IFTM

CURSO DE GRADUAÇÃO TECNOLÓGICA EM ANALISE E DESENVOLVIMENTO DE SISTEMAS

Nome: João Soares

Pesquisa: Padrões de Projetos

Ituiutaba – MG

2022

1- Enunciado

Trabalho consiste em realizar uma pesquisa sobre os padrões de projetos listados abaixo:

- Padrões Interator e Composite;

- Padrão State;

- Padrão Proxy;

Para cada padrão deve explicar o funcionamento do padrão de projetos, apresentar o seu diagrama de classe.

Para cada padrão listado acima, implemente um projeto demonstrando o seu funcionamento. Todas as implementações devem ser entregues utilizando apenas um repositório, através de link do github via google classroom, juntamente com o documento descritivo da pesquisa..

2. Padrão Iterator

O Iterator é um padrão de projeto comportamental que permite a você percorrer elementos de uma coleção sem expor as representações dele (lista, pilha, árvore, etc.). Coleções são um dos tipos de dados mais usados em programação. Não obstante, uma coleção é apenas um contêiner para um grupo de objetos.

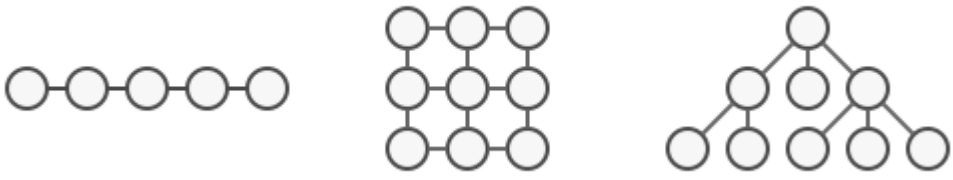

Figura 2.1: Vários tipos de coleções

A maioria das coleções armazena seus elementos em listas simples. Contudo, alguns deles são baseados em pilhas, árvores, grafos, e outras estruturas complexas de dados. Mas independente de quão complexa uma coleção é estruturada, ela deve fornecer uma maneira de acessar seus elementos para que

outro código possa usá-los. Deve haver uma maneira de ir de elemento em elemento na coleção sem ter que acessar os mesmos elementos repetidamente.

Isso parece uma tarefa fácil se você tem uma coleção baseada numa lista. Você faz um loop em todos os elementos. Mas como você faz a travessia dos elementos de uma estrutura de dados complexas sequencialmente, tais como uma árvore. Por exemplo, um dia você pode apenas precisar percorrer em profundidade em uma árvore. No dia seguinte você pode precisar percorrer na amplitude. E na semana seguinte, você pode precisar algo diferente, como um acesso aleatório aos elementos da árvore.

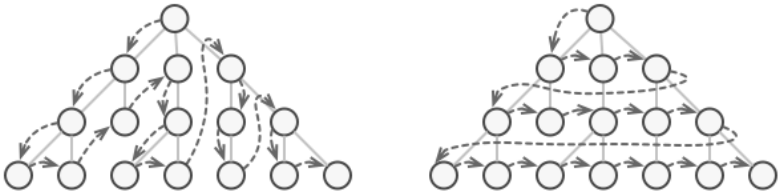

Figura 2.2: A mesma coleção pode ser atravessada de diferentes formas.

Adicionando mais e mais algoritmos de travessia para uma coleção gradualmente desfoca sua responsabilidade primária, que é um armazenamento de dados eficiente. Além disso, alguns algoritmos podem ser feitos para aplicações específicas, então incluí-los em uma coleção genérica pode ser estranho.

Por outro lado, o código cliente que deveria trabalhar com várias coleções pode não se importar com a maneira que elas armazenam seus elementos. Contudo, uma vez que todas as coleções fornecem diferentes maneiras de acessar seus elementos, você não tem outra opção além de acoplar seu código com as classes de coleções específicas.

A ideia principal do padrão Iterator é extrair o comportamento de travessia de uma coleção para um objeto separado chamado um iterador. Além de implementar o algoritmo em si, um objeto iterador encapsula todos os detalhes da travessia, tais como a posição atual e quantos elementos faltam para chegar ao fim. Por causa disso, alguns iteradores podem averiguar a mesma coleção ao mesmo tempo, independentemente um do outro.

