Herdar ou não herdar de JPanel? Eis a questão!

Eu vejo essa questão da utilização ou não da herança é no desenvolvimento de aplicações desktop em Swing. Ao se criar uma nova tela para aplicação, uma dúvida comum é se essa classe deve estender a classe `JPanel` ou simplesmente utilizá-la. A extensão de `JPanel` pode fazer sentido a princípio, visto que a nova classe será um painel (obedecendo a famosa regra *"é um"*) e que pode ser adicionada em uma interface gráfica.

Por outro lado, se pensarmos nos princípios da herança, uma subclasse deve poder ser utilizada no lugar de sua superclasse. Raciocinando dessa forma, é muito fácil pensar em diversas situações em que a classe com a tela de uma aplicação não faria sentido de ser utilizada no lugar de um painel genérico. Isso também quebraria o contrato da superclasse, visto que diversos métodos, como para adição de componentes e configuração de layout deixariam de fazer sentido.

Somente verificar se a subclasse *é uma* superclasse pode não ser suficiente. Antes de utilizar herança verifique se o polimorfismo faz sentido, ou seja, se qualquer subclasse pode ser utilizada no lugar da superclasse. Em caso negativo, isso é um indício de que a herança está sendo utilizada de forma inadequada. Esse é conhecido como o **Princípio de Substituição de Liskov**, que defende que se uma classe é um subtipo de outra, então os objetos dessa classe podem ser substituídos pelos objetos do subtipo sem que seja necessário alterar as propriedades do programa.

O **`NullObject`** é um padrão que demonstra bem essa características da herança, pois ele permite que o código cliente possa ser utilizado, mesmo para o caso de um objeto nulo. Observe que outras implementações de `Carrinho`, como um que talvez acessa informações remotamente, também poderiam ser retornadas pela fábrica e utilizadas livremente.

2.2 Hook Methods

Um importante uso que pode ser feito da herança é para permitir a especialização de comportamento. Dessa forma, a superclasse pode fornecer uma base para uma determinada funcionalidade, a qual invoca um método que somente é definido pela

superclasse. Esse método funciona como um ponto de extensão do sistema é chamado de método-gancho, ou em inglês, **hook method**.

A Figura 2.1 representa o conceito de *hook method*. A superclasse possui um método principal público que é invocado pelos seus clientes. Esse método delega parte de sua execução para o *hook method*, que é um método abstrato que deve ser implementado pela subclasse. Ele funciona como um "gancho" no qual uma nova lógica de execução para a classe pode ser "pendurada". Cada subclasse o implementa provendo uma lógica diferente. Como essa lógica pode ser invocada a partir do mesmo método público, definido na superclasse, os *hook methods* permitem que o objeto possua um comportamento diferente de acordo com a subclasse instanciada.

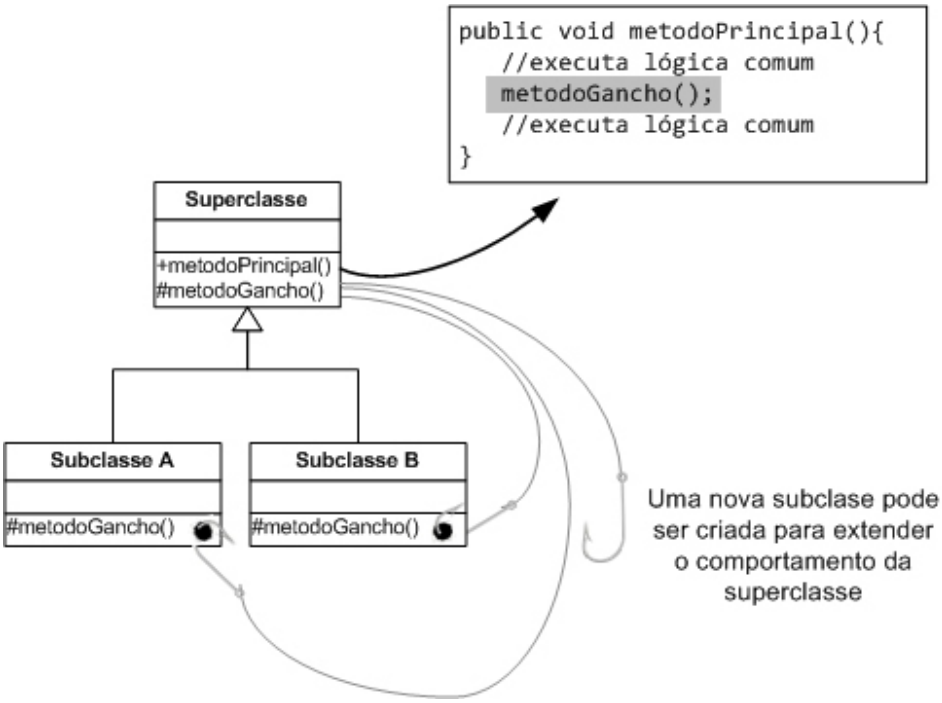

Figura 2.1: Representação de Hook Methods

Essa prática é muito utilizada em frameworks para permitir que as aplicações possam especializar seu comportamento para seus requisitos. Nesse caso, o framework provê algumas classes com *hook methods* que precisam ser especializadas pela aplicação. Sendo assim, a aplicação deve estender essa classe e implementar o *hook method* de forma a inserir o comportamento específico de seu domínio. Imagine, por exemplo, um framework que realiza o agendamento de tarefas. Usando

essa estratégia ele poderia prover uma classe para ser estendida, na qual precisaria ser implementado um *hook method* para a definição da tarefa da aplicação. Enquanto isso, na classe do framework estaria a lógica de agendamento que chamaria o método definido pela aplicação no momento adequado.

Um exemplo de uso dessa técnica está na Servlets API [21], na qual um servlet precisa estender a classe `HTTPServlet`. Essa classe possui o método `service()` que é invocado toda vez que ele precisa tratar uma requisição HTTP. Esse método chama outros métodos, como `doPost()` ou `doGet()`, que precisam ser implementados pela subclasse. Neles ela deve inserir a lógica a ser executada para tratar a requisição recebida. Nesse caso, esses métodos possuem uma implementação *default* vazia e precisam ser implementados somente se necessário.

2.3 Revisando modificadores de métodos

Quando aprendemos o básico da linguagem, aprendemos modificadores de acesso e outros tipos de modificadores de métodos e qual o seu efeito nas classes. O que nem todos compreendem é quando cada um deles precisa ser utilizado e qual o tipo de mensagem que se passa com cada um em termos de design. A seguir estão relacionados alguns modificadores de acesso e qual o recado que uma classe manda para duas subclasses quando utiliza cada um deles:

- `abstract`: Um método abstrato precisa obrigatoriamente ser implementado em uma subclasse concreta. Isso significa que esse é um *hook method* que foi definido na superclasse e obrigatoriamente precisa ser definido.

- `final`: De forma contrária, um método do tipo final não pode ser sobrescrito pelas subclasses e por isso nunca irá representar um *hook method*. Esses métodos representam funcionalidades da classe que precisam ser imutáveis para seu bom funcionamento. Costumam ser os métodos que invocam os *hook methods*.

- `private`: Os métodos privados só podem ser invocados dentro da classe que são definidos. Dessa forma, eles representam métodos de apoio internos da classe e nunca poderão ser substituídos pela subclasse como um *hook method*.

- `protected` e `public`: Todos os métodos públicos e protegidos, desde que não sejam `final`, são candidatos a *hook method*. É isso mesmo! Qualquer