UNIVERSIDADE FEDERAL
DE ALAGOAS

Universidade Federal de Alagoas
Instituto de Computação

Rafael Luciano Lima da Silva
Vinícius da Costa Neitzke

Relatório do Projeto WePayU

Milestone 2

Maceió
2024

Design arquitetural

O design arquitetural do sistema é formado por quatro camadas:

Camada de Lógica de Negócios

Esta camada contém a lógica central do sistema, sendo responsável por definir a manipulação de todas as operações e dados relacionados aos empregados, tais como seu tipo ou métodos de pagamento.

Dentro do diretório “br/ufal/ic/p2/wepayu”, a classe Facade – que utiliza um padrão de projeto de mesmo nome – faz parte desta camada, fornecendo uma interface menos complexa para interagir com o sistema, permitindo a execução de todos os métodos presentes nele.

No mesmo diretório temos a classe Sistema, que inclui a lógica de adição e remoção de empregados no sistema, utilizando a estrutura de dados HashMap. Contém métodos getters e setters para atributos dos empregados, também incluindo a lógica de vendas, presença e pagamento deles.

Também na camada de lógica temos a classe Memento, que guarda um snapshot (cópia) do HashMap de empregados, e a classe History, que contém o histórico de mudanças na forma de duas pilhas, sendo uma para a ação de desfazer determinada operação e outra para refazê-la.

Camada de Acesso a Dados

A camada de acesso a dados é construída na classe Database. Os dados serão armazenados de forma compartilhada no formato XML, persistindo tanto nas etapas de execução dos códigos durante os casos testes quanto após a execução deles. No Milestone 1, a adaptação ao XML se dava por salvar o conteúdo do HashMap dos empregados no arquivo XML logo após o encerramento do sistema, com isso, ao ser construída uma nova Facade, os dados presentes neste arquivo eram carregados. Para o Milestone 2 foram adicionados um atributo de lista de pagamentos à classe, que será utilizada para definir os pagamentos que serão possíveis de realizar no sistema como também um novo método “setData”

responsável pela atribuição na HashMap da classe, sendo usado em caso de ter que recuperar para uma cópia daquela HashMap. No arquivo .xml serão salvos tanto o HashMap de empregados quanto a lista de pagamentos (armazenada na forma de uma lista de String).

Camada de Modelagem de Dados

Dentro da pasta “br/ufal/ic/p2/wepayu/models” estão localizadas as classes que modelam os objetos de domínio do sistema, como Empregado e suas subclasses (EmpregadoAssalariado, EmpregadoComissionado, EmpregadoHorista), além de outras classes como MetodoPagamento, PointCard, ResultadoDeVenda, TaxaServico, etc. Essas classes representam os dados manipulados pelo sistema e integram a camada de modelo.

Camada de Exceções

No diretório “br/ufal/ic/p2/wepayu/Exception”, estão localizadas as classes de exceções personalizadas que lidam com erros específicos do sistema. Essas classes tratam as situações excepcionais e garantem a robustez do projeto. Para o Milestone 2 foi criada uma exceção para a adição de dia de pagamento (“addNewPayday”), que trata todos os possíveis erros relacionados à adição de uma agenda de pagamentos. Também foi criada uma exceção de “undo”, para quando não há nenhuma ação a ser desfeita ou o sistema tenha sido encerrado, o que impossibilita de desfazer algo.

No geral, o trabalho segue uma arquitetura de quatro camadas, com a lógica de negócios sendo o núcleo do sistema e as outras camadas interagindo com ela para oferecer as funcionalidades necessárias.

Principais componentes e suas interações

Antes de apresentar os componentes do projeto, é necessário destacar que o problema em ter utilizado o operador “instanceof” foi solucionado abstraindo o

método “getTipo” na classe Empregado, retornando o tipo de cada empregado nas subclasses dela.

Os principais componentes do sistema são:

Classes do empregado

As classes do empregado são feitas para representar os diferentes tipos de empregados no sistema, onde cada um possui seus próprios atributos e métodos específicos. Na classe pai “Empregado” são definidos os atributos básicos “nome”, “endereço” e “metodoPagamento”, bem como os métodos getters e setters desses atributos e getters e setters abstratos para atributos que são definidos somente de acordo com o tipo de “Empregado”, tais como “salario” e “tipo”. Também é definido um get para a checagem se o membro faz parte do sindicato.

