UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO DE COMPUTAÇÃO
ENGENHARIA DE SOFTWARE

DANIELA RODRIGUES COUTO COSTA RA: 2454297
MARCELO GREGÓRIO VIEIRA RA: 2454408

LISTA 2 - PADRÕES DE PROJETO E REFATORAÇÃO

ARQUITETURA DE SOFTWARE

CORNÉLIO PROCÓPIO
2024

Objetivo do projeto:

Aplicar um padrão de projeto criacional, estrutural e comportamental, bem como realizar três fatorações usando os princípios SOLID. Além disso, deve-se criar um script de teste para a execução do código, garantindo que as saídas sejam correspondentes após as alterações realizadas.

Estrutura do sistema:

Padrão de projeto criacional - Singleton:

O padrão de projeto criacional escolhido foi o Singleton e foi aplicado na classe CitiesReporter.js, com o objetivo de garantir que apenas uma instância da classe pudesse ser instanciada durante a execução, o que garante o acesso seguro da classe e seus métodos.

Padrão de Projeto Estrutural - Facade:

No sistema foi implementado o padrão estrutural Facade que possibilita a divisão do sistema em camadas, sendo uma camada simples para obtenção dos dados a serem trabalhados e apresentados, e uma camada complexa responsável pelo funcionamento das funções, dado que o usuário não possuirá acesso às classes de forma direta, conforme o objetivo e forma de operação do sistema.

Padrão de Projeto Comportamental - Strategy:

O padrão de projeto comportamental utilizado foi o Strategy, sendo que foram desenvolvidas duas estratégias distintas para entradas do usuário, as quais são gerenciadas de forma diferente, bem como apresentam um resultado final distinto. Esse padrão facilita a organização e compreensão do código bem como facilita a expansão conforme novas estratégias sejam implementadas

O padrão foi implementado na classe Facade.js no seguinte trecho:

```
        defineEstrategia(estrategia) {
                if (estrategia == 'html') {
                    try {
                                    this.CitiesReporter = CitiesReporter.getInstancia({
formaterStrategy: this.FormaterHTML })
                    } catch (error) {
                        console.error(`Error: ${error.message}`);
                        process.exit(1);
                    }
                }
                else if (estrategia == 'txt') {
                    try {
                                    this.CitiesReporter = CitiesReporter.getInstancia({
formaterStrategy: this.FormaterTXT })
                    } catch (error) {
                        console.error(`Error: ${error.message}`);
                        process.exit(1);
                    }
                }
                else {
                    // Implementar erro
                }
            }
```

Refatorações:

Open-Closed Principle (OCP):

OCP garante que objetos estejam abertos para extensão mas fechados para modificações, a aplicação da classe **Facade** garante que o arquivo **Index** não acesse ou altere as classes **AbstractFormater**, **FormaterHTML**, **FormaterTXT**, visto que essas classes são implementadas na classe **Facade** e não no arquivo **Index** (Como foi realizado antes da fatoração). Além disso, a classe **Facade** é instanciada na classe **Index** para garantir o acesso indireto das principais funcionalidades do sistema sem prejudicar sua integridade.

Liskov* Substitution Principle (LSP):

A classe **AbstractFormater** lança o erro mas não garante a implementação do método ‘output’ nas subclasses. Nesse caso, caso haja a tentativa de substituição da classe **AbstractFormater** por uma subclasse sem implementar o método *output*, o sistema falhará, pois o método da classe base lança um erro. Como solução, foi implementado um método construtor que forçará a implementação do método *output* em todas as subclasses que herdam **AbstractFormater**, ainda, garantindo que a classe base não seja instanciada diretamente, aumentando a robustez da classe.

Dependency Inversion Principle (DIP):

Na classe **CitiesReport** o método *report* misturava as responsabilidades de leitura, parsing de JSON e formatação, violando o Princípio da Responsabilidade Única. Como solução, foi implementada a classe **FileReader** que encapsula a lógica de leitura de arquivos e parsing de JSON.

O seguinte diagrama de classes representa o sistema com as modificações para a sua adequação aos padrões de projeto implementados e as refatorações realizadas.

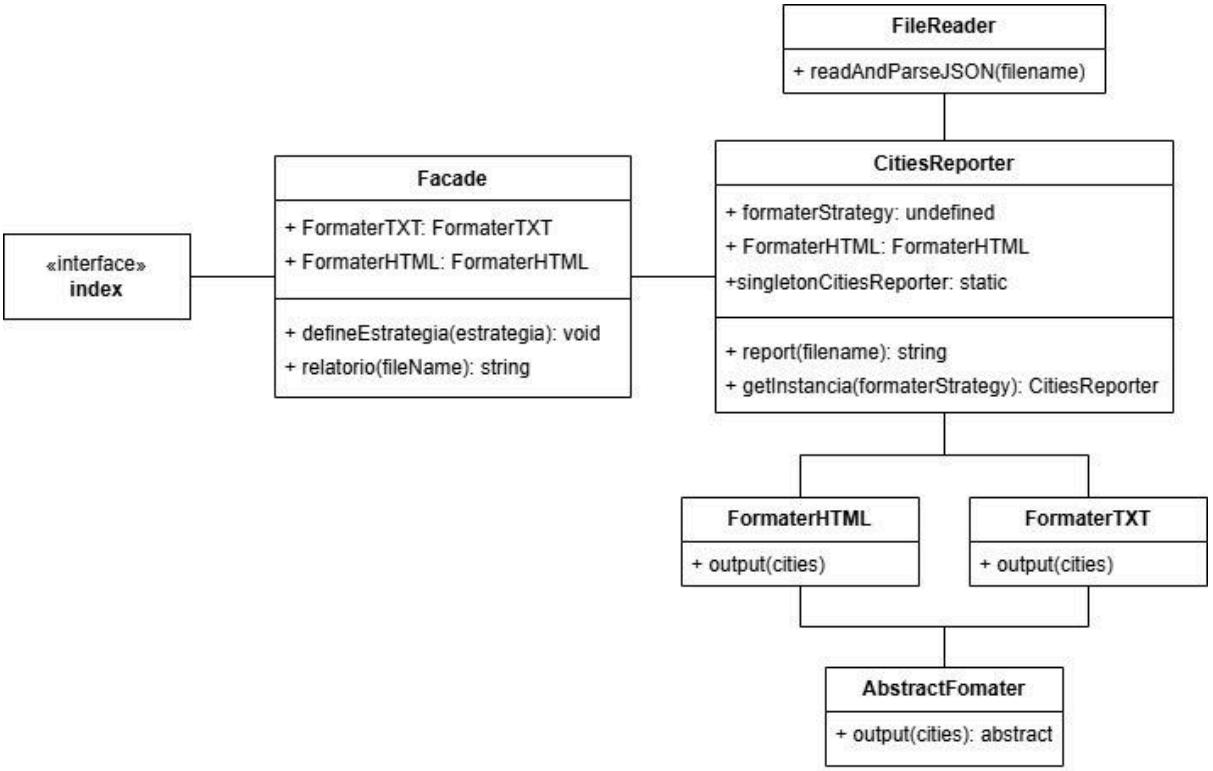

Figura 1: Diagrama de Classe UML do sistema com suas adequações

O código também pode ser acessado por um repositório no GitHub clicando aqui.