

PROGRAMAÇÃO 2
CIÊNCIA DA COMPUTAÇÃO
MATHEUS RYAN DA SILVA NASCIMENTO

RELATÓRIO DE PROGRAMAÇÃO 2 - WEPAYU

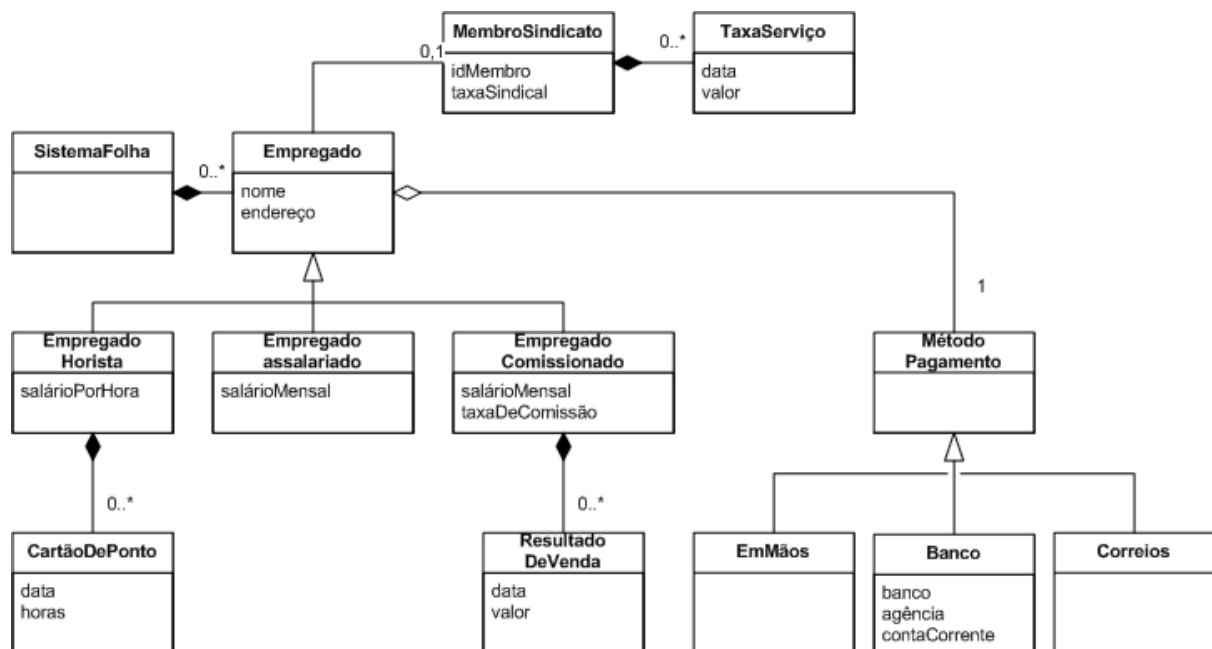
MACEIÓ/AL
2024

SUMÁRIO

SUMÁRIO.....	2
1. INTRODUÇÃO.....	3
2. DESIGN ARQUITETURAL DO SISTEMA.....	3
2.1. Exceptions.....	4
2.2. Models.....	4
2.3. Services.....	4
2.4. DAOs (Data Access Objects).....	4
3. PRINCIPAIS COMPONENTES.....	4
3.1 EMPREGADOS.....	4
3.2 DBMANAGER.....	5
3.3 UTILS.....	6
3.4 VALIDATE.....	6
4. PADRÕES DE PROJETO.....	7
4.1. FACADE.....	7
4.2. DECORATOR.....	8
4.3. FACTORY METHOD.....	9
4.4 SINGLETON.....	10
4.5. DAO.....	11
4.6. MEMENTO.....	12
5. REFERÊNCIAS.....	14

1. INTRODUÇÃO

O WePayU é um sistema simples de pagamento de funcionários, o sistema permite que o mesmo ocorra para diferentes tipos de funcionários, como comissionado, horista e assalariado. Permite a manipulação dos empregados do sistema com a capacidade de adicionar, remover, alterar e visualizar os mesmos, além de ter funcionalidades como lançamento de taxas, impostos e serviços relacionados à geração das folhas de pagamento. O projeto foi desenvolvido usando Desenvolvimento Orientado a Testes. Foi utilizado um arquivo do tipo XML para fazer a simulação do banco de dados do sistema e garantir que o projeto atende ao requisito de persistência de dados.