Nas classes filhas “EmpregadoAssalariado”, “EmpregadoComissionado”, “EmpregadoHorista” os métodos abstratos da classe pai são definidos de acordo com os requisitos de cada classe, sendo também definidos outros atributos, junto com seus getters e setters, que estão presentes em apenas um tipo de empregado, como o “salarioPorHora” e o “cartao” no caso do horista, “salarioMensal” para comissionados e assalariados, e “comissao” e “vendas” para os comissionados.

Para o Milestone 2, a classe Empregado recebeu um novo atributo, “agendaPagamento”, responsável em definir a data e a frequência com que o empregado será pago. Para o User Story 8, foi necessário implementar um método para clonar a classe Empregado, já que precisamos salvar diversas versões da HashMap que armazena os empregados.

Classes do pagamento

As classes de pagamento são utilizadas para armazenar os dados relacionados à remuneração dos empregados. Ao ser cadastrado no sistema, o empregado inicia o método de pagamento no estado “em mãos” e pode alterá-lo de acordo com seu desejo. Os métodos de pagamento “em mãos” e “correios” utilizam apenas as informações já pré estabelecidas na criação de um empregado, como seu endereço, assim, não são necessárias desenvolver as classes “EmMaos” e “Correios”. Outro

método de pagamento é pelo banco, para isso, existe a classe Banco, que contém atributos que guardam as informações acerca da conta bancária de um empregado (banco, agência, conta corrente), bem como seus métodos getters e setters.

No milestone 2 foi criada a classe AgendaPagamento, que guarda uma String que contém a escolha do pagamento de um empregado. A classe possui métodos que ajudam na hora da folha de pagamento, como: “ehDia”, que verifica se o localdate no parâmetro é o dia em que o empregado deve receber o pagamento; O “getDaysSinceLastPay” que retorna um localdate com a data do último pagamento; E o “getWeeks” que retorna a frequência em semanas com que o empregado deve ser pago.

Classes de serviço

Classes de serviço são aquelas que estão vinculadas com a realização de tarefas e atribuição de horas e taxas por elas. Uma dessas classes é a “CartaoDePonto”, que terá atributos da quantidade de horas que um empregado trabalhou atrelada ao dia em que ele o fez, bem como seus métodos getters. Outras duas classes são o “ResultadoDeVenda”, que irá guardar o valor da venda e sua data e a “TaxaServico” também guardando os mesmos atributos, onde o valor é relacionado à taxa que deve ser paga ao sindicato. E por último, há a classe “MembroSindicado”, que terá como atributos o id do membro, sua taxa sindical e uma lista de taxas de serviço.

Para o User Story 8, foi necessário torná-las clonáveis, já que não são tipos naturais do Java e é necessário salvar diversas versões da HashMap que armazena os empregados.

Classe Sistema

Como falado na seção de design arquitetural, a classe sistema é responsável por manter uma coleção de empregados utilizando um HashMap estático. Possui métodos getters e setters para manipular os dados presentes diretamente no HashMap, onde utilizam-se dos métodos definidos nas classes das seções acima.

Exemplos de métodos presentes: “getEmployeeAttribute”, que retorna o atributo do empregado pelo seu id**;** “getWorkedHours”, que retorna a quantidade de horas trabalhadas por um empregado**;** e métodos que mudam o tipo de um empregado, como “changeEmployeeTypeToHorista”.

Para o user story 7, que não foi feito no Milestone 1, foram desenvolvidos os métodos para gerar toda a folha de pagamento, sendo dividida por cada tipo de empregado e possuindo também o total a ser pago para os empregados, e também o método “totalFolha”, que não roda a folha de pagamento, mas fala quanto seria o total se rodasse.

Para o segundo milestone, cada método que faz alguma alteração no sistema, como “addNewSale” por exemplo, possui um método “history.push”, proveniente da classe History, que irá salvar a operação na pilha de histórico do memento, padrão de projeto que será explicado mais abaixo, para que ela possa ser desfeita conforme a vontade do cliente. Foram criados os métodos “undo” e “redo” que também interagem com a classe History e permitem que operações sejam refeitas e desfeitas. Além disso, foram feitas modificações nos métodos “getRawSalary” e “generateTotalPayroll” para satisfazerem a generalização feita nos user stories 9 e 10, já que agora os empregados possuem mais maneiras de serem pagos.

