universidade de aveiro

Tópico A

ITERATOR

ABSTRACT FACTORY

Padrões e Desenho de Software
Mário Silva (93430)

Índice

Padrão 1 – Iterator

Introdução/Descrição

O padrão do **Iterador** é um padrão bastante usado de design comportamental, existem muitas estruturas/coleções que estão disponíveis aos programadores, cada uma destas deve possuir um **Iterador** que deixa iterar sobre os objetos de essas estruturas, mas de forma a não expor a sua implementação. Isto quer dizer que, se tivermos a construir algo que use Array Lists ou Hash Sets, não interessa, pois, esse **Iterador** percorre de qualquer forma, assim, o utilizador fica abstraído do tipo de implementação do conjunto iterado.

Problema

Pretende-se criar uma maneira de gerir um centro comercial que deve possuir um conjunto de lojas comerciais já por default, e essas lojas tem também um conjunto de itens.

Crie um código que permita ao gestor do centro de comercial, adicionar e remover lojas e ver os itens de cada loja. Tem como objetivo promover a abstração do tipo de estruturas/coleções. Para esses dois conjuntos, de lojas e de itens, use estruturas diferentes e demonstre que ao executar o código o gestor não tem acesso ao tipo de estruturas usadas nesses dois conjuntos.

Solução

Inicialmente foi criada uma interface **Iterator** que possui dois métodos, *hasNext()* e *next()*. Foi criada uma classe **Store** que possui uma estrutura, neste caso ArrayList, que agrupa os itens da loja, para tal foi criada também uma classe **Item** que possui apenas uma **String** como nome, por defeito são adicionados itens de roupa a todas as lojas do centro comercial. `ArrayList<Item> itemList;`

Foi também criada uma classe **Mall** que possui uma estrutura, foi usado um Array desta vez de objetos do tipo Store. `Store[] storeList;`. Tanto o **Mall** como a **Store** implementam uma interface **Collection** que tem um método `createIterator()` que retorna um objeto do tipo **Iterator**.

Para iterar as lojas foi criada uma classe **StoreIterator**, que implementa a interface **Iterator** e aceita um Array de objetos de tipo Store no seu construtor. Agora para iterar os itens da loja, foi criada a classe **ItemIterator** que neste caso aceita um ArrayList de objetos do tipo **Item**. A diferença entre estas duas classes é nos métodos *next()* e *hasNext()*, visto que trata-se de estruturas diferentes, por exemplo para aceder a um valor no Array é `storeList[index]`, e no ArrayList é `itemList.get(index)`.

Agora, para demonstrar a abstração das estruturas usadas na classe **Client**, a qual o gerente do centro comercial tem acesso, foi criada uma função *manageMall()* que permite ao gerente adicionar lojas e ver a lista de lojas bem como os itens de cada uma.

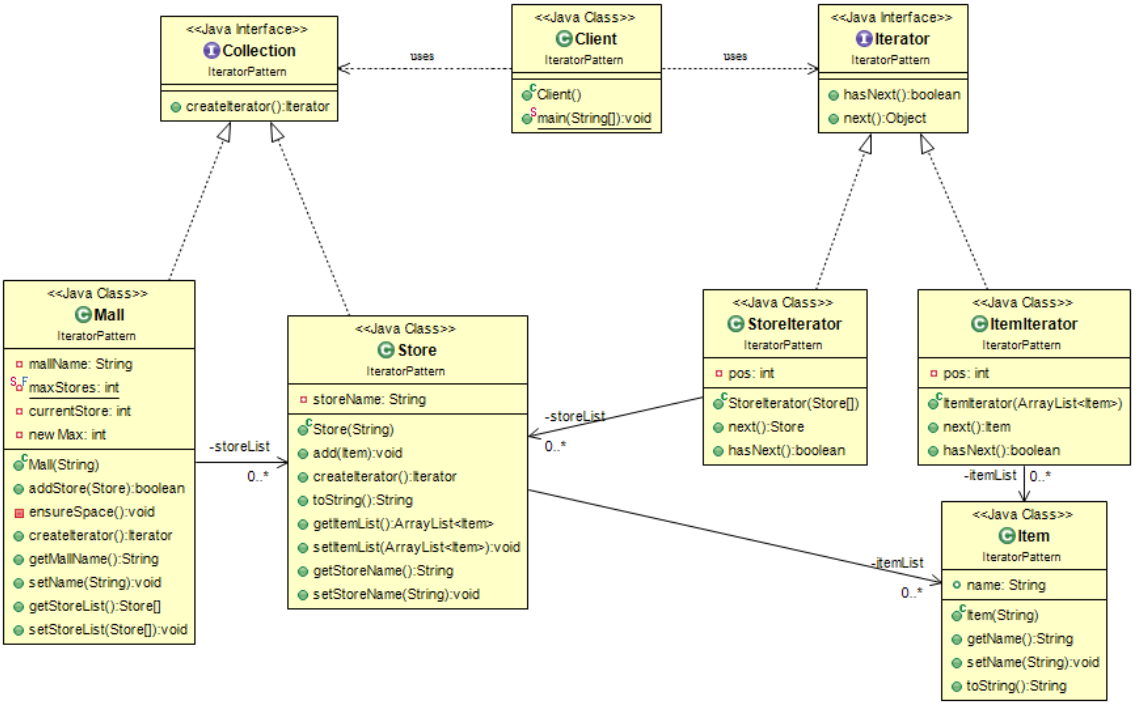
Para iterar as lojas é feito da seguinte forma:

```
Iterator storeIterator = mall.createIterator();
while (storeIterator.hasNext()) {
    Object currentStore = storeIterator.next();
}
```

