UNIVERSIDADE FEDERAL DE ALAGOAS –
UFAL CAMPUS A. C. SIMÕES

Ciência da Computação

Programação Orientada a Objetos 2023.2- Professor Mário Hozano

Gabriela Batista Tenório das Neves

Caio Mascarenhas Soares

Relatório WePayU

Fevereiro de 2024

● Projeto

- Nome do Padrão de Projeto = Facade

- Descrição Geral: A classe facade é um design patterns onde todos os métodos são criados mas a implementação dos mesmos não é implementada na facade e sim em outro método que é chamado nele.

- Problema Resolvido: Esse projeto resolve um problema de folha de pagamento de usuários, onde um usuário é criado, atribuído, e pode ser lançado e pegado seu cartão de ponto, suas vendas, sua taxa sindical, suas horas trabalhadas,folha de pagamento e etc. O programa salva em arquivo .XML os dados de cada empregado gerando um ID único e dai resolve o problema específico.

- Identificação da Oportunidade: O uso do padrão facade foi de extrema importância já que o projeto tem muitos métodos, muitas classes e muitos objetos relacionados, o facade garante a organização e melhor leitura do código, sabendo onde está resolvendo cada problema.

- Aplicação no Projeto: O projeto pode ser facilmente usado em uma empresa que existe trabalhadores horistas, assalariados e comissionados. Ele pode ajudar e contribuir com lançamento de ponto, recebimento de salários e etc.

● Descrição Geral do Design Arquitetural

O sistema é projetado como um aplicativo de folha de pagamento que gerencia informações sobre empregados, incluindo seus detalhes pessoais, tipo de emprego, salários, vendas realizadas, horas trabalhadas, taxa de serviço sindical, entre outros. O sistema permite a criação, atualização e remoção de empregados, bem como o lançamento de vendas e taxas de serviço sindical e gerar folha de pagamento.

O padrão utilizado Facade fornece uma interface simplificada para um conjunto mais complexo de classes, tornando mais fácil para os clientes interagirem com o sistema. No nosso caso, a classe Facade fornece métodos simplificados que encapsulam a lógica de várias classes de controle.

São essas classes:

1. **public Facade()** throws FileNotFoundException {

SistemaController.**iniciarSistema**();}

O construtor da classe Facade é responsável por iniciar o sistema. Ele chama o método iniciarSistema() da classe SistemaController. Esse método contém a lógica de inicialização, incluindo a inicialização do arquivo .XML para persistência de dados, usando o XMLDecoder.

2. public void **zerarSistema**() {
   SistemaController.zerarSistema();}

Este método chama o método zerarSistema() da classe SistemaController. A função desse método redefine o estado do sistema, limpando dadospara "zerar" o sistema, usando o XMLEncoder e método .clear() na lista que desejar zerar.

3. public void **encerrarSistema**() {
   SistemaController.encerrarSistema();}

   Similar ao método anterior, este chama o método encerrarSistema() da classe SistemaController. Esse método contem a lógica para encerrar o sistema de maneira adequada, como salvando os dados com XMLEncoder no arquivo desejado , garantindo assim a persistência e o uso futuro desses.

4. public void **lancaTaxaServico**(String emp, String data, String valor) {
   ServicoController.lancaTaxaServico(emp, data, valor);}

Esse método chama o método lancaTaxaServico() da classe ServicoController. Ele é utilizado para lançar taxas de serviço para um empregado em uma determinada data.

5. public String **getTaxasServico**(String emp, String dataInicial, String dataFinal) {
   return ServicoController.getTaxasServico(emp, dataInicial, dataFinal);}

Este método chama getTaxasServico() da classe ServicoController e retorna as taxas de serviço para um empregado em um intervalo de datas, se houver.

6. public void **lancaVenda**(String emp, String data, String valor) {
   VendasController.lancaVenda(emp, data, valor);}

Este método chama o método lancaVenda() da classe VendasController. Serve para lançar informações(valor) sobre vendas realizadas por um empregado comissionado em uma determinada data.