Padrões de projeto adotados

Facade

Descrição geral

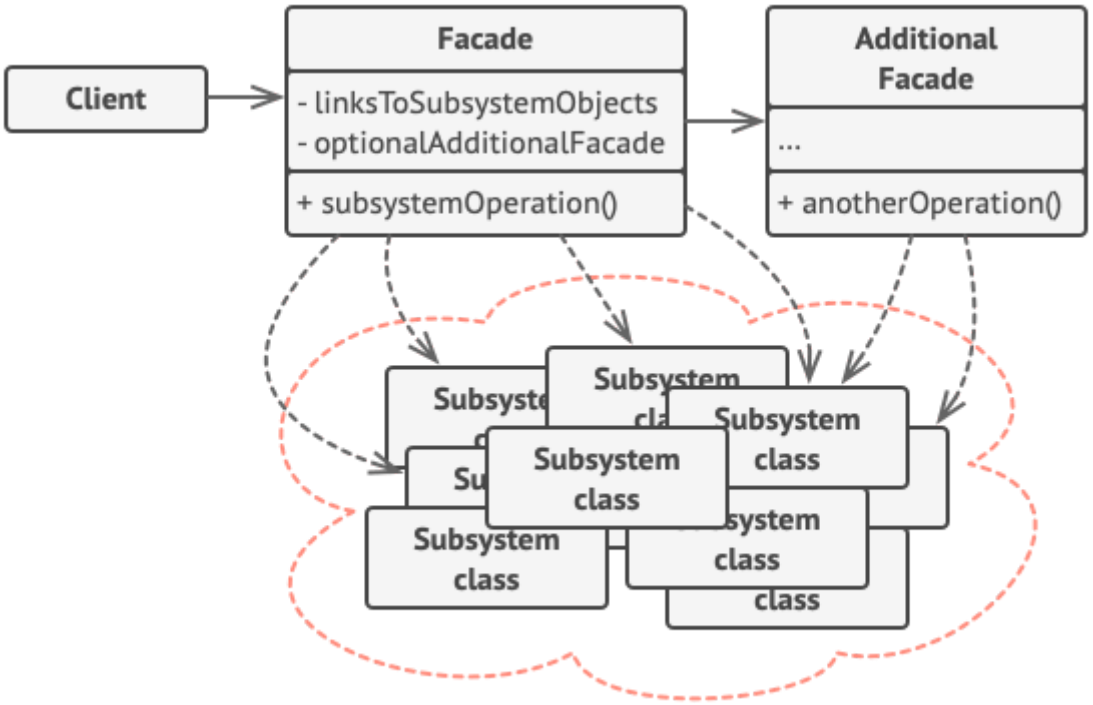
Uma Facade é um padrão de projeto que dispõe uma área/interface mais simplificada para interagir com o sistema, ocultando a complexidade dos componentes do sistema e melhorando a usabilidade do trabalho para os clientes.

De acordo com o site Refactoring Guru em 2014, a Facade estabelece a seguinte estrutura:

“A Facade fornece um acesso à uma parte da funcionalidade do subsistema e direciona o pedido do cliente, operando em todas as partes móveis.

O Subsistema consiste em dúzias de objetos variados. As classes do subsistema não estão cientes da existência da fachada. Elas operam dentro do sistema e trabalham entre si diretamente.

O Cliente usa a Facade ao invés de chamar os objetos do subsistema diretamente.”



Fonte: Refactoring Guru
Disponível em: <https://refactoring.guru/pt-br/design-patterns/facade>. Acesso em: 8 fev. 2024.

Problema resolvido

Em sistemas que possuem uma grande quantidade de componentes e interações, tornaria-se muito complexo criar, agrupar e manipular todos os objetos juntamente com seus métodos de maneira lógica e usual, respeitando as dependências entre os mesmos. Então, faz-se necessário a criação de um espaço que consiga solucionar tais desafios.