2. DESIGN ARQUITETURAL DO SISTEMA

Por meio da *Facade* pudemos organizar o sistema de maneira centralizada, dessa forma os subsistemas do projeto ficaram simplificados e unificados. Para garantir uma boa manutenção, escalabilidade e reutilização do projeto, o mesmo foi organizado por camadas tendo em vista as diferentes funções e responsabilidades das classes. As camadas que foram criadas para interagir com a *Facade* são principalmente: *exceptions*, *models*, *services*, e *DAOs*.

2.1. *Exceptions*

Para lidar com todas as exceções de forma centralizada e facilitar a manutenção foi criado *Exceptions*. Ela lida com todas as situações que envolvem algum tipo de exceção gerada por alguma situação inesperada na execução do sistema.

2.2. *Models*

Os modelos representam basicamente as principais classes do projeto, é responsável por estruturar os empregados e outros objetos relacionados ao mesmo. Foi criado inicialmente com base no diagrama de classes exibido pelo professor e posteriormente modificado conforme os avanços relacionados às User Stories.

2.3. *Services*

Os serviços são responsáveis pelo gerenciamento do sistema, lidar com as constantes que representam as configurações do sistema, garantir um histórico dos estados do sistema, ter um construtor para gerar as folhas de pagamento, lidar com os métodos com funcionalidades ligadas a verificação e tratamento da saída e entrada de dados. Os serviços também são responsáveis pelo controle da persistência de dados por meio de um arquivo XML.

2.4. *DAOs (Data Access Objects)*

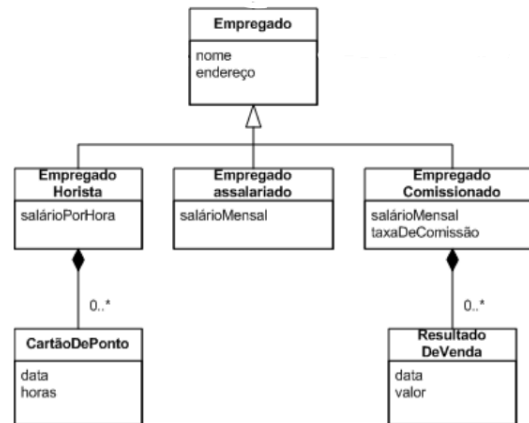
Para representar as estruturas de gerenciamento dos modelos utilizamos os DAOs. Eles podem se comunicar com o repositório local e garantem a segurança na manipulação e manutenção dos objetos, como por exemplo: criação, alteração e remoção.

3. PRINCIPAIS COMPONENTES

3.1 EMPREGADOS

A classe Empregados representa a entidade chave do sistema de pagamento. Sendo este herdado por outras 3 classes: Empregado Horista,

Empregado Assalariado e Empregado Comissionado. Além do mais, é, também, uma classe abstrata, pois há somente os 3 tipos de empregados citados, não havendo um “empregado”. A praxe disso se dá por meio do princípio de OO da Abstração, seguindo pela Herança, já que os demais tipos de empregados herdam da classe Empregado.



3.2 DBMANAGER

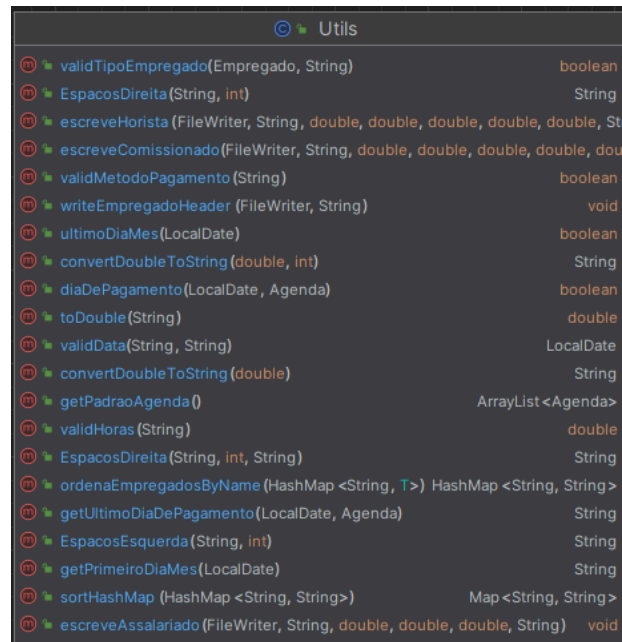
É a classe responsável pelo gerenciamento do banco de dados do sistema, não obstante, gerencia também a persistência do sistema. Aplicada ao padrão Singleton, há somente uma instância de DBManager no sistema inteiro.

