja ele a cria.

10 – Qual dos padrões (Lazy Singleton ou Eager Singleton) foi utilizado no cenário do controlador de vôo do

pacote 2 ?

Eager Singleton