Identificação da Oportunidade

No sistema, foi notado que havia muitas classes, resultando em diversos métodos que ao decorrer do desenvolvimento deveriam se relacionar, o que

aumentaria em grande escala a dificuldade de construir o projeto de uma maneira logicamente correta e eficiente. Logo, para amenizar tal situação, descobriu-se a existência do padrão de projeto que possibilitou esclarecer a visão acerca do trabalho e solucionar o desafio.

Aplicação no projeto

No projeto foi implementada uma classe Facade, que oferece uma área mais simplificada para interagir com o sistema. Nela todos os métodos presentes nas classes descritas na seção anterior foram utilizados para calcular e desenvolver métodos de CRUD (Create, Read, Update, Delete) no sistema.

Como exemplo, temos os métodos de criação, remoção e update de um empregado: “criarEmpregado”, “removerEmpregado”, “alteraEmpregado”**;**

Métodos de leitura de informações acerca do empregado, onde também irão utilizar métodos de classes fora das classes de empregados para calcular e retornar o dado desejado: “getAtributoEmpregado”, “getHorasNormaisTrabalhadas”, “getHorasExtrasTrabalhadas”**;**

Método que verifica se um empregado está associado ao sindicato: “idSindicadoExiste”**;**

Método que checa se um empregado comparece ao serviço, servindo para calcular a quantidade de horas trabalhadas: “lancaCartao”**;**

Métodos relacionados à realização de serviços: “lancaVenda”, “getVendasRealizadas”, “lancaTaxaServico”, “getTaxasServico”**;**

E por fim, operações que apagam ou encerram o sistema: “zerarSistema”, “encerrarSistema”.

Para a segunda entrega, foram feitas alterações para o funcionamento do user story 7, que havia ficado pendente. Com isso, foram criados os métodos “totalFolha” e “rodaFolha”, que utiliza-se dos métodos da classe Sistema, como “getRawSalary” e outros, para gerar a folha de pagamento.

Já para o Milestone 2, foi adicionado o método “criarAgendaDePagamentos”, que adiciona um padrão de pagamento na lista de possíveis pagamentos a partir de métodos da classe Sistema e da camada de exceções.

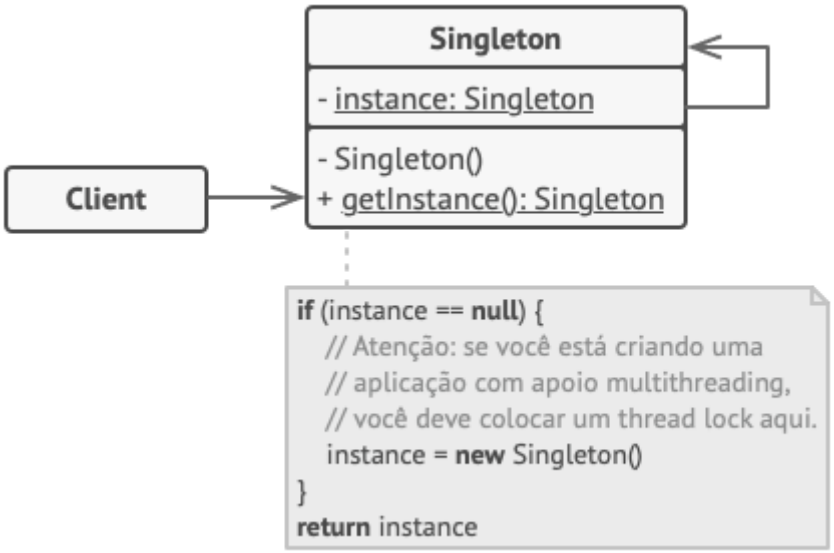
Também foram adicionados métodos “undo” e “redo”, também executando métodos da classe Sistema, que possibilitam o cliente desfazer e refazer uma operação. Esses métodos estão melhor explicados na seção de Aplicação do padrão de projeto Memento.

Singleton

Descrição geral

Singleton é um padrão de projeto que consiste em criar apenas uma instância de uma classe, não sendo permitido criar objetos adicionais dessa classe, e provendo um ponto de acesso global à ela.

