UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

MARJORY MEL FERREIRA FERRO LEMOS

LISTA 2: DESIGN PATTERNS E REFATORAÇÃO

ARQUITETURA DE SOFTWARE

CORNÉLIO PROCÓPIO

2024

1 PADRÕES DE PROJETOS UTILIZADOS

Este projeto implementa três padrões de projeto: Command, Factory Method e Facade. Além de três refatorações que usam o princípio SOLID: Princípio da Responsabilidade Única, Princípio de Substituição de Liskov e Princípio da Inversão de Dependência.

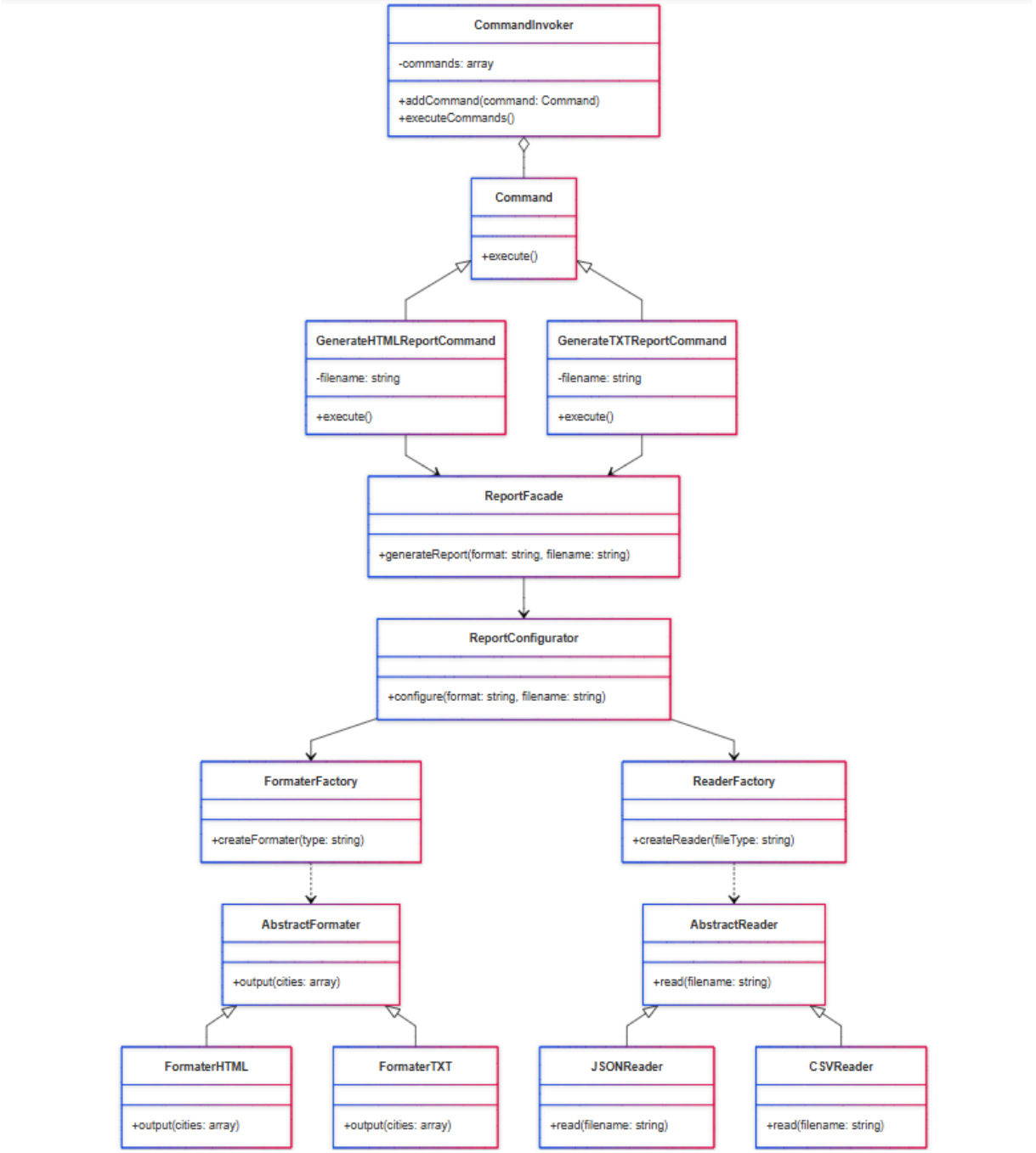


Figura 1. Diagrama do Projeto

1.1. Padrão Command

Permite encapsular uma solicitação como um objeto, promovendo a parametrização de objetos com diferentes solicitações e o enfileiramento ou registro de operações.