Principais métodos e atributos de DBManager:

1. **public static HashMap<String, Empregado> carregarEmpregadosDeXML():** Responsável por carregar a persistência toda vez que o sistema é zerado na Facade.
2. **public static DBmanager getDatabase():** Método responsável por levar a instância única da base de dados globalmente a qualquer lugar entre as classes.
3. **public static String add(Empregado e):** Método responsável pela adição de empregados ao Banco de Dados.
4. **public static HashMap<String, Empregado> empregados:** É a estrutura de dados que armazena os empregados no banco de dados, sendo a estrutura central da persistência.

3.3 UTILS

Utils nada mais é do que o cerne algorítmico do sistema, sendo responsável pela parte lógica de cálculos complexos e de validação de atributos. Métodos presentes em Utils:



Utils	
validTipoEmpregado(Empregado, String)	boolean
EspacosDireita(String, int)	String
escreveHorista(FileWriter, String, double, double, double, double, double, String)	
escreveComissionado(FileWriter, String, double, double, double, double, double, String)	
validMetodoPagamento(String)	boolean
writeEmpregadoHeader(FileWriter, String)	void
ultimoDiaMes(LocalDate)	boolean
convertDoubleToString(double, int)	String
diaDePagamento(LocalDate, Agenda)	boolean
toDouble(String)	double
validData(String, String)	LocalDate
convertDoubleToString(double)	String
getPadraoAgenda()	ArrayList<Agenda>
validHoras(String)	double
EspacosDireita(String, int, String)	String
ordenaEmpregadosByName(HashMap<String, T>)	HashMap<String, String>
getUltimoDiaDePagamento(LocalDate, Agenda)	String
EspacosEsquerda(String, int)	String
getPrimeiroDiaMes(LocalDate)	String
sortHashMap(HashMap<String, String>)	Map<String, String>
escreveAssalariado(FileWriter, String, double, double, double, String)	void

3.4 VALIDATE

Surge como a opção para otimização de algoritmos por meio de uma classe que instancia as relações de validação de atributos dos empregados, do banco de dados e das demais classes. Sendo de acesso global e suportando as entradas preteridas pelo usuário.

Alguns dos principais métodos de Validate:

1. **public static void validDuplicateID(String idSindicato):** Método que busca a duplicação de Empregados em um sindicato.
2. **public static void validEmployInfo(String nome, String endereco, String tipo, String salario):** Método que averigua se os atributos de um empregado qualquer estão consistentes.
3. **public static void validSalario(String valor):** Método que avalia se o salário está consistente.
4. **public static void validsyndicalWarm(String idSindicato, String taxaSindical):** Método que avalia se

o Membro Sindicato está persistente com a taxa sindical e se o ID do membro realmente existe.

5. **public static void validValue(double value)** : Método que avalia se algum valor número é válido.

4. PADRÕES DE PROJETO

4.1. FACADE

Descrição geral: O *Facade* é um padrão de projeto que fornece uma interface simplificada para um conjunto complexo de classes. Se consiste em comprimir várias classes e ocultar a complexidade real do sistema, oferecendo uma interface mais amigável para o cliente.

Problema resolvido: É capaz de lidar com problemas que envolvem subsistemas complexos enquanto o usuário precisa de uma interação fácil e simples. Dito isso, ele apresenta uma fachada para esconder as complexidades e garantir um acesso fácil às funcionalidades dos subsistemas.

