Aula 4

Herança

O primeiro passo no processo de aprendizagem da programação orientada a objetos é compreender a filosofia básica de organizar a solução de uma tarefa como a interação de componentes de software fracamente acoplados. Esta abordagem organizacional foi o ponto central da solução apresentada em sala de aula para o "Problema das Oito Rainhas". O próximo passo é aprender como organizar classes dentro de uma hierarquia baseada no conceito de herança.

Herança é um mecanismo para derivar novas classes a partir de classes existentes através de um processo de refinamento. Uma classe derivada herda a representação de dados e operações de sua classe base, mas pode seletivamente adicionar novas operações, estender a representação de dados ou *redefinir* a implementação de operações já existentes. Uma classe base proporciona a funcionalidade que é comum a todas as suas classes derivadas, enquanto que uma classe derivada proporciona a funcionalidade adicional que especializa o seu comportamento.

A terminologia usada para herança engloba vários termos que são usados como sinônimos:

- **Classe derivada** ou **subclasse** ou **classe filha**: é uma classe que herda atributos e métodos de outra classe.
- **Classe base** ou **superclasse** ou **classe pai**: é uma classe a partir da qual classes novas podem ser derivadas.
- **Classes ancestrais**: são aquelas das quais uma superclasse herda.
- **Classes descendentes**: são aquelas que herdam de uma subclasse.

Uma Noção Intuitiva de Herança

Vamos retornar ao exemplo da entrega de flores, visto na aula 1. Como foi visto naquela aula, há um certo comportamento esperado de todos os floristas, não porque eles são floristas mas simplesmente porque eles são *vendedores*. Por exemplo, nós sabemos que Florisvaldo pedirá dinheiro pelo serviço que pedimos para ele fazer e, por sua vez, ele nos dará uma nota fiscal. Estas atividades não são exclusivas dos floristas, mas comuns a vendedores de carro, corretores de imóveis, vendedores de revistas e muitos outros vendedores. Como imaginado, nós associamos certo comportamento à categoria geral `Vendedor` e, como `Floristas` são uma forma especializada de vendedores, o comportamento foi automaticamente identificado com a subclasse.

Em linguagens de programação, herança significa que o comportamento e os dados associados às classes filhas são sempre uma *extensão* (isto é, um conjunto maior) das propriedades associadas às classes pai. Uma subclasse deve ter todas as propriedades das classes pais e outras propriedades também. Por outro lado, desde que uma classe filha é uma forma mais especializada (ou restrita) da classe pai, ela é também, em um certo sentido, uma *contração* do tipo do pai. Esta "tensão" entre herança como expansão e herança como contração é uma fonte para muito da potência de herança, mas ao mesmo tempo também causa muita confusão no emprego e entendimento do mecanismo de herança.

Herança é sempre transitiva, tal que uma classe pode herdar características de superclasses que estão a muitos níveis acima do seu. Isto é, se a classe `Cachorro` é uma subclasse da classe `Mamífero`, e a classe `Mamífero` é uma subclasse da classe `Animal`, então `Cachorro` herdará atributos tanto de `Mamífero` quanto de `Animal`. Um fator complicador em nossa noção intuitiva de herança é o fato que subclasses podem *sobrescrever* comportamento herdado de classes pai. Por exemplo, a classe `Ornitorrinco` sobrescreve o comportamento de reprodução herdado da classe `Mamífero`, pois ornitorrincos põem ovos. Nós lidaremos com esta questão em aulas subseqüentes.

Formas de Herança

Herança é usada de várias formas. Vamos examinar algumas das formas mais comuns:

- **Especialização**. Provavelmente, o uso mais comum de herança é por especialização. Neste uso, a classe nova é uma forma especializada da classe pai, mas satisfaz as especificações da classe pai em todos os aspectos relevantes. Por exemplo, uma classe `Janela` fornece as operações gerais de janela (mover, redimen-

sionar, transformar em ícone e assim por diante). Uma subclasse especializada `JanelaTexto` herda as operações de janela e *adiciona* facilidades que permitem a exibição e edição de texto. Devido ao fato que `JanelaTexto` satisfaz todas as propriedades esperadas de uma janela em geral, reconhecemos esta situação como um exemplo de especialização.

- **Generalização**. Usar herança por generalização é, de certa forma, o oposto de especialização. Aqui, uma subclasse estende o comportamento da classe pai para criar um tipo mais geral de objeto. Generalização é freqüentemente aplicada quando construímos uma base de classes existentes que não desejamos modificar ou não podemos modificar. Como exemplo, considere um sistema gráfico no qual uma classe `Janela` tenha sido definida para exibir desenhos com um fundo preto-e-branco. Nós poderíamos criar uma subclasse `JanelaColorida` que permitisse uma cor de fundo diferente de preto-e-branco. Isto poderia ser feito com a adição de um campo para armazenar a cor de fundo e por sobrescrever o código herdado de `Janela` que especifica a cor de fundo.