E para iterar sobre os itens da loja:

```
Iterator itemIterator=((Store) currentStore).createIterator();
while (itemIterator.hasNext()) {
      Object item = itemIterator.next();
}
```

Como podemos observar não existem diferenças entre a maneira como se percorre cada um dos conjuntos, e o padrão **Iterator** é exatamente isso, fornece uma maneira de aceder aos elementos de um conjunto sem que o utilizador se aperceba do tipo de estruturas usadas para representar esses conjuntos.



Ao analisarmos o diagrama UML, podemos ver que a interface Collection desassocia o cliente da implementação da coleção dos objetos. Tanto o **Mall** como a **Store** implementam *createIterator()* que retorna um Iterador para as suas coleções, lojas e itens respetivamente, o **Mall** instancia um **StoreIterator** e a **Store** um **ItemIterator**. A interface **Iterator** fornece métodos de iterar a coleção, como o *next()* e o *hasNext()*, também poderia fazer outro tipo de funções não implementadas como pesquisa e remoção.

Referências

https://www.geeksforgeeks.org/iterator-pattern/

https://www.baeldung.com/java-abstract-factory-pattern

Padrão 2 – Abstract Factory

Introdução/descrição

O padrão de criação **Fábrica Abstrata** fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas. Este é um padrão muito simples, tal como o **Iterador,** mas é dos mais importantes certamente. O método da **Fábrica** é usado para criar apenas um produto, mas o de **Fábrica Abstrata** trata da criação de famílias de produtos relacionados ou dependentes, tendo por isso um nível mais alto na abstração que o padrao de **Fábrica**. Outra diferença é que **Fábrica** é um método único e uma **Fábrica Abstrata** é um objeto.

Resumidamente, a **Fábrica** abstrai a maneira como os objetos são criados, enquanto a **Fábrica Abstrata** também abstrai a maneira como as fábricas são criadas, que por sua vez abstraem a maneira de como os objetos são criados.

Problema

Pretende-se criar um stand automóvel, que possua veículos tanto de duas rodas como de quatro rodas e também possui um fundo monetário (inicialmente a 0). Para os de quatro rodas, use um simples carro e uma carrinha, e para os de duas rodas, uma bicicleta e um motociclo.

Ao serem criados esses veículos devem também ter um custo associado de fábrica (use valores aleatórios) e o fundo monetário do stand deve ser subtraído ao valor desse veículo (pode ficar a negativo). Ao serem inseridos no stand os carros devem ter um preço associado dado pelo administrador (considere 4 vezes superior ao valor de fábrica para haver lucro).

O programa deve correr de forma a que sempre que houver uma venda no stand o administrador possa dizer qual dos veículos vendeu e o lucro obtido com essa venda é atualizado no dinheiro do stand.

Solução

Foi elaborada uma classe **Vehicle** que possui um nome e um custo associado (custo de produção) e os métodos usados por todos os veículos.

De seguida, os respetivos veículos, que estendem a classe anterior. Visto que é necessário ter veículos de duas e de quatro rodas, foram feitas as classes **Van**, **Car**, **Bicycle** e **Motorcycle**.

Visto que foi pedido um stand automóvel, foi criada a classe **Stand,** que possui um nome, um conjunto de veículos e um custo associado a cada um, como pedido no enunciado, de forma a obter lucro, este custo é quatro vezes superior ao valor de fábrica. Possui também um fundo monetário a 0 inicialmente. Ao adicionar veículos esse fundo monetário é subtraído ao valor de fábrica do veículo e, ao vender é aumentado pelo preço de venda menos o preço de fábrica, ou seja, pelo lucro.