Identificação da oportunidade e aplicação no projeto: O escopo do projeto por ser grande e envolver diversos modelos para administrar e vários processos de validação de dados gera o ambiente perfeito para o uso de uma estrutura capaz de simplificar a complexidade. Nesse caso a *Facade* faz isso além de facilitar o acesso aos processos do sistema, contendo métodos que atendem os requisitos funcionais do sistema como: criação, alteração, leitura e remoção de empregado e de seus dados, e da geração da folha de pagamento.

Alguns métodos da Facade:

1. **public String criarEmpregado(String nome, String endereco, String tipo, String salario)**
2. **public String getAtributoEmpregado(String emp, String atributo)**
3. **public void lancaCartao(String emp, String data, String horas)**
4. **public void lancaVenda(String emp, String data, String valor)**

Métodos esses que, como já supracitado, facilitam a comunicação com o sistema propriamente dito. Respectivamente, temos os métodos que criam empregados, buscam atributos destes empregados, cria Cartão de Ponto para `EmpregadoHorista` e cria venda para `EmpregadoComissionado`.

4.2. DECORATOR

Descrição geral: O decorator é um padrão que permite a inserção de novos comportamentos para objetos ao inseri-los em invólucros de objetos que contém os comportamentos.

Problema resolvido: Com este padrão de projeto pudemos moldar o código de forma que sempre que necessário poderíamos estender o comportamento de objetos de forma dinâmica, sendo crucial para adicionar e remover funcionalidades de forma independente e modular.

Identificação da oportunidade e aplicação no projeto: Havia momentos em que seria conveniente adicionar novas funcionalidades a objetos existentes de forma dinâmica e flexível e devido a facilidade de reutilização e manutenção do Decorator isso era capaz. Uma analogia rápida seria como adicionar e remover agasalhos de frio ou capas de chuva de pessoas conforme elas precisassem. As classes que usaram Decorator foram de `EmpregadoHorista`, `EmpregadoAssalariado` e `EmpregadoComissionado`, respectivamente com `Hora`, `Service` e `Venda`.

Exemplificação do decorator de `EmpregadoComissionado` com a extensão `Venda` na classe `VendaDao`:

```
EmpregadoComissionado empregado = (EmpregadoComissionado) e;  
Venda venda = new Venda(data, valorFormato, empregado);
```

É demonstrado que o `EmpregadoComissionado` recebe uma venda sempre que o decorator `Venda` é criado, constando a data de inicio, o valor e o

empregado que sofrerá a adição. O mesmo pode ser observado para os outros decoradores nas classes `TaxaDao` e `CartaoDao`.

4.3. FACTORY METHOD

Descrição geral: O Factory Method é um padrão de projeto que fornece uma interface para criar objetos em uma superclasse ainda permitindo que as subclasses alterem o tipo de objetos que serão criados. Ele também encapsula a lógica de criação de objetos e é responsável por organizar a responsabilidade de instanciar objetos para suas subclasses.

Problema resolvido: Para evitar que o código fique completamente dependente de uma só classe, o que iria enormemente dificultar futuras alterações, e evitar complicações geradas ao criar famílias de objetos relacionados. Nós podemos usar este padrão de projeto para criar um método fábrica e criar os objetos a partir dele, esses objetos podem ter métodos em comum mas os mesmos serão executados de maneiras diferentes conforme necessários.

Identificação da oportunidade e aplicação no projeto: Por ter muitas classes com funcionalidades similares e exigências em comum, o uso do *Factory* dá a possibilidade de criar objetos com funcionalidades em comum mas sem o risco de qualquer mudança afetar outras partes do código. A Factor Method foi utilizada para a criação de empregados, inibindo a necessidade de usar um construtor distinto para criar diferentes tipos de empregados.

