Refatoramento de Código

Exemplo extraído do livro *Refactoring: Improving the Design of Existing Code* (Fowler, 1999)

Parte 6: Adicionando Extensibilidade às Classificações de Veículos

I. Planejando e criando a hierarquia de classificações de Veículos

Ainda não temos polimorfismo porque apenas uma classe implementa a interface **`Alugavel`**. Considerando que desejamos uma solução flexível para adicionar novas classificações de **`Automovel`** no futuro sem causar grande impacto no código, vamos definir uma interface (ou classe abstrata) **`Classificacao`** e criar objetos concretos referente a cada tipo de classificação: **`BASICO`**, **`FAMILIA`** e **`LUXO`**.

Importante

Observe que a solução apresentada não foi criar uma superclasse **`Classificacao`** e definir subclasses **`BASICO`**, **`FAMILIA`** e **`LUXO`**. Sabe por que não vamos usar **herança**? Por que no nosso problema, um **`Automovel`** pode mudar de classificação durante sua vida e se usarmos herança, um **`Luxo`** (subclasse) não pode mudar para a subclasse **`BASICO`** durante sua execução. **Essa mutação não acontece com herança**. O objeto **`LUXO`** teria que ser destruído e outro objeto da hierarquia, **`BASICO`**, seria instanciado para resolver o problema, e isso traz algumas implicações.

Para lidar com esse problema, separamos o que é igual daquilo que muda e encapsulamos aquilo que muda em objetos diferentes. O resultado final está ilustrado na Figura 1. Observe que cada **`Automovel`** agora será composto de dois objetos:

- Um para o **`Automovel`** em si (o que é igual).
- Um para a classificação do **`Automovel`** (o que muda).