A estrutura do padrão consiste em uma classe permitir ser instanciado apenas um objeto, possuindo um método “getInstance()”, que retornará a instância da própria classe, sendo esse método a única forma de obter o objeto singleton (Refactoring Guru, 2014).



Fonte: Refactoring Guru
Disponível em: <https://refactoring.guru/pt-br/design-patterns/singleton>. Acesso em: 9 fev. 2024.

Problema resolvido

Garante que uma classe tenha somente um objeto, controlando o acesso a algum recurso compartilhado, sendo usada apenas quando necessário. Além disso, protege a instância de ser sobrescrita por outro código.

Identificação da Oportunidade

Como haveria um acesso compartilhado dos dados ao ter que implementar a persistência de dados utilizando XML, identificou-se a necessidade de evitar a sobrescrição do arquivo e também evitar gerar novas instâncias para o armazenamento das informações, tornando o processo complexo e ineficiente.

Aplicação no projeto

Para garantir a persistência dos dados, foi desenvolvido o método “getInstance”, que segue os conceitos de Singleton, garantindo que caso não exista uma instância da Classe Database, uma nova instância será criada e retornada pelo método. Tal instância é responsável em conectar o sistema com os dados salvos no formato XML.

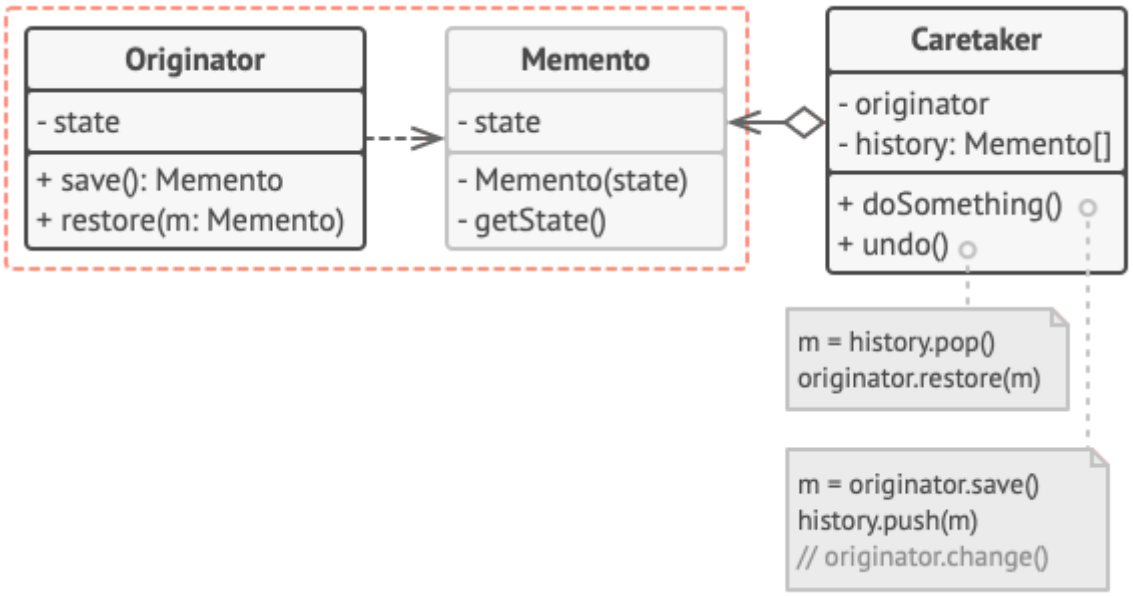
Memento

Descrição geral

Memento é um padrão de projeto comportamental que permite salvar e restaurar o estado interno de um objeto, sem violar o encapsulamento e sem revelar os detalhes de sua implementação interna.