Como foi implementado?

- A classe Command define uma interface base com o método execute, que é implementada pelas subclasses.
- Os comandos concretos, como GenerateHTMLReportCommand e GenerateTXTReportCommand, encapsulam a lógica específica para gerar relatórios em formatos diferentes (html e txt).
- O CommandInvoker gerencia a fila de comandos e executa cada comando em sequência.

Por que usar Command?

- Segue o **Princípio da Inversão de Dependência (DIP)** ao desacoplar quem solicita a operação (CommandInvoker) de quem a executa (Command).
- Facilita a extensão com novos comandos, como a criação de um GeneratePDFReportCommand.

1.2. Padrão Factory Method

Define uma interface para criar objetos em uma superclasse, permitindo que subclasses decidam quais objetos criar.

Por que usar Factory Method?

- Segue o **Princípio da Responsabilidade Única (SRP)** ao centralizar a lógica de criação de objetos.
- Evita acoplamento direto entre ReportFacade e implementações específicas de formatadores ou leitores.

Como foi implementado?

- A classe FormaterFactory é usada para criar instâncias de formatadores (FormaterHTML e FormaterTXT) com base no tipo solicitado.
- A classe ReaderFactory cria instâncias de leitores (JSONReader, CSVReader) com base na extensão do arquivo.

1.3. Padrão Facade

Fornece uma interface simplificada para um conjunto de interfaces em um subsistema, reduzindo a complexidade para o cliente.

Por que usar Facade?

- Simplifica a interface para os usuários finais, que interagem apenas com ReportFacade em vez de gerenciar múltiplos objetos.
- Segue o **Princípio da Responsabilidade Única (SRP)** ao delegar a configuração de dependências para o ReportConfigurator.

Como foi implementado?

- A classe ReportFacade atua como a interface principal para gerar relatórios, escondendo a complexidade de configuração e integração com formatadores e leitores.
- Internamente, ela utiliza o ReportConfigurator (após refatoração) para configurar as dependências e os leitores necessários.

2 REFATORAÇÕES

A refatoração do código foi realizada com foco em três princípios fundamentais do SOLID: o Princípio da Responsabilidade Única (SRP), o Princípio de Substituição de Liskov (LSP) e o Princípio da Inversão de Dependência (DIP).

2.1 .Refatoração com o Princípio da Responsabilidade Única (SRP)

No que diz respeito ao Princípio da Responsabilidade Única, o objetivo era garantir que cada classe tivesse uma única responsabilidade bem definida. Para alcançar isso, foi criada a classe ReportConfigurator, cuja função é delegar a configuração das dependências, como FormaterFactory e ReaderFactory. Essa mudança permitiu remover essa lógica do ReportFacade, simplificando sua estrutura. Agora, o ReportFacade tem como única responsabilidade orquestrar a geração do relatório, delegando a criação das dependências ao ReportConfigurator. Essas alterações foram implementadas no arquivo ReportConfigurator.js e utilizadas no ReportFacade.js.

Implementação: Antes da refatoração, o ReportFacade era responsável tanto pela geração do relatório quanto pela configuração das dependências:

```
// Antes: ReportFacade.js
export default class ReportFacade {
    static async generateReport(format, filename) {
      const formaterStrategy = FormaterFactory.createFormater(format);
      const fileType = filename.split('.').pop().toLowerCase();
      const reader = ReaderFactory.createReader(fileType);
      const cities = await reader.read(filename);
      return formaterStrategy.output(cities);
    }
}
```

Após a refatoração, foi criada a classe ReportConfigurator para lidar com a configuração das dependências:

```
// Depois: ReportConfigurator.js
export default class ReportConfigurator {
    static configure(format, filename) {
      const formaterStrategy = FormaterFactory.createFormater(format);
      const fileType = filename.split('.').pop().toLowerCase();
      const reader = ReaderFactory.createReader(fileType);
      return { formaterStrategy, reader };
    }
  }

  // Depois: ReportFacade.js
  import ReportConfigurator from "./ReportConfigurator";

  export default class ReportFacade {
    static async generateReport(format, filename) {
      const { formaterStrategy, reader } = ReportConfigurator.configure(format, filename);
      const cities = await reader.read(filename);
      return formaterStrategy.output(cities);
    }
  }
```

Impacto:

- Melhorou a modularidade ao separar as responsabilidades de configuração e geração de relatórios.
- Facilitou a manutenção, pois alterações na lógica de configuração agora podem ser feitas sem afetar a lógica de geração de relatórios.
- Aumentou a flexibilidade, permitindo que diferentes configurações sejam facilmente implementadas no futuro.