7. public String **getVendasRealizadas**(String emp, String dataInicial, String dataFinal) {
   return VendasController.getVendasRealizadas(emp, dataInicial, dataFinal);}

   Esse método chama getVendasRealizadas() da classe VendasController e retorna as vendas realizadas por um empregado

comissionado em um intervalo de datas, se houver.

8. public String **getHorasExtrasTrabalhadas**(String emp, String dataInicial, String dataFinal) {
   return PontoController.getHorasExtrasTrabalhadas(emp, dataInicial, dataFinal);
   }

   Este método chama getHorasExtrasTrabalhadas() da classe PontoController e retorna as horas extras trabalhadas por um empregado horista em um intervalo de datas.

9. public String **getHorasNormaisTrabalhadas**(String emp, String dataInicial, String dataFinal) {
   return PontoController.getHorasNormaisTrabalhadas(emp, dataInicial, dataFinal);
   }

   Semelhante ao método anterior, este chama getHorasNormaisTrabalhadas() da classe PontoController, retornando as horas normais trabalhadas por um empregado horista em um intervalo de datas, sendo 8 horas o máximo de horas em um único dia.

10. public void **lancaCartao**(String emp, String data, String horas) {
    PontoController.lancaCartao(emp, data, horas);
    }

    Este método chama lancaCartao() da classe PontoController. É utilizado para lançar informações sobre o cartão de ponto de um empregado horista, especificando a data e as horas trabalhadas.

11. public String **getAtributoEmpregado**(String emp, String atributo) {
    return EmpregadoController.getAtributoEmpregado(emp, atributo);
    }

    Este método chama getAtributoEmpregado() da classe EmpregadoController e retorna um atributo específico de um empregado, com base no Id do empregado e no nome do atributo.

12. public String **getEmpregadoPorNome**(String nome, int indice) {
    return EmpregadoController.getEmpregadoPorNome(nome, indice);
    }

    Semelhante ao método anterior, esse método chama getEmpregadoPorNome() da classe EmpregadoController e retorna o Id de um empregado com base no nome e no índice fornecidos, indice serva caso haja 2 ou mais pessoas com o mesmo nome.

13. public void **removerEmpregado**(String emp) {
    EmpregadoController.removerEmpregado(emp);
    }

    Este método chama removerEmpregado() da classe EmpregadoController e

é utilizado para remover um empregado do sistema e da lista com base no Id do empregado.

14. public String **criarEmpregado**(String nome, String endereco, String tipo, String salario) {
    return EmpregadoController.criarEmpregado(nome, endereco, tipo, salario);
    }
    Esse método chama criarEmpregado() da classe EmpregadoController e é utilizado para criar um novo empregado do tipo horista ou assalariado no sistema com base no nome, endereço, tipo e salário fornecidos. Esse método adiciona o empregado na lista que será salva no XML.

15. public String **criarEmpregado**(String nome, String endereco, String tipo, String salario, String comissao) {
    return EmpregadoController.criarEmpregado(nome, endereco, tipo, salario, comissao);
    }
    Semelhante ao método anterior, este método chama criarEmpregado() da classe EmpregadoController, mas permite a criação de um empregado do tipo comissionado com atributo adicional de comissão.

16. public void **undo**() {
    SistemaController.popUndo();
    }

    public void **redo**() {
    SistemaController.popRedo();
    }
    Os métodos undo() e redo() da classe Facade estão relacionados à funcionalidade de desfazer e refazer ações no sistema. Eles chamam os métodos correspondentes na classe SistemaController. O método undo() permite desfazer a última operação realizada no sistema, enquanto o método redo() permite refazer a última operação desfeita.

17. public void **rodaFolha**(String data, String saida){
    FolhaController.rodaFolha(data, saida);
    }
    O método rodaFolha() chama o método homônimo na classe FolhaController. Esse método é responsável por calcular e processar a folha de pagamento para a data especificada, gerando informações de saída conforme necessário.

18. public String **totalFolha(**String data){
    return FolhaController.totalFolha(data);}

O método totalFolha() da classe Facade chama o método correspondente na classe FolhaController. Esse método retorna o total da folha de pagamento para a data especificada, incluindo salários, comissões, horas extras, e outros componentes relevantes.