Segue a demonstração da classe `FactoryEmpregados` para a criação de empregados.

```
public class FactoryEmpregados {  
    4 usages  ↕ Matheus Ryan  
    public Empregado makeEmployee(String nome, String endereco, String tipo, double salario) throws Exception {  
        switch (tipo){  
            case "horista" -> {  
                Agenda ag = new Agenda(Settings.PADRAO_HORISTA);  
                return new EmpregadoHorista(nome, endereco, ag, salario);  
            }  
            case "assalariado" -> {  
                Agenda ag = new Agenda(Settings.PADRAO_ASSALARIADO);  
                return new EmpregadoAssalariado(nome, endereco, ag, salario);  
            }  
        }  
        return null;  
    }  
}
```

Desta maneira, toda chamada do método `makeEmployee` retornará um empregado horista ou assalariado. Também há o mesmo método para a criação de um `EmpregadoComissionado` que não foi exibido em função do Overfitting de métodos, considerando que para a criação deste haja mais atributos.

4.4 SINGLETON

Descrição geral: O padrão de projeto *Singleton* permite que uma classe tenha apenas uma instância enquanto oferece um ponto de acesso global a essa instância. Ele usualmente está relacionado a criação de uma classe com um método estático que retorna a mesma instância sempre que chamada.

Problema resolvido: Ele resolve problemas relacionados a manutenção de um só estado de acesso a uma classe. Com isso, ele permite um controle de instâncias globais de forma estrita e consegue evitar que a memória se desgaste com a multiplicação de uma informação, que poderia gerar overload e erros de conflito de estados.

Identificação da oportunidade e aplicação no projeto: Os DAOs precisam gerenciar os modelos que necessitam do acesso ao banco de dados para efetuar a leitura e atualização do XML. Graças ao padrão podemos fazer isso mais facilmente devido à criação de uma instância única de acesso ao banco. Em diversas ocasiões no software foi utilizada o padrão Singleton, como por exemplo para criar instâncias únicas de Memento e DBManage.

Demonstração do padrão Singleton:

```
private DBmanager() {
    this.empregados = carregarEmpregadosDeXML();
    this.fabrica = new FactoryEmpregados();
    this.agendas = readAgendas();
}

3 usages new *
public static DBmanager getDatabase() {
    if(session == null)
    {
        session = new DBmanager();
    }
    return session;
}
```

Neste exemplo podemos observar a aplicação do Singleton com a criação de um construtor privado de `DBmanager`. Desta maneira, o método `getDataBase()` será responsável por retornar sempre o mesmo `DBmanager`, acertando com o padrão Singleton e globalizando o banco de dados.

Podemos notar a presença do Singleton também no padrão Memento.

```
private Memento(DBmanager session) { this.session = session; }

1 usage new *
public static Memento getCommand(DBmanager session){
    if(backup == null){
        backup = new Memento(session);
    }
    backup.deleteStacks();
    return backup;
}
```

Desta vez, estaremos usando uma instância única e global do `Memento`, já que o mesmo armazena os estados do sistema.

4.5. DAO

Descrição geral: O DAO fornece uma interface abstrata para lidar com as operações de CRUD (Create, Read, Update, Delete) existentes num banco de dados. Com isso o projeto não precisa da implementação específica do banco de dados criando uma maior independência no código.

Problema resolvido: Por não precisar dos detalhes da implementação do acesso aos dados, o padrão acaba por facilitar a substituição de um sistema de gerenciamento de banco de dados por outro e permite separação da lógica de negócios com a lógica de acesso aos dados.

Identificação da oportunidade e aplicação no projeto: O uso de DAOs é basicamente necessário para o prosseguimento do projeto pois há a necessidade da manipulação de dados (CRUD) e este padrão de projeto facilita bastante o gerenciamento dos objetos em diferentes processos do projeto. Por