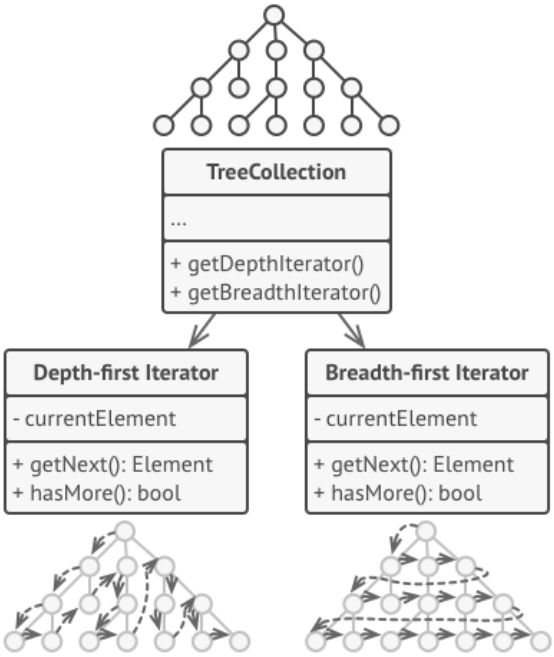

Figura 2.3: A mesma coleção pode ser atravessada de diferentes formas.

Geralmente, os iteradores fornecem um método primário para pegar elementos de uma coleção. O cliente pode manter esse método funcionando até que ele não retorne mais nada, o que significa que o iterador atravessou todos os elementos.

Todos os iteradores devem implementar a mesma interface. Isso faz que o código cliente seja compatível com qualquer tipo de coleção ou qualquer algoritmo de travessia desde que haja um iterador apropriado. Se você precisar de uma maneira especial para a travessia de uma coleção, você só precisa criar uma nova classe iterador, sem ter que mudar a coleção ou o cliente.

2.1 Estrutura

A interface **Iterador** declara as operações necessárias para percorrer uma coleção: buscar o próximo elemento, pegar a posição atual, recomeçar a iteração, etc.

Iteradores Concretos implementam algoritmos específicos para percorrer uma coleção. O objeto iterador deve monitorar o progresso da travessia por conta própria. Isso permite que diversos iteradores percorram a mesma coleção independentemente de cada um.

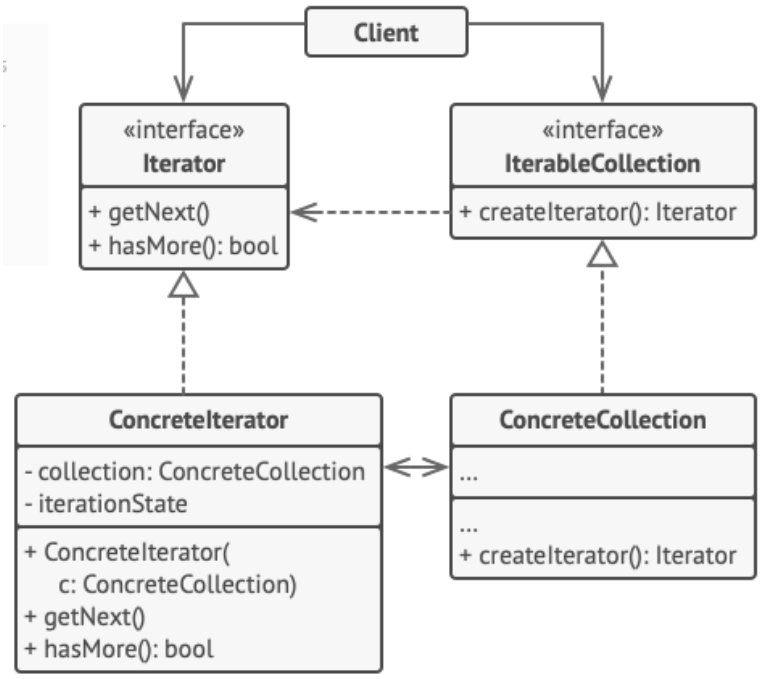

Figura 2.4: Estrutura do diagrama de classe do padrão Iterator.

A interface **Coleção** declara um ou mais métodos para obter os iteradores compatíveis com a coleção. Observe que o tipo do retorno dos métodos deve ser declarado como a interface do iterador para que as coleções concretas possam retornar vários tipos de iteradores.