19. public void **criarAgendaDePagamentos**(String agenda){ FolhaController.criarAgenda(agenda);}

O método criarAgendaDePagamentos() da classe Facade chama o método criarAgenda() na classe FolhaController. Esse método é utilizado para criar uma nova agenda de pagamentos no sistema, permitindo a configuração de regras específicas para o pagamento dos empregados.

20. public void **alteraEmpregado**(String emp,String atributo, String valor) throws {
EmpregadoController.alteraAtributoEmpregado(emp, atributo, valor);
}

Este método chama alteraAtributoEmpregado() da classe EmpregadoController e é utilizado para alterar um atributo específico de um empregado, com base no Id do empregado, nome do atributo e novo valor.

21. public void **alteraEmpregado**(String emp,String atributo, String valor1, String comissao) throws ValorTrueFalse, IdentificacaoNula, NaoComissionado, EmpregadoNaoExisteException {
EmpregadoController.alteraEmpregado(emp,atributo,valor1, comissao); }

Semelhante ao método anterior, mas permite a alteração de um atributo específico para um empregado comissionado, incluindo a comissão.

22. public void **alteraEmpregado**(String emp, String atributo, String valor, String idSindicato, String taxaSindical) throws {
EmpregadoController.alteraEmpregadoSindicalizado(emp, "sindicalizado", true, idSindicato, taxaSindical); }

Este método chama alteraEmpregadoSindicalizado() da classe EmpregadoController e é utilizado para alterar um empregado para ser sindicalizado, incluindo o id do sindicato e a taxa sindical.

23. public void alteraEmpregado(String emp, String atributo, String valor1, String banco, String agencia, String contaCorrente) throws IdentificacaoNula {
EmpregadoController.adicionaMetodoPagamento(emp, "metodoPagamento", "banco", banco, agencia, contaCorrente); }

Este método chama adicionaMetodoPagamento() da classe EmpregadoController e é utilizado para adicionar um método de pagamento para um empregado, incluindo informações bancárias.

● Principais Componentes e Interações

1. EmpregadoController:

- **Responsabilidade:** Gerencia as operações relacionadas aos empregados, incluindo criação, atualização e remoção.
- **Interações:**
  - Interage com a classe `VendasController` para realizar operações relacionadas ao lançamento de vendas.
  - Colabora com a classe `PontoController` para registrar as horas trabalhadas pelos empregados.
  - Envia solicitações à classe `ServicoController` para o lançamento de taxas de serviço sindical.
  -Colabora com a classe 'FolhaController' para o lançamento da folha de pagamento dos empregados.
  - Pode interagir com a classe `SistemaController` para a execução de operações globais do sistema.

2. VendasController:

- **Responsabilidade:** Controla as operações relacionadas ao lançamento de vendas realizadas pelos empregados.
- **Interações:**
  - Pode interagir com a classe `EmpregadoController` para obter informações sobre os empregados envolvidos nas vendas.

3. PontoController:

- **Responsabilidade:** Gerencia o registro de horas trabalhadas pelos empregados, incluindo horas extras e horas normais.
- **Interações:**
  - Colabora com a classe `EmpregadoController` para atualizar informações relacionadas às horas trabalhadas pelos empregados.

4. ServicoController:

- **Responsabilidade:** Responsável pelo lançamento de taxas de serviço sindical para os empregados.
- **Interações**:
  - Pode interagir com a classe `EmpregadoController` para associar as taxas de serviço aos empregados correspondentes.

5. SistemaController:

- **Responsabilidade:** Controla o fluxo geral do sistema, inicializando-o, encerrando-o e realizando operações de inicialização.
- **Interações:**
  - Pode interagir com todas as outras classes controladoras para coordenar operações globais do sistema.
  - Envia comandos de inicialização e encerramento às classes controladoras específicas.

6. FolhaController:

-**Responsabilidade:** O FolhaController desempenha um papel central no processamento da folha de pagamento. Sua principal responsabilidade é gerenciar o cálculo de salários, comissões, horas extras e outras operações relacionadas ao pagamento dos empregados.