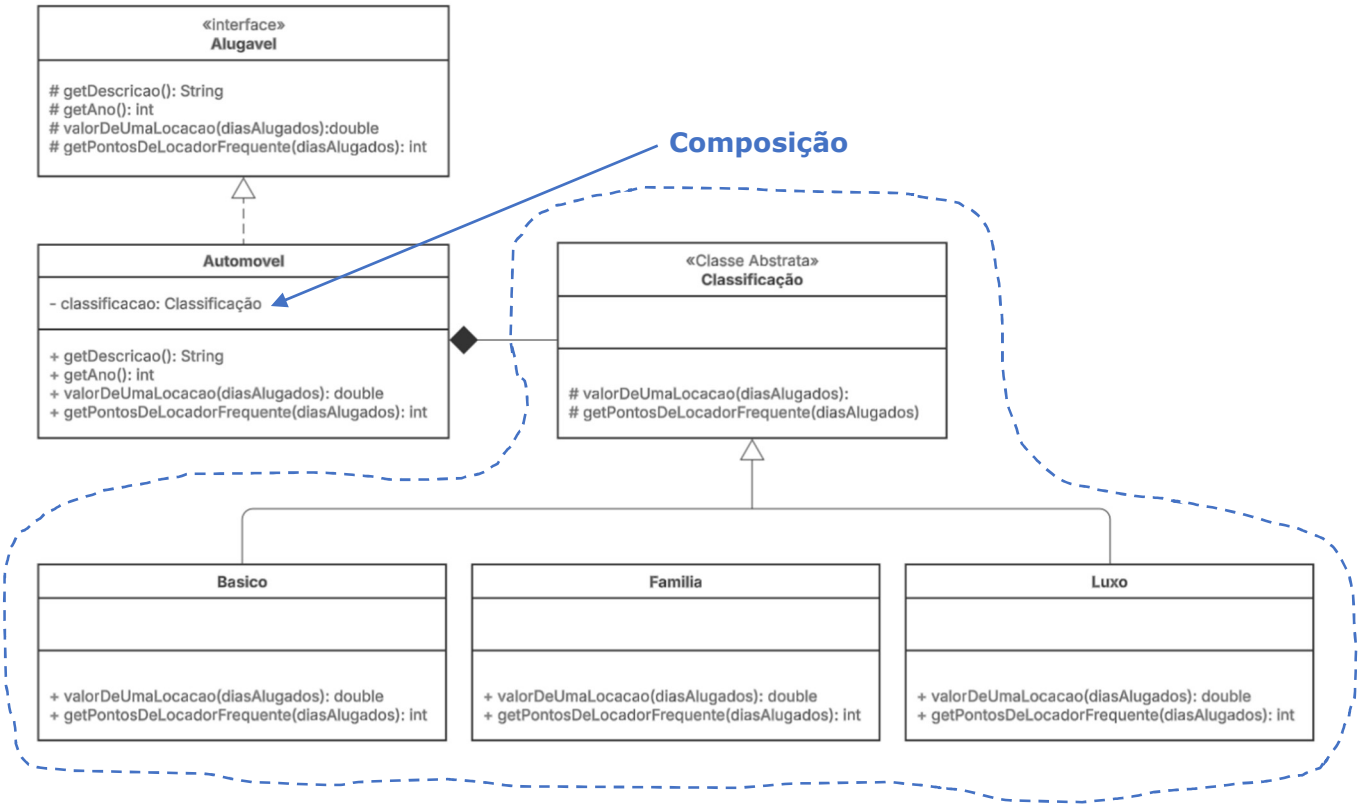

Figura 1: diagrama de classes da solução

Com essa estrutura, estamos definindo uma **composição de objetos**. Para implementar **`valorDeUmaLocacao()`**, por exemplo, o **`Automovel`** delega para o objeto concreto que implementa a inferface ou classe abstrata **`Classificacao`.**

Perceba agora por que essa solução é melhor:

(i) A composição **pode ser alterada** em tempo de execução, ou seja, o **`Automovel`** recebe um novo objeto concreto de classificação. Não deixará de ser um **`Automovel`**, porém, terá um objeto que sabe realizar seus cálculos de acordo com o tipo;

(ii) A flexibilidade da composição fará com seja frequentemente a melhor escolha em relação ao uso da herança;

(iii) **A herança ainda ocorre**, mas não no mundo dos **`Automoveis`**, e sim no mundo das **`classificações`**.

II. Tarefa (refactoring5)

Altere o código-fonte corrente de maneira que um **`Automovel`** seja dividido em dois objetos: um objeto **`Automovel`** que está associado a um objeto **`Classificação`** (Ver Figura 1). A abstração **`Classificação`** será implementada como uma classe abstrata e deve contemplar a seguinte estrutura:

```
public abstract class Classificacao {
  abstract int getCodigoDoPreco();
  abstract double getValorDaLocacao(int diasAlugado);
```

```
  int getPontosDeAlugadorFrequente(int diasAlugado) {
    return 1;
  }
}
```

- Crie as classes concretas **`Basica`**, **`Familia`** e **`Luxo`** herdando de **`Classificacao`**. Implemente os métodos abstratos
- Cada objeto concreto de **`classificação`** deve responder ao seu código de preço. Sendo assim, o método deve retornar sua constante correspondente. Então, não terá mais sentido manter o atributo **`codigoDoPreco`** na classe **`Automovel`**, porém, deve manter o método **`getCodigoPreco()`** que vai **delegar** a tarefa para o objeto concreto de classificação.
- O método **`getPontosDeAlugadorFrequente()`** possui um comportamento padrão de retornar **`PONTO_SIMPLES`** para **`Automoveis`** de classificação **`BASICA`** e **`FAMILIA`**. Portanto é conveniente oferecer esse método com uma implementação padrão, que retorne sempre **1**. Apenas a classe de classificação concreta relativa a **`LUXO`** deve sobrescrever o método e adicionar a lógica correspondente que incrementa o ponto de bonificação.

Quem ficará responsável por criar a classificação concreta é a própria classe **`Automovel`**. O método pertinente para realizar essa tarefa é o **`setCodigoDoPreco().`** A assinatura do método será a mesma, entretanto, dentro do método, a lógica deve criar a classificação concreta de acordo com o código do preço passado como argumento. Ou seja, a classe **`Automovel`** será o **creator** de classificações concretas.

Depois das mudanças, compile e rode o código novamente. A saída deve ser a mesma e a classe **`Locadora`** não deve ser afetada pelas mudanças. Reflita sobre como foi o impacto desta mudança no seu código.