Coleções Concretas retornam novas instâncias de uma classe iterador concreta em particular cada vez que o cliente pede por uma. Você pode estar se perguntando, onde está o resto do código da coleção? Não se preocupe, ele deve ficar na mesma classe. É que esses detalhes não são cruciais para o padrão atual, então optamos por omiti-los.

O **Cliente** trabalha tanto com as coleções como os iteradores através de suas interfaces. Dessa forma o cliente não fica acoplado a suas classes concretas, permitindo que você use várias coleções e iteradores com o mesmo código cliente.

Tipicamente, os clientes não criam iteradores por conta própria, mas ao invés disso os obtêm das coleções. Ainda assim, em certos casos, o cliente pode criar um diretamente; por exemplo, quando o cliente define seu próprio iterador especial.

2.2 Exemplo de implementação

Aplicação para gerenciamento de coleção de vídeos numa locadora. Conforme o diagrama abaixo, cada VídeoItem tem um nome e um preço. As outras classes farão a exibição do nome e o preço de cada filme existente na locadora. O diagrama foi implementado no arquivo disponibilizado no link do Github deste trabalho.

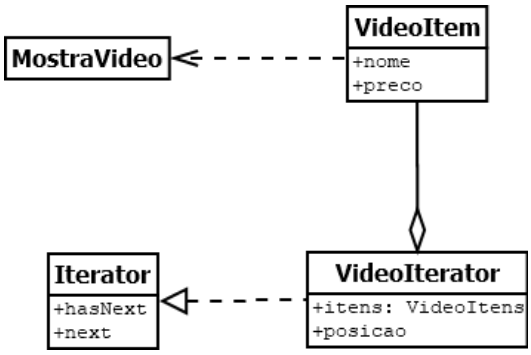

Figura 2.5: Diagrama de classe do exemplo da locadora de vídeo.

Na classe VideoItem foi criado o construtor com os atributos nome e preço. Através setPreço será possível modificar o preço de todos os filmes e exibí-los. A interface iterator tem dois métodos, next() e hasNext() mais um array do tipo VideoItem (cada um tem um nome e um preço) e o atributo posição. O método next() retorna o item numa posição e incrementa o array. O método hasNext() verifica se há elementos no array para serem percorridos retornando um booleano true or false que será utilizado no loop while, entregando o resultado esperado.

3. Padrão Composite

O Composite é um padrão de projeto estrutural que permite compor objetos em uma estrutura semelhante a uma árvore e trabalhar com eles como se fosse um objeto singular. A ideia desse padrão é montar uma árvore onde tanto as folhas (objetos individuais) quanto os compostos (grupos de objetos) sejam tratados de maneira igual. Em termos de orientação a objetos, isso significa aplicarmos

polimorfismo para chamar métodos de um objeto na árvore sem nos preocuparmos se ele é uma folha ou um composto.

“Compor objetos em estruturas de árvore para representarem hierarquias parte-todo. Composite permite aos clientes tratarem de maneira uniforme objetos individuais e composições de objetos.” (Design Patterns: Elements of Reusable Object-Oriented Software)

3.1 Estrutura

Suponha que tenhamos um questionário formado de questões, que podem estar agrupadas em blocos e blocos podem conter outros blocos. Podemos então ter a estrutura a seguir para um questionário:

Bloco A

—— Q1

—— Q2

—— Q3

Bloco B

—— Bloco B1

———- Q4

———- Q5

—— Bloco B2

———- Q6

—— Q7

Além da flexibilidade mostrada acima, queremos um meio fácil de exibir essa estrutura em tela/página, sem ter que nos preocupar se estamos exibindo um bloco ou uma questão. Neste cenário o Composite se encaixa perfeitamente e pode ser ilustrado pela estrutura abaixo:

