ACH 2003 - Computação Orientada a Objetos

Herança e interfaces

Prof. Flávio Luiz Coutinho
flcoutinho@usp.br

Herança

É um mecanismo que permite a declaração de uma nova classe (classe derivada), baseada na declaração de uma outra já existente (classe base).

Atributos e métodos da classe base são “incorporados” (herdados) automaticamente na classe derivada, sem a necessidade de copy e paste ;)

A classe derivada pode estender as funcionalidades da classe base, pelo acréscimo de novos atributos e métodos.

Pode também redefinir algum comportamento da classe base, fornecendo uma nova declaração de um método já existente (sobreescrita, ou *overriding*).

Define uma relação hierárquica de tipos e tipos derivados, o que permite explorar o que é chamado de polimorfismo.

Herança

Vantagens:

- Criação de hierarquia de tipos que permite o uso do polimorfismo.
- Criação de novos tipos escrevendo o mínimo de código.
- Favorece reutilização de código já existente.

Quando usar?

- Se haverá, efetivamente, reaproveitamento da implementação herdada.
- Se é importante que a nova classe (derivada) seja hierarquicamente subtipo da classe base, para que o polimorfismo possa ser explorado.
- Ainda assim, há ressalvas quanto ao uso de herança (veremos mais tarde)

Herança: exemplo

Classe **Produto** (aproveitamos também para rever algumas questões de encapsulamento):

- Versão inicial da classe **Produto**.
- Nova demanda por uma classe similar, porém com funcionalidade extra: contabilizar o número de vezes que houve alteração de preço.
- Versão 0: solução “copy e paste”. Há problemas e limitações? Sim, código redundante e impossibilidade de explorar o polimorfismo.

Herança: exemplo

Duas soluções usando herança:

- Versão 1: atributo *preco* da superclasse com nível de acesso protected, e redefinição total do método *setPreco* na classe derivada, em que é feito acesso direto ao atributo. Temos redundância de implementação (apesar de a lógica envolvida na atualização do atributo ser extremamente simples no exemplo considerado) e, de certa forma, uma “brecha” no encapsulamento.
- Versão 2: atributos da superclasse são mantidos privados, e a implementação já existente do método *setPreco* da superclasse é aproveitada para implementar a extensão deste mesmo comportamento na subclasse. Não há redundância de implementação, e o encapsulamento é mantido.

Interfaces

Uma interface define um conjunto de funcionalidades (métodos) que todas as classes que a implementam são obrigadas a fornecer.

É como se fosse um “contrato”. Se uma classe declara seguir o contrato, ela terá certas obrigações (fornecer implementações dos métodos especificados no contrato).

Define uma categoria de classes que têm em comum as mesmas operações, mas cujas implementações podem ser muito diferentes entre si.

Define o que uma classe deve fazer, mas não como fazer.

Assim como com herança, interfaces definem estruturas hierárquicas, o que permite a exploração do polimorfismo.

Interfaces: exemplo

Considere a classe **Lista** que contém diversas versões de um método para realizar filtragem dos elementos (ou seja, devolvem novas listas contendo apenas os elementos que satisfazem o critério especificado em cada variação do método):

- Versão inicial: há redundância de implementação nos diversos métodos "*filtraAlgumaCoisa"*. Possuem a mesma estrutura em comum (responsável pela criação da lista resposta, iteração pelos elementos da lista e adição dos elementos selecionados na lista resposta) mudando apenas a condição que define a seleção.

Interfaces: exemplo

Solução usando interface:

- Diversas classes diferentes implementam diversos critérios de filtragem distintos, mas todas com um papel em comum, que é decidir se um elemento qualquer deve (ou não) ser selecionado no processo de filtragem.
- Declaração da interface **Criterio** que é supertipo comum a todas as classes que implementam um critério (estas declaram que implementam a interface).
- Uma única versão do método *filtra* na classe **Lista** recebe um objeto do tipo **Criterio** como parâmetro**.** Este encapsula algum critério de filtragem específico, sem que a implementação do método saiba exatamente qual. Uma única versão do método está apta a realizar a filtragem por qualquer critério que venha a existir, desde que este seja implementado em uma classe que respeite a interface.

Composição como alternativa a herança

Revisitando o exemplo das classes Produto e ProdutoContador:

- Dá para resolver o problema de outra forma? Sim, com composição!
- Há benefícios e vantagens? Sim, o relacionamento entre a classe “base” e a “derivada” é estabelecido em tempo de execução, o que traz maior flexibilidade.
- Há desvantagens? À primeira vista, parece menos intuitivo do que usar herança diretamente. Solução define uma estrutura de classes/interfaces um pouco mais complexa.

Voltando ao exemplo da classe Lista…

Revisitando o exemplo da classe Lista (e critérios de filtragem):

- Criação de critérios pela composição de outros critérios.
- Para refletir:
  - Uso de herança para criar subclasses de Lista, com cada subclasse implementando um critério de filtragem diferente: considerações…
  - Uso de herança para criação dos diversos critérios: seria possível? Sim, não haveria tanta flexibilidade. Geraria um número de classes muito mais elevado.

Lembretes

Palavra chave super: análoga ao this, mas refere-se às coisas da superclasse.

Lembretes

Na linguagem Java, toda classe possui apenas uma superclasse.

- Não existe herança múltipla, como no C++ por exemplo.

Lembretes

Classes que explicitamente não derivam de nenhuma outra, implicitamente derivam da classe Object:

- toString()
- equals()
- clone()

Lembretes

Palavra chave abstract: definem lacunas na superclasse, que devem ser preenchidas em subclasses.