

Unidade Federal de Alagoas
Instituto de
Computação (IC)

Disciplina: Programação 2
Professor: Mario Hozano

RELATÓRIO DO PROJETO
WEPAYU
ISSUES 1-8

Emanuele Vitória de Jesus Lima

O projeto **WePayU** integra padrões de projeto clássicos e princípios de **Domain-Driven Design (DDD)**, garantindo modularidade, consistência e facilidade de manutenção.

1. Padrões de Projeto

Padrão	O que é	Implementação no WePayU	Métodos/Classes Principais
Singleton	Garante que uma classe tenha apenas uma instância na aplicação, oferecendo ponto de acesso global.	<code>EmpregadoRepository</code> mantém todos os empregados e garante que todos os managers acessem a mesma coleção.	<code>EmpregadoRepository.getInstance()</code>
Facade	Fornece uma interface simplificada para um subsistema complexo, ocultando detalhes internos.	<code>Facade.java</code> centraliza chamadas de todos os managers. A <code>MainManager</code> também funciona como Facade.	<code>Facade.criarEmpregado()</code> , <code>Facade.rodaFolha()</code> , <code>Facade.removerEmpregado()</code>

Strategy	Permite encapsular algoritmos intercambiáveis, escolhidos em tempo de execução.	Tipos de empregados (<code>EmpregadoHorista</code> , <code>EmpregadoAssalariado</code> , <code>EmpregadoComissionado</code>) e métodos de pagamento implementam estratégias de cálculo de salário.	<code>FolhaPagamentoManager.calcularPagamentoCompleto()</code>
Template Method	Define o esqueleto de um algoritmo na superclasse, permitindo que subclasses implementem partes específicas.	Método <code>copiaAtributos()</code> na classe <code>Empregado</code> define a lógica base de cópia de atributos; subclasses implementam <code>clone()</code> chamando <code>super.copiaAtributos()</code> .	<code>Empregado.copiaAtributos()</code> , <code>EmpregadoHorista.clone()</code>
Memento	Captura e armazena o estado interno de um objeto para restaurá-lo posteriormente, usado em undo/redo.	<code>EmpregadoRepository.Memento</code> salva snapshots do estado; <code>CommandHistoryManager</code> mantém pilhas de <code>undo</code> e <code>redo</code> .	<code>CommandHistoryManager.undo()</code> , <code>CommandHistoryManager.redo()</code>

2. Domain-Driven Design (DDD)

Conceito	Aplicação no WePayU	Exemplos no Código
Linguagem Ubíqua	Termos consistentes do domínio de negócio no código	<code>lancaCartao()</code> , <code>getHorasNormaisTrabalhadas</code> , <code>FolhaPagamentoManager</code>
Entidades (Entities)	Objetos com identidade única e ciclo de vida próprio	<code>Empregado</code> , <code>CartaoDePonto</code> , <code>ResultadoVenda</code>
Objetos de Valor (Value Objects)	Representam conceitos descritivos sem identidade própria	<code>MetodoPagamento</code> e subclasses (<code>EmMaos</code> , <code>Banco</code>)

Agregados (Aggregates)	Agrupa entidades relacionadas garantindo consistência	<code>Empregado</code> como raiz, gerenciando <code>CartaoDePonto</code> , <code>ResultadoVenda</code> , <code>TaxaDeServico</code>
Repositórios (Repositories)	Abstraem a persistência, permitindo que a lógica de domínio opere em memória	<code>EmpregadoRepository</code> para armazenamento e recuperação em XML

- `EmpregadoRepository.getInstance()` garante que todo o sistema trabalhe com a mesma coleção de empregados, evitando inconsistência de dados.
- `Facade.criarEmpregado()` e `Facade.removerEmpregado()` simplificam a comunicação com o sistema, evitando que a camada de apresentação precise conhecer a complexidade interna dos managers.
- `FolhaPagamentoManager.calcularPagamentoCompleto()` decide automaticamente como calcular o salário com base no tipo de empregado, aplicando as estratégias corretas.
- `Empregado.copiaAtributos()` e `EmpregadoHorista.clone()` permitem criar cópias de empregados sem repetir lógica, garantindo que atributos comuns sejam preservados.
- `CommandHistoryManager.undo()` e `redo()` permitem desfazer ou refazer operações críticas, aumentando a segurança e confiabilidade do sistema.
- Todos os métodos relacionados a **lançamentos de cartão, vendas e taxas** (`lancaCartao()`, `lancaVenda()`, `lancaTaxaServico()`) mantêm o domínio do negócio consistente, funcionando dentro do agregado `Empregado`.