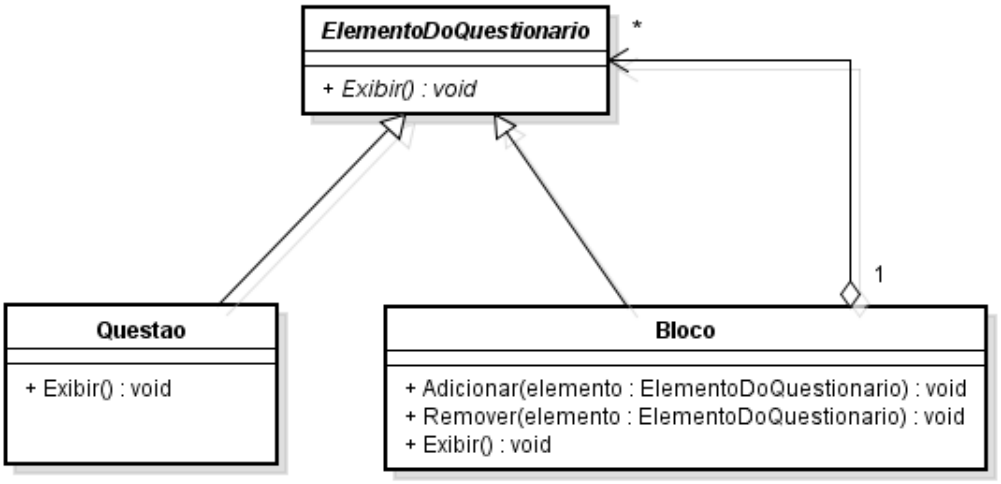

Figura 3.1: Diagrama de classe do padrão Composite para questionário

Dentro da estrutura do Composite, temos 3 participantes:

O **Componente** (ElementoDoQuestionario): é a interface que define métodos comuns às classes dentro da composição e, opcionalmente, define uma interface para acessar o objeto “pai” de um componente.

A **Folha** (Questao) é um componente que, como o nome indica, não possui filhos (está nas extremidades da árvore).

O **Composto** (Bloco) é um componente que, como o nome indica, é composto de outros componentes, que podem ser folhas ou outros compostos.

Notem no questionário acima que o Bloco B possui 3 filhos: 2 blocos (B1 e B2) e uma questão (Q7). Por sua vez, B1 possui 2 filhos (Q4 e Q5) e B2 possui um filho (Q6).

3.2 Exemplo de Implementação

Aplicação para gerenciamento de funcionários para um departamento de recursos humanos de uma empresa (Figura 3.2). A classe funcionário inclui os métodos necessários para as classes herdeiras (imprimir o nome do funcionário, adicionar funcionário ou remover funcionário) funcionando como interface, a classe supervisor inclui os mesmos métodos necessários para as classes Gerente e Presidente (Supervisores). Estas classes poderão mandar nas outras classes pois podem adicionar e remover elementos e ainda mandar as outras classes executar

métodos como imprimir os nomes dos funcionários ou outras ações, e portanto comandam a estrutura da árvore.

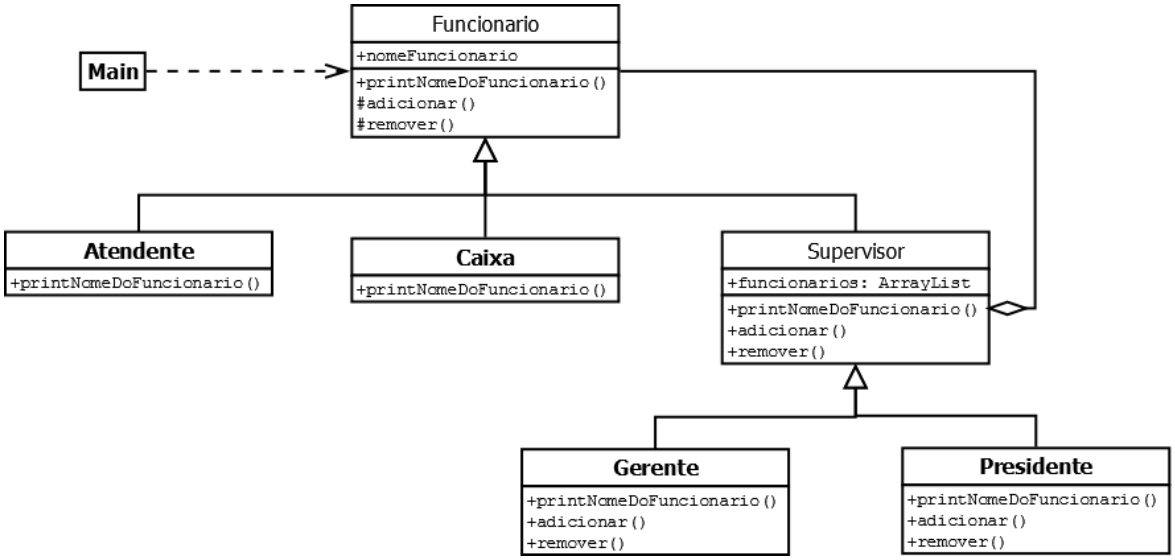

Figura 3.2: Estrutura de padrão Composite em aplicação de gerenciamento de recursos humanos