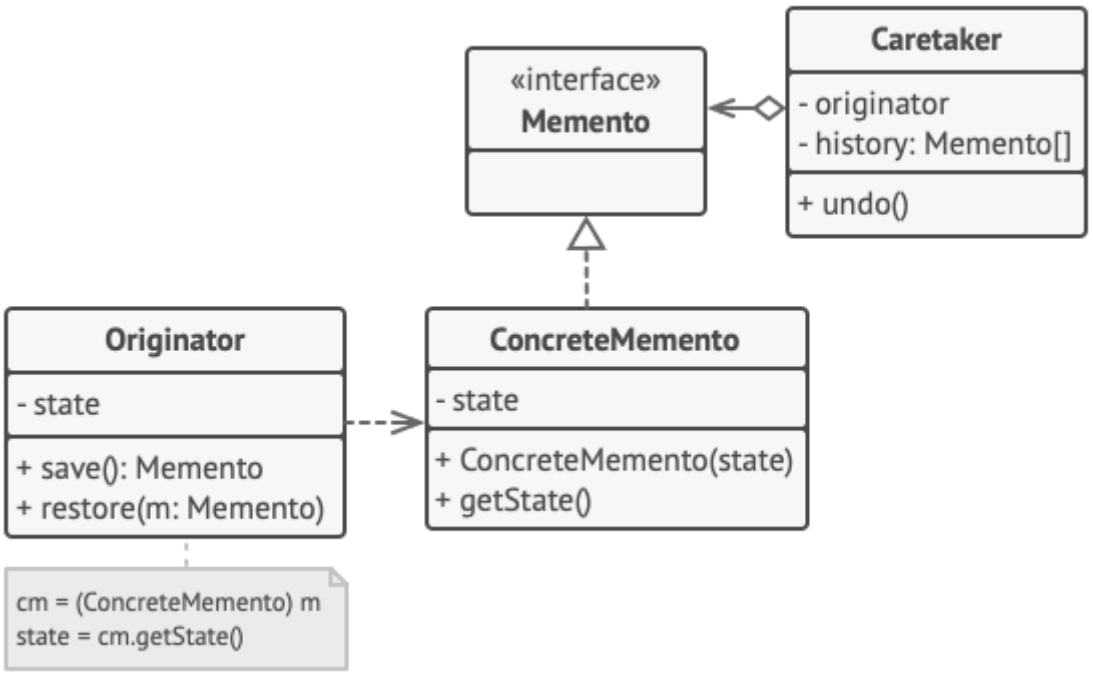
Segundo o site Refactoring Guru, em 2014, o padrão pode ter 3 tipos de estrutura:

- **Implementação baseada em classes aninhadas:** Organiza as classes envolvidas dentro da classe Originator. Essa abordagem permite encapsular a classe Memento dentro da classe Originator, simplificando a implementação e ocultando a interface Memento do mundo externo.



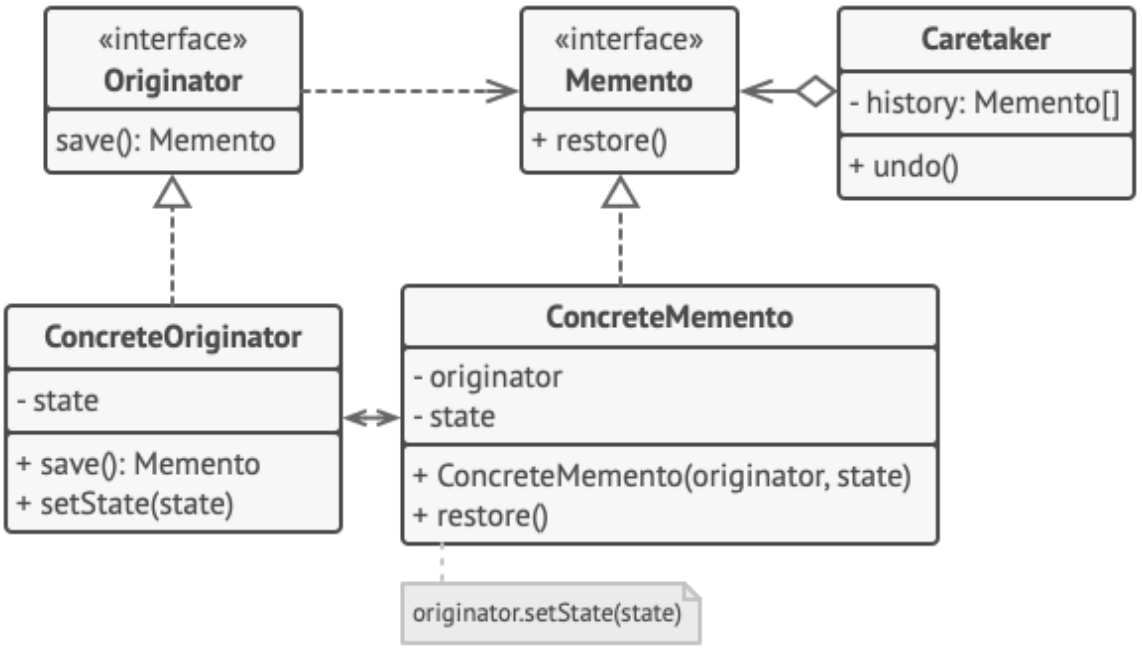
Fonte: Refactoring Guru
Disponível em: <https://refactoring.guru/pt-br/design-patterns/memento>. Acesso em: 2 mar. 2024.

