O padrão não é o diagrama!

Um erro comum ao se aprender os primeiros padrões é achar que a implementação deve ser sempre exatamente como é mostrada no diagrama geral do padrão. O diagrama é apenas uma referência, pois a mesma solução pode ser implementada de diversas outras formas. Por exemplo, dependendo do contexto, `Implementacao` poderia ser implementada como uma classe abstrata ou a classe `Abstracao` poderia possuir mais de um *hook method* a ser implementado por suas subclasses. É por esse motivo, que ferramentas que geram código ou provêem uma estrutura de classes para a implementação de padrões são bastante criticadas por especialistas da área. É mais importante se preocupar em ter um solução adequada ao seu problema, do que seguir cegamente a estrutura do padrão.

3.3 Hook Classes

No capítulo anterior foi introduzido o conceito de *Hook Methods*, que são métodos que podem ser sobrescritos pelas subclasses como forma de estender e especializar o comportamento da classe. Nesse capítulo estamos vendo uma outra forma de alterar o comportamento das classes a partir da composição. Em vez de os pontos em que o comportamento pode variar serem definidos na mesma classe, eles são definidos em uma outra classe que compõe a classe principal. A essa classe que pode ser alterada damos o nome de *Hook Class*. A Figura 3.5 ilustra o conceito apresentado.

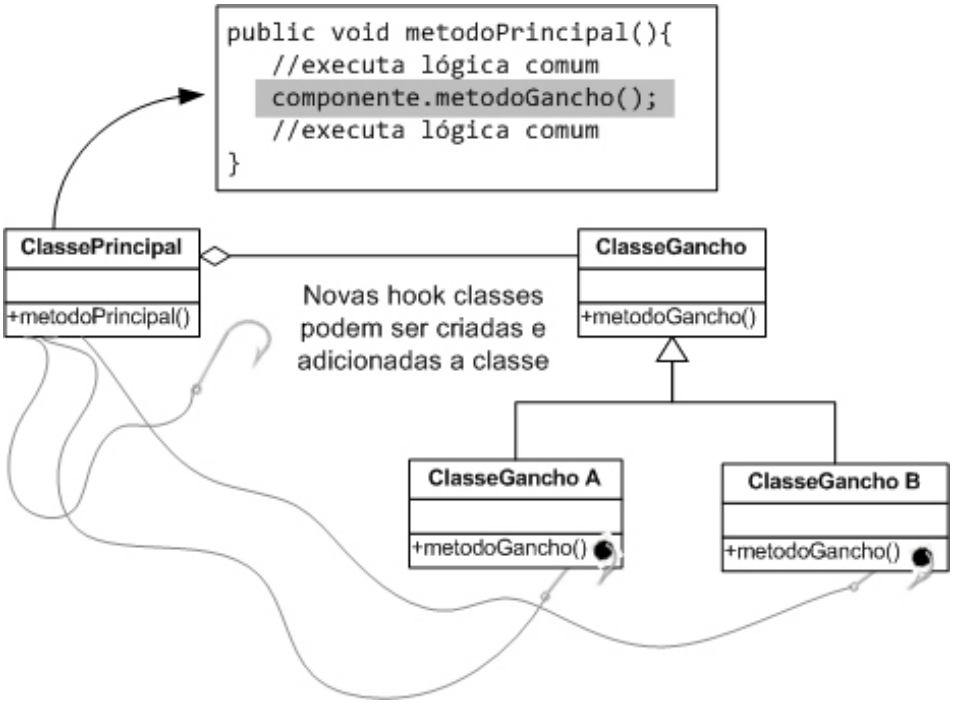

Figura 3.5: Representação do Conceito de Hook Classes

Da mesma forma que os *hook methods*, as *hook classes* são uma técnica utilizada pelos padrões para chegar a uma solução para um problema mais específico. O padrão **`Bridge`**, por exemplo, utiliza ao mesmo tempo um *hook method* e uma *hook class* como forma de separar dois pontos de extensão cujo comportamento pode variar de modo independente. Dessa maneira, é importante não apenas conhecer os padrões, mas compreender os princípios por trás de sua estrutura para entender melhor as consequências trazidas por uma decisão de design.

Diferenças Entre os Tipos de Hooks

Quando entendemos a ideia por trás dos *hook methods* e das *hook classes*, eles até que são bem parecidos. Em ambos os casos, a classe principal chama um método cuja implementação pode variar. No caso dos *hook methods* esse método está na mesma classe, podendo a implementação variar com a subclasse, e no caso das *hook classes* o método está em um objeto que compõe a classe, fazendo com que a implementação varie com a instância. Apesar de alguma semelhança, a utilização de uma técnica ou outra possui consequências bem diferentes.

A primeira delas está no momento em que a implementação que será utilizada

é escolhida. Quando um *hook method* é utilizado, a escolha é feita no momento em que o objeto é instanciado. Isso ocorre pois ao criarmos um novo objeto devemos escolher qual a classe concreta que será utilizada, e fazendo isso estamos escolhendo a sua implementação dos *hook methods*. Já no caso da utilização de *hook classes*, a implementação pode ser trocada a qualquer momento, bastando para isso trocar a instância que foi configurada. Observe que isso não é obrigatório, como no exemplo da classe `GeradorArquivo` apresentada nesse capítulo, que permitia a configuração da classe que a compunha apenas no construtor.

Nesse ponto, vale a pena retomar o exemplo do primeiro capítulo, no qual era necessário poder trocar a lógica que calculava o valor do estacionamento de acordo com o tempo e com o tipo de veículo. No cenário apresentado era extremamente importante poder trocar a implementação do cálculo após a criação do objeto `ContaEstacionamento`. Esse foi um dos motivos que fizeram a escolha do padrão **`Strategy`**, que utiliza uma *hook class*, ser adequado para a resolução do problema.

Outra questão que deve ser levada em consideração é o grau de dependência de diferentes pontos de extensão. Caso os *hook methods* sejam definidos na mesma classe, sendo eles na própria classe ou em uma *hook class*, as implementações ficam ligadas. Foi esse o problema apresentado no exemplo do `GeradorArquivo`, no qual pontos de extensão eram independentes mas ficavam definidos na mesma classe. Quando usamos apenas *hook methods*, pelo fato de não haver herança múltipla em Java, eles sempre ficam interligados. Ao utilizarmos *hook classes*, se colocarmos os métodos na mesma classe, o mesmo problema irá acontecer, porém se pusermos em classes diferentes, cada implementação poderá ser configurada de forma independente. O padrão **`Bridge`** aborda justamente essa questão, utilizando diferentes estratégias de extensão para que as implementações possam variar de forma independente.

Evoluindo o Modelo de Classes

Em um artigo clássico sobre padrões para evolução de frameworks [8], é mostrado que normalmente os pontos de extensão são primeiramente definidos utilizando herança, o que é conhecido como *whitebox framework*, para só em seguida evoluir para o uso de composição, chamado de *blackbox framework*. Por mais que a composição pareça ser mais vantajosa que a herança por suas características estruturais, é muito mais fácil deixar sua classe aberta à extensão através de herança.

O termo *whitebox framework* se refere a frameworks cujas classes permitem sua