-**Interações:**

-EmpregadoController: Colabora com o EmpregadoController para obter informações detalhadas sobre os empregados, permitindo o cálculo preciso de salários e comissões.

-VendasController: Interage com o VendasController para acessar dados sobre vendas realizadas por empregados comissionados, incorporando essas informações no processo de folha de pagamento.

-PontoController: Utiliza o PontoController para obter dados sobre as horas trabalhadas pelos empregados, incluindo horas extras e horas normais, essenciais para o cálculo dos salários.

-SistemaController: Envia comandos para a criação de novas agendas de pagamento, utilizando a classe FolhaController. Além disso, pode interagir com o SistemaController para coordenar operações globais relacionadas à folha de pagamento.

● Estrutura do projeto

O projeto em java no diretório SRC que contém o conteúdo principal foi dividido em demais pastas nomeadas Models, Controllers , Utils e Exception.

- **Models**: Contém a classe principal de cada entidade/objeto do problema: Empregado ,Venda,banco,Cartao de Ponto…
  As subclasses herdam atributos dessa classe, e nela contém os atributos comuns a todos da entidade e lá são setados, recebidos e guardados.

- **Controller:** Contém os controladores dos problemas de cada entidade, os métodos chamados na Facade são implementados em seus respectivos controllers, é onde toda a lógica do problema está implementada.

- **Exception:** Contém as exceções que o problema retorna a cada erro citado no script de testes.

- **Utils**: Alguns métodos não relacionados ao problema em si e sim a lógica para execução do problema, que são implementados separadamente para maior organização, por exemplo verificar se um data é válida, se é dia de pagamento, se o endereço está em branco e etc.

● Undo Redo

- O padrão de design "Undo/Redo" é uma técnica comum em sistemas computacionais que permite aos usuários desfazer (undo) ou refazer (redo) ações. Essa funcionalidade é útil para recuperar um estado anterior do sistema ou repetir ações previamente desfeitas. O código fornecido implementa essa lógica usando duas pilhas, **undo** e **redo,** para rastrear as alterações feitas no sistema.

```
public static void pushUndo(EmpregadoController e) throws Exception {
   HashMap<String, Empregado> novaHash = new HashMap<>();
   if (undo == null)
      undo = new Stack<>();
   undo.push(novaHash);
}
```

- Este método é responsável por empilhar um estado atual do EmpregadoController na pilha undo. Cada estado é representado por um novo HashMap<String, Empregado>. Isso permite armazenar diferentes versões do estado do sistema.

```
public static void pushUndo(HashMap<String, Empregado> e) throws Exception {
   if (undo == null)
      undo = new Stack<>();
   undo.push(e);
}
```

- Este método é semelhante ao anterior, mas recebe diretamente um HashMap<String, Empregado> para ser empilhado na pilha undo. Isso é útil quando o estado do sistema é representado apenas pela lista de empregados.

```
public static HashMap<String, Empregado> popUndo() throws Exception {
   HashMap<String, Empregado> e = undo.peek();
   if (undo.size() > 1)  SistemaController.pushRedo(e);
   undo.pop();
   return e;
}
```

- popUndo desempilha o estado atual do undo e retorna esse estado. Se houver mais de um estado na pilha, ele empilha o estado desfeito na pilha redo.

```
public static void pushRedo(HashMap<String, Empregado> e) throws Exception {
   if (redo == null) redo = new Stack<>();
   redo.push(e);
}
```

- pushRedo adiciona um estado à pilha redo quando uma ação é desfeita.

```
public static HashMap<String, Empregado> popRedo() throws Exception {
   HashMap<String, Empregado> e = redo.peek();
   SistemaController.pushUndo(e);
   redo.pop();
   return e;
}
```

- popRedo desempilha o estado atual do redo e empilha esse estado de volta na pilha undo para permitir a ação de refazer.

Essas operações de push e pop nas pilhas undo e redo permitem a implementação efetiva do padrão Undo/Redo, possibilitando aos usuários desfazer e refazer alterações no estado do sistema.