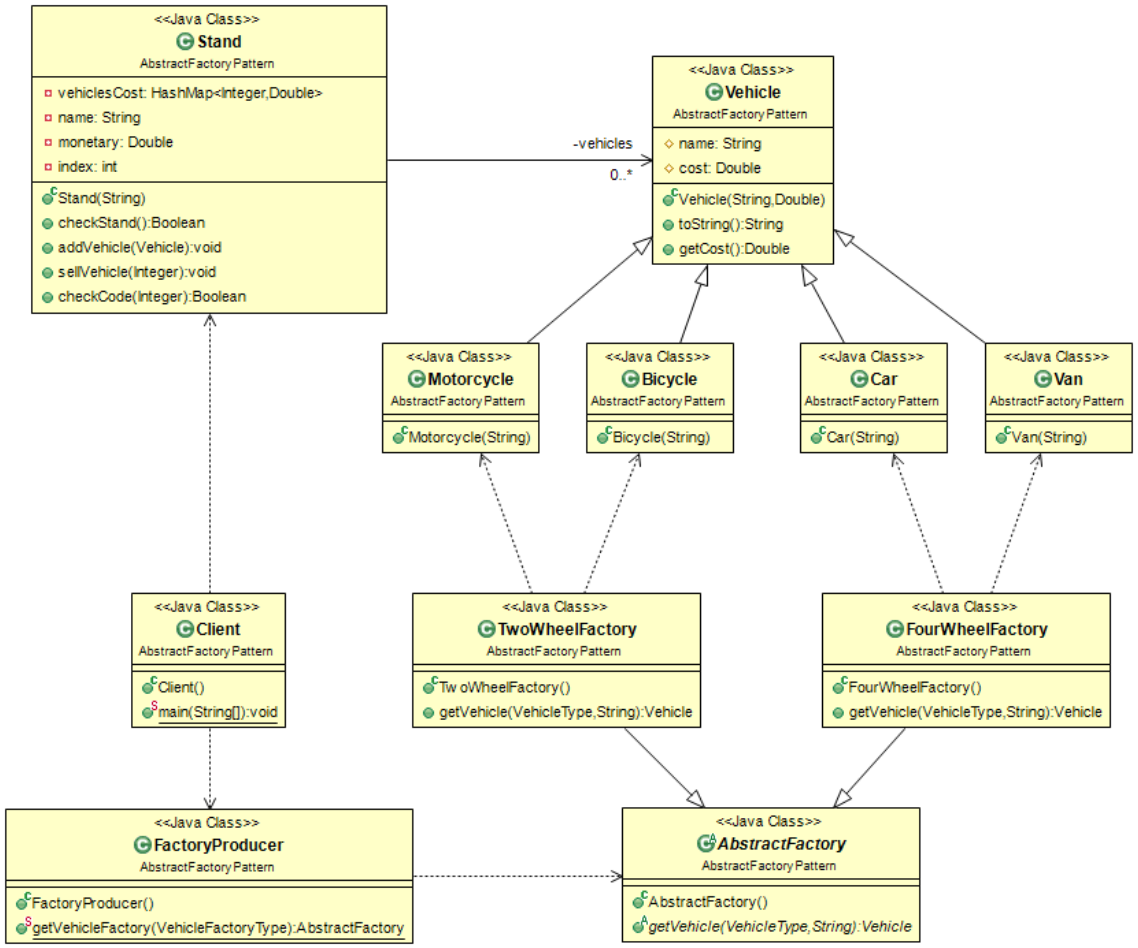
Inicialmente foram criadas as classes **AbstractFactory** que declara uma interface para operações de criação de veículos abstratos, recebe um enumerado para distinguir entre o tipo de veículos a serem criados (CAR, VAN, BICYCLE, MOTORCYCLE). Depois as classes **TwoWheelFactory** e **FourWheelFactory** que criam os veículos em concreto dependendo do tipo de veículos.

De seguida, para concluir o padrão foi feita a classe **FactoryProducer**, que fornece as fábricas de acordo com um enumerado passado pelo cliente (TWOWHEEL, FOURWHEEL), e assim, podemos obter os veículos correspondentes. Neste ponto, conseguimos observar uma das desvantagens do padrão de **Fábrica Abstrata**, o difícil suporte a novos tipos de veículos, requer a extensão da interface de fábrica, que envolve a alteração da classe **AbstractFactory** e de todas as suas subclasses.

Finalmente, para testar o padrão foi criada a classe **Client**. Aqui inicialmente é criado o **Stand**, depois as **Abstract Factories** para veículos de duas e quatro rodas (é adicionado por defeito, no C**lient**, um veículo de cada tipo existente ao Stand).

```
AbstractFactory twoWheelFactory = FactoryProducer.getVehicleFactory(Vehic
leFactoryType.TWOWHEEL);
AbstractFactory fourWheelFactory = FactoryProducer.getVehicleFactory(Vehi
cleFactoryType.FOURWHEEL);
Vehicle car = fourWheelFactory.getVehicle(VehicleType.CAR, "Fiat 500X");
Vehicle motorcycle = twoWheelFactory.getVehicle(VehicleType.MOTORCYCLE, "
Kawasaki Ninja 400");
```

Verificamos aqui um ponto crucial do padrão, para criar um objeto do tipo **Vehicle** não especificamos a sua classe concreta, mas sim apenas a **Abstract Factory** correta. Depois dá oportunidade ao gestor de adicionar mais veículos ao Stand ou vender os veículos possuídos num `while(true)` até escolher a opção de terminar o programa. Consegue observar à medida que adiciona ou vende veículos o fundo monetário a variar.



Referências

https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm

https://dzone.com/articles/factory-method-vs-abstract

https://dzone.com/articles/abstract-factory-design-pattern

https://www.geeksforgeeks.org/abstract-factory-pattern/