ser um padrão para o gerenciamento destes o objetos, a aplicabilidade dele no projeto se deu da seguinte forma: foi criada uma instancia de cada variável do DAO em Manager, assim acessando ela sempre pelo DAO. Segue a demonstração:

```
8 usages  ↕ Matheus Ryan
public EmpregadoDao getEmpregadoDao() {
    if(empregadoDao == null){
        empregadoDao = new EmpregadoDao(session, backup);
    }
    return empregadoDao;
}

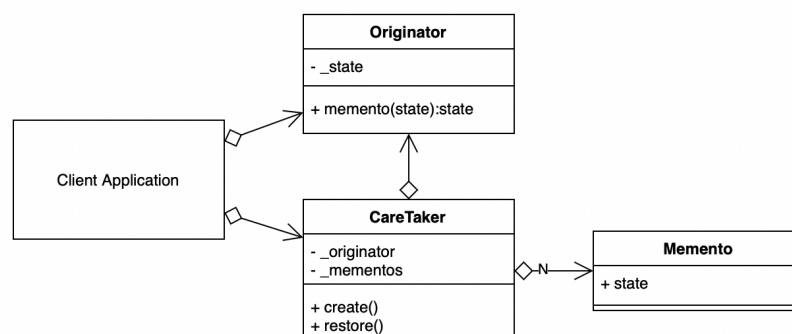
3 usages  ↕ Matheus Ryan
public CartaoDao getCartaoDao() {
    if(cartaoDao == null){
        cartaoDao = new CartaoDao(session, backup);
    }
    return cartaoDao;
}

2 usages  ↕ Matheus Ryan
public VendaDao getVendaDao() {
    if(vendaDao == null){
        vendaDao = new VendaDao(session, backup);
    }
    return vendaDao;
}
```

É trivial enxergar os atributos criados com seus respectivos getters, além do mais, da forma como isso centraliza o acesso destes dados por meio de uma classe DAO.

4.6. MEMENTO

Definição geral: O memento é um padrão que permite que você salve e restaure o estado anterior de um objeto sem revelar os detalhes de sua implementação. Isso ocorre ao encapsular e externalizar o estado interno de um objeto de modo que o objeto possa ser restaurado a esse estado..



Problema resolvido: Ele permite que o estado interno de um objeto possa ser salvo e restaurado sem violar o encapsulamento, e permite a criação de funcionalidades como undo/redo de forma conveniente. Ela também separa a responsabilidade de salvar o estado do objeto da própria classe do objeto, garantindo a coesão e o baixo acoplamento.

Identificação da oportunidade e aplicação no projeto: O projeto exigia a funcionalidade de desfazer e refazer algumas operações (undo/redo) e com isso o *Memento* se tornou crucial para o projeto. Ambas as pilhas foram criadas em uma classe de mesmo nome, tal qual armazenam a *HashMap* de empregados, ou seja, o estado posterior e o anterior. Segue uma demonstração de ambas as pilhas dentro da classe *Memento*.

```
11 usages
private static Stack<HashMap<String, Empregado>> undo; //Pilha de Undo
7 usages
private Stack<HashMap<String, Empregado>> redo; //Pilha de Redo
```

Ademais, as operações de `Push()` da pilha de Undo são observadas em demasiadas partes do código no momento de alteração de estados, ou seja, em métodos que criam, removem, adicionam e fazem alterações na *HashMap* de *DBmanager*. A seguir será demonstrada as funções `PushUndo` e `PushRedo`.

```
public void pushRedo(HashMap<String, Empregado> e) throws Exception {
    if (!systemOn) {
        throw new MementoFazerException();
    }

    if (redo == null) redo = new Stack<>();

    redo.push(e);
}
```

```
public static void pushUndo(HashMap<String, Empregado> e) throws Exception {

    if (!systemOn) {
        throw new CannotMementoException();
    }

    if (undo == null)
        undo = new Stack<>();

    undo.push(e);
}
```

5. REFERÊNCIAS

Memento. Disponível em: <<https://refactoring.guru/design-patterns/memento>>.

Decorator. Disponível em: <<https://refactoring.guru/design-patterns/decorator>>.

Facade. Disponível em: <<https://refactoring.guru/design-patterns/facade>>.

Factory Method Disponível em:
<<https://refactoring.guru/design-patterns/factory-method>>.

Singleton. Disponível em: <<https://refactoring.guru/design-patterns/singleton>>.

BAELDUNG. The DAO Pattern in Java | Baeldung. Disponível em:
<<https://www.baeldung.com/java-dao-pattern>>.

Java Design Patterns - **Example Tutorial** | DigitalOcean. Disponível em:
<<https://www.digitalocean.com/community/tutorials/java-design-patterns-example-tutorial>>.

Design Patterns in Java - Javatpoint. Disponível em:
<<https://www.javatpoint.com/design-patterns-in-java>>.

Patterns - Java Design Patterns. Disponível em:
<<https://java-design-patterns.com/patterns/>>.