- **Implementação baseada em uma interface intermediária:** É introduzida uma interface intermediária entre o Originator e o Memento. Essa interface proporciona uma abstração adicional, permitindo que o Originator trabalhe com Mementos sem precisar conhecer os detalhes específicos de cada implementação de Memento.



Fonte: Refactoring Guru
Disponível em: <https://refactoring.guru/pt-br/design-patterns/memento>. Acesso em: 2 mar. 2024.

- **Implementação com um encapsulamento ainda mais estrito:** O estado interno do Originator é encapsulado dentro do próprio Memento, sem expor qualquer detalhe do estado interno para o mundo externo. Isso proporciona um nível extremamente alto de encapsulamento, garantindo que o estado do Originator seja protegido e acessível apenas por meio de uma interface controlada.



Fonte: Refactoring Guru
Disponível em: <https://refactoring.guru/pt-br/design-patterns/memento>. Acesso em: 2 mar. 2024.

Problema resolvido

Ao tentar implementar a funcionalidade diretamente, gravando o estado dos objetos antes de cada operação e restaurando-os posteriormente, surgem diversos desafios. Esses desafios incluem a dificuldade de obter retratos do estado dos objetos devido a restrições de acesso aos seus campos privados, a necessidade de refatorar as classes quando ocorrem mudanças na estrutura dos objetos e a complexidade de armazenar e gerenciar os retratos do estado. A utilização do Memento resolve esses problemas encapsulando o estado interno dos objetos em objetos Memento, permitindo que os objetos originadores capturem e restaurem seu estado sem expor sua implementação interna.

Identificação da Oportunidade

No sistema, foi identificado a necessidade de criar operações de desfazer ou refazer outras operações, já que isto é um requisito do user story 8. Logo, para evitar os problemas descritos anteriormente acerca dessas operações, foi aplicado o padrão de projeto em nosso sistema.

Aplicação no projeto

No projeto foram construídas as classes Memento e History, já citadas no design arquitetural e na classe Sistema. A classe Memento é responsável por armazenar um snapshot do estado do HashMap de empregados após alguma alteração. O atributo “employeesSnapshot” é uma cópia da HashMap principal do sistema após alguma alteração. No construtor da classe, Ela recebe a referência do HashMap de empregados e faz uma cópia do HashMap para ser armazenada como atributo daquele objeto memento. O método restore permite restaurar o HashMap, fornecendo a referência do snapshot armazenado (o qual será feito uma copiado para a nossa HashMap do sistema utilizando o método setData na classe Database).

A classe History mantém o histórico de mudanças no sistema na forma de duas pilhas: uma para desfazer operações (undoStack) e outra para refazê-las (redoStack).

O método push é usado para inserir um novo Memento na pilha de desfazer. Ao fazer isso, a pilha de refazer é limpa, pois a inclusão de um novo Memento invalida qualquer operação que possa ser refeita.

Os métodos undo e redo são responsáveis por desfazer e refazer operações, respectivamente. Eles movem os Mementos entre as pilhas de desfazer e refazer, permitindo a reversão e a reaplicação de alterações no estado do sistema.
Para integrar ao sistema, o objeto History é utilizado em métodos que fazem alterações no sistema para salvar a operação na pilha de histórico do Memento. Isso é feito chamando o método push da classe History e passando o Memento apropriado que contém o estado do sistema após a operação.

Dessa forma, é permitido que os clientes desfaçam ou refaçam operações conforme desejado, utilizando a pilha de histórico mantida pela classe History.