Um Pouco mais Sobre os Padrões

1. Padrão Singleton

O que é: O padrão Singleton garante que uma classe tenha apenas uma única instância em toda a aplicação. Isso é útil para classes que gerenciam recursos compartilhados, pois evita a criação de múltiplas instâncias que poderiam levar a estados inconsistentes.

- **No código:** A classe `EmpregadoRepository` implementa o Singleton. Ela é responsável por manter e persistir o estado de todos os empregados do sistema. Ao garantir que apenas uma instância de `EmpregadoRepository` exista, o sistema garante que todos os outros componentes (como os "managers") acessem a mesma coleção de empregados, evitando problemas de sincronização e dados duplicados. O construtor é privado, e o método estático `getInstance()` é o único ponto de entrada para obter a instância.

2. Padrão Facade

O que é: O padrão Facade oferece uma interface simplificada e de alto nível para um sistema complexo de subsistemas. Ele "esconde" a complexidade de múltiplos objetos e suas interações, tornando o sistema mais fácil de usar e de entender.

- **No código:** A classe `Facade` é a principal representação desse padrão. Em vez de a camada de apresentação ter que interagir com `EmpregadoManager`, `FolhaPagamentoManager`, etc., ela se comunica apenas com a classe `Facade`. A `Facade` delega as chamadas de método para os "managers" apropriados, como `criarEmpregado()`, `removerEmpregado()`, e `rodaFolha()`, simplificando a API para o cliente. A classe `MainManager` também age como um `Facade`, agrupando os diferentes `Managers` em um único ponto de acesso.

3. Padrão Strategy

O que é: O padrão Strategy permite definir uma família de algoritmos, encapsular cada um deles e torná-los intercambiáveis. O cliente pode escolher o algoritmo a ser usado em tempo de execução sem alterar a estrutura da classe que o utiliza.

- **No código:** Este padrão é aplicado na modelagem dos tipos de empregados e métodos de pagamento. A classe abstrata `Empregado` e `MetodoPagamento` definem o contrato comum. As classes concretas, como `EmpregadoHorista`, `EmpregadoAssalariado` e `EmpregadoComissionado`, fornecem implementações específicas (estratégias) para como o salário é calculado ou como os atributos são tratados. A lógica de folha de pagamento, por exemplo, utiliza o tipo de empregado para determinar a forma de cálculo adequada.

4. Padrão Template Method

O que é: O padrão Template Method define o esqueleto de um algoritmo em uma superclasse, deixando que as subclasses forneçam a implementação para algumas etapas sem modificar a estrutura geral do algoritmo.

- **No código:** O método `copiarAtributos()` na classe `Empregado` é um exemplo. Ele define a sequência de passos para copiar atributos genéricos de um empregado para outro. As classes filhas, ao implementarem o método `clone()` para criar cópias de si mesmas, chamam `super.copiarAtributos(clone)` para reutilizar a lógica comum e, em seguida, adicionam a lógica específica de clonagem de seus próprios atributos, sem modificar a estrutura da cópia dos atributos base.

5. Padrão Memento

O que é: O padrão Memento é usado para capturar e armazenar o estado interno de um objeto em um "memento" para que o objeto possa ser restaurado para esse estado posteriormente. Ele é frequentemente utilizado em sistemas que necessitam de funcionalidades de "desfazer" ou "refazer".

- **No código:** A classe interna `EmpregadoRepository.Memento` age como o memento, armazenando um snapshot do estado do repositório. O `EmpregadoRepository` é o objeto que pode ser salvo e restaurado. O `CommandHistoryManager` é o `Caretaker`, que gerencia as pilhas de mementos (`undoStack` e `redoStack`). Para cada comando executado, um memento é criado e adicionado à pilha de undo, permitindo que o estado anterior seja restaurado.

6. Domain-Driven Design (DDD)

O que é: O Domain-Driven Design é uma abordagem de desenvolvimento de software que se concentra na modelagem do domínio do negócio. Ele promove uma estrutura de código que reflete o mundo real, separando claramente as responsabilidades.

- **No código:** A organização dos pacotes demonstra uma arquitetura DDD. O pacote `models` representa o núcleo do domínio, com entidades como `Empregado` e seus subtipos. O pacote `managers` atua como a camada de serviço, orquestrando as operações do domínio. Por fim, o `EmpregadoRepository` abstrai a camada de persistência de dados, um pilar central do DDD, que desacopla o domínio dos detalhes de como os dados são armazenados.

Em resumo, **cada método é cuidadosamente desenhado para cumprir seu papel no domínio do sistema**, e os padrões de projeto que utilizei garantem modularidade, extensibilidade e clareza no código. Com isso, o WePayU não apenas processa folhas de pagamento, mas também mantém integridade, flexibilidade e facilidade de manutenção.