2.2 Refatoração com o Princípio de Substituição de Liskov (LSP)

Quanto ao Princípio de Substituição de Liskov, o foco era assegurar que as subclasses pudessem ser substituídas por suas superclasses sem comprometer o funcionamento do sistema. Para atingir esse objetivo, foi realizado um ajuste no AbstractReader, garantindo que todas as suas subclasses, como JSONReader e CSVReader, retornassem uma Promise no método read. Essa modificação unificou o comportamento esperado entre os diferentes leitores, permitindo que o ReportFacade pudesse utilizá-los de forma intercambiável, sem se preocupar com suas implementações específicas. Essas alterações foram aplicadas nos arquivos AbstractReader.js, JSONReader.js e CSVReader.js.

Implementação: Antes da refatoração, o método `read` nas classes de leitores poderia ter implementações inconsistentes:

```
// Antes: AbstractReader.js
export default class AbstractReader {
    read(filename) {
      throw new Error("Should implement read method");
    }
  }

  // Antes: JSONReader.js
  export default class JSONReader extends AbstractReader {
    read(filename) {
      // Implementação síncrona
      const data = JSON.parse(fs.readFileSync(filename, "utf8"));
      return data;
    }
  }
```

Após a refatoração, foi garantido que todas as implementações do método `read` retornem uma Promise:

```
// Depois: AbstractReader.js
export default class AbstractReader {
    async read(filename) {
      return Promise.reject(new Error("Should implement read method"));
    }
  }

  // Depois: JSONReader.js
  export default class JSONReader extends AbstractReader {
    async read(filename) {
      return new Promise((resolve, reject) => {
        try {
          const data = JSON.parse(fs.readFileSync(filename, "utf8"));
          resolve(data);
        } catch (error) {
          reject(error);
        }
      });
    }
  }
```

Impacto:

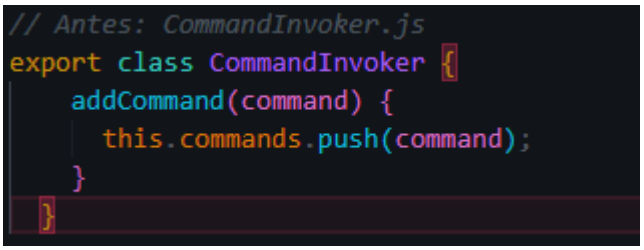
- Melhorou a consistência entre as diferentes implementações de leitores.

- Permitiu que o ReportFacade tratasse todos os leitores de forma uniforme, sem se preocupar com implementações síncronas ou assíncronas.
- Facilitou a adição de novos tipos de leitores no futuro, garantindo que eles sigam o mesmo contrato.

2.3 Refatoração com o Princípio da Inversão de Dependência (DIP)

Por fim, para atender ao Princípio da Inversão de Dependência, buscou-se desacoplar as classes de alto nível das classes de baixo nível. Isso foi alcançado garantindo que o CommandInvoker trabalhasse exclusivamente com a classe base Command, sem depender diretamente de comandos concretos como GenerateHTMLReportCommand ou GenerateTXTReportCommand. Foi implementada uma verificação explícita no método addCommand para assegurar que todos os comandos adicionados ao invocador estendessem a classe Command. Essas modificações foram realizadas nos arquivos Command.js, CommandInvoker.js, e nos arquivos de comandos específicos como GenerateHTMLReportCommand.js e GenerateTXTReportCommand.js.

Implementação: Antes da refatoração, o CommandInvoker poderia não ter uma verificação explícita do tipo de comando:



Após a refatoração, foi adicionado uma verificação explícita:

```
// Depois: CommandInvoker.js
import { Command } from "./Command.js";

export class CommandInvoker {
  addCommand(command) {
    if (!(command instanceof Command)) {
      throw new Error("Command must extend the Command class");
    }
    this.commands.push(command);
  }
}
```

Impacto:

- Aumentou a robustez do sistema, garantindo que apenas comandos válidos sejam adicionados ao invocador.
- Melhorou a detecção precoce de erros, lançando uma exceção se um comando inválido for adicionado.
- Reforçou o acoplamento com a abstração (Command) em vez de implementações concretas.

REFERÊNCIAS

Código-Fonte no Github. Disponível em <https://github.com/marjorymell/gerenciador-contatos>.

Design Patterns. Disponível em: <https://refactoring.guru/design-patterns>. Acesso em 5 nov. de 2024.

Refactoring Guru. Disponível em: <https://refactoring.guru/refactoring>. Acesso em 5 nov. de 2024.