4. Padrão State

O padrão state permite que um objeto altere o seu comportamento quando o seu estado interno muda. O objeto parecerá ter mudado de classe. O padrão encapsula os estados em classes separadas e delega as tarefas para o objeto que representa o estado atual, nós sabemos que os comportamentos mudam juntamente com o estado interno.

Quando temos um objeto, onde além de lidarmos com a mudança de estado do mesmo ainda precisamos definir comportamentos diferentes para estados diferentes, tudo no mesmo código, tornamos nosso código mais complicado de entender e manter. Além disto, testes acabam ficam inviáveis. Neste caso, temos implementações diferentes do estado de nossa classe. Analisando a UML que representa esse padrão, temos a classe contexto, que pode ser uma das classes de domínio de nosso software, como Carro, Livro, Controle Remoto, etc., onde temos uma interface para o estado e as implementações de comportamento para cada estado.

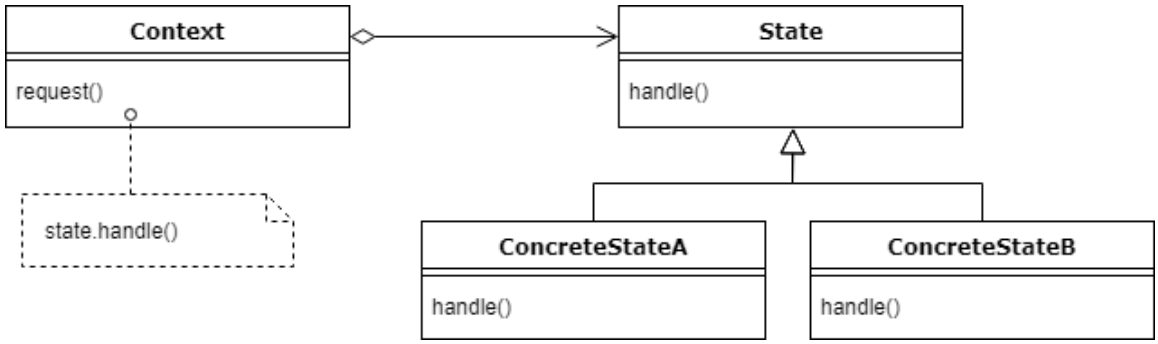

Figura 4.1: Diagrama de classe para o padrão State

No diagrama podemos ver que o contexto tem um estado associado que irá se modificar durante a execução do programa. Nosso contexto delegará o comportamento ao estado de implementação. Em outras palavras, todos os pedidos feitos serão executados pela implementação concreta do estado.

4.1 Exemplo de Implementação

No exemplo utilizado, deseja-se implementar um controle remoto de TV com um botão simples para realizar uma ação. Se o estado da TV é LIGADO (ON), ligará a TV e se o estado é DESLIGADO (OFF), desligará a TV. Assim, no exemplo cria-se uma interface State que definirá o método que deveria ser implementado para diferentes estados concretos e classes de contexto. Podemos ter dois estados: um para ligar a TV e outro para desligá-la. Assim, cria-se duas implementações concretas para estes comportamentos nos arquivos TVStartState e TVStopState. Para implementar o objeto de contexto que mudará seu comportamento baseado no estado interno cria-se o arquivo TVContext com os métodos setState, getState e DoAction. É importante notar que o contexto também implementa o estado e mantém uma referência do estado atual e encaminha o pedido para a implementação do estado. Para testar o padrão de estado para o controle remoto foi criado o programa TVRemote anexo neste trabalho. Ao ser executado fornece a saída dos estados da TV.

5 Padrão Proxy

O Proxy é um padrão de projeto estrutural que fornece um objeto que atua como um substituto para um objeto de serviço real usado por um cliente. Seu objetivo principal é encapsular um objeto através de um outro objeto que possui a mesma interface, de forma que o segundo objeto, conhecido como “Proxy”, controla o acesso ao primeiro, que é o objeto real. Um proxy recebe solicitações do cliente, realiza alguma tarefa (controle de acesso, armazenamento em cache etc.) e passa a solicitação para um objeto de serviço. Uma vez que neste padrão o objeto real é encapsulado, o objeto Proxy pode armazenar o resultado de um acesso em cache. Por exemplo, se um Remote Proxy é utilizado, o custo deste acesso remoto pode ser grande para a aplicação, neste caso o Proxy salva localmente os resultados em cache, diminuindo assim, a quantidade de acessos remotos.

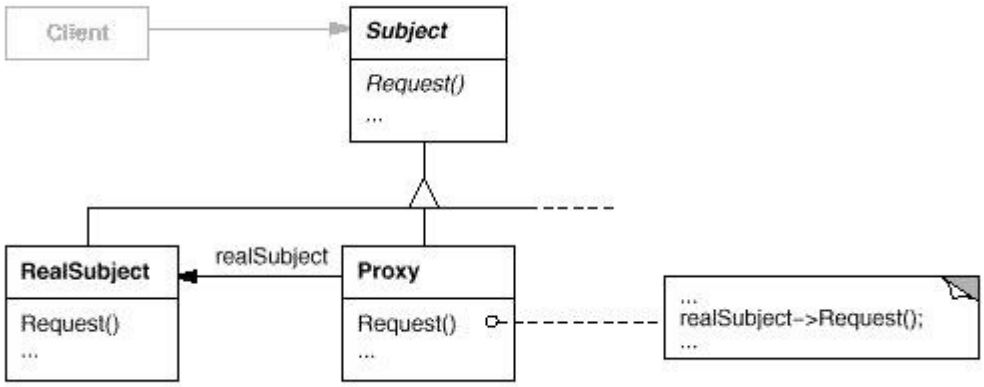

Figura 5.1: Diagrama de classe para o padrão Proxy

O diagrama demonstra que o objeto Proxy e o objeto real (RealSubject) implementam a mesma interface, neste caso chamada de Subject. O Proxy encapsula o acesso ao RealSubject.expressões aritméticas ter-se-ia para cada nó externo um valor associado e para cada nó interno um operador aritmético associado, esse algoritmo calcularia facilmente o resultado da expressão.

5.1 Exemplo de Implementação

Programa para visualização de lista de contato de cidades diferentes, pela pesquisa por cidade, a lista é exibida. Para mostrar a lista na tela é mais fácil e rápido do que a impressão. Na classe ProxyListaContato o método verifica se o atributo do tipo ListaCompleta foi instanciado, se não, ele o faz. Mas se existir, irá executar o método mais barato. Então só executará a operação mais cara e demorada uma vez, da segunda será a operação mais barata e rápida. A figura abaixo ilustra o diagrama de classes para o exemplo.

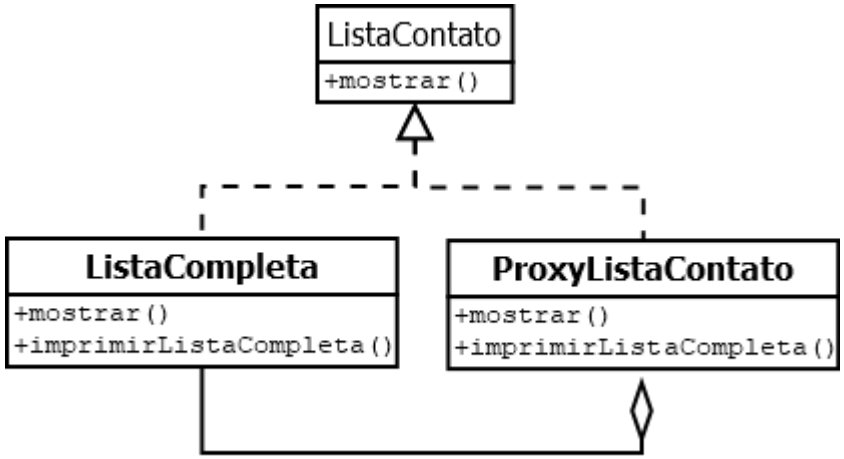

Figura 5.2: Diagrama de classe para o exemplo da lista de contatos no padrão Proxy