- **Limitação**. Limitação ocorre quando o comportamento da subclasse é menor ou mais restritivo do que o comportamento da classe pai. Assim como generalização, limitação ocorre mais freqüentemente quando um programador está construindo sobre uma base de classes existentes que não deveria, ou não pode, ser modificada. Por exemplo, uma biblioteca existente de classes oferece uma fila de duas extremidades (ou uma estrutura de dados *deque*). Elementos podem ser adicionados ou removidos de qualquer uma das extremidades, mas o programador deseja escrever uma classe `Pilha`, que possui a propriedade que os elementos podem ser adicionados ou removidos de apenas uma extremidade da pilha. Então, o programador pode definir uma classe `Pilha` como uma subclasse da classe `Deque` e modificar ou sobrescrever métodos existentes e indesejáveis de modo que eles produzam uma mensagem de erro se forem usados. Estes métodos sobrescrevem métodos existentes e eliminam a funcionalidade deles, o que caracteriza a herança por limitação.

- **Especificação**. Um uso freqüente de herança é para garantir que classes mantenham uma certa interface comum, isto é, implementem os mesmos métodos. A classe pai pode ser uma combinação de operações implementadas e operações que são "deixadas" para ser implementadas pelas classes filhas. Freqüentemente, não há mudança de interface de nenhuma ordem entre a classe pai e a classe filha: a classe filha meramente implementa o comportamento descrito, mas não implementado, pela classe pai. Especificação é de fato um caso especial de especialização, exceto que as subclasses não são refinamentos de um tipo existente, mas sim *realizações* de uma especificação abstrata e incompleta. Em tais casos, a classe pai é algumas vezes conhecida como *classe de especificação abstrata*. Uma

situação interessante para o uso de especificação é exatamente o Trabalho 1 do nosso curso. A solução poderia conter uma classe abstrata `ObjetoGráfico` que descreveria, mas não implementaria, métodos para desenhar o objeto e responder a uma colisão com uma bola. As classes `Bola`, `Parede` e `Caçapa` seriam subclasses de `ObjetoGráfico` e forneceriam implementações para os métodos dela.

- **Combinação**. Uma situação comum é uma subclasse representar uma combinação de características de duas ou mais classes pai. Um monitor de ensino pode possuir características tanto de um professor quanto de um estudante e pode, portanto, comportar-se como ambos. A capacidade de uma classe herdar de duas ou mais classes pais é conhecida como *herança múltipla*. Esta forma de herança será objeto de estudo de uma aula futura devido a sua sutileza e complexidade.

A lista acima de usos de herança não descreve todos os casos, mas apenas os mais conhecidos. Além disso, duas ou mais descrições podem ser aplicadas a uma única situação.

Benefícios e Custos do Uso de Herança

Alguns dos benefícios alcançados com o uso de herança são:

- **Reusabilidade de código**. Quando comportamento é herdado de uma outra classe, o código que fornece aquele comportamento não precisa ser reescrito. Isto parece óbvio, mas as implicações são importantes. Muitos programadores gastam muito do tempo deles reescrevendo código que eles escreveram muitas vezes antes. Por exemplo, para encontrar um padrão em uma cadeia ou inserir um novo elemento em uma tabela. Com técnicas de orientação a objetos, estas funções podem ser escritas apenas uma vez e reutilizadas.

- **Compartilhamento de código**. Compartilhamento de código ocorre em muitos níveis com técnicas orientadas a objetos. Uma forma de compartilhamento se dá quando muitos usuários ou projetos podem usar as mesmas classes. Uma outra ocorre quando duas ou mais classes desenvolvidas por um único programador como parte de um projeto herda de uma única classe pai. Quando isto acontece, dois ou mais tipos de objetos compartilharão o código que eles herdaram.

- **Consistência de interface**. Quando duas ou mais classes herdam da mesma superclasse, podemos estar certos de que o comportamento que elas herdaram será o mesmo em todas as classes. Logo, é mais fácil garantir que interfaces para

objetos semelhantes sejam, de fato, semelhantes e que o usuário não se depare com um conjunto confuso de objetos que são quase os mesmos mas se comportam e interagem de forma bem diferente.

- **Componentes de software**. Herança permite que programadores construam componentes de software reutilizáveis. O objetivo é permitir o desenvolvimento de aplicações novas que requeiram pouca ou nenhuma codificação. Várias bibliotecas de classes reutilizáveis já se encontram disponíveis comercialmente e podemos esperar muito mais com o passar do tempo.

- **Prototipação rápida**. Quando um sistema de software é grandemente construído com base em componentes reutilizáveis, o tempo de desenvolvimento pode ser concentrado em entender a parte nova do sistema. Logo, sistemas de software podem ser gerados mais facilmente e mais rapidamente, levando a um estilo de programação conhecido como *prototipação rápida* ou *programação exploratória*.

- **Ocultamento de informação**. Um programador que reutiliza um componente de software necessita apenas entender a natureza do componente e sua interface. Não há a necessidade do programador ter informação detalhada a respeito de fatos como, por exemplo, as técnicas usadas para implementar o componente. Logo, a interconexão entre sistemas de software é reduzida e, conseqüentemente, a complexidade também é reduzida.

Embora os benefícios de herança em programação orientada a objetos sejam grandes, herança, assim como outros mecanismos, não possui custo zero. Por esta razão, devemos considerar o custo de técnicas orientadas a objetos e, em particular, o custo de herança:

- **Velocidade de execução**. Dificilmente, ferramentas de software de propósito geral são tão rápidas quanto aquelas desenvolvidas cuidadosamente para um propósito específico. Logo, métodos herdados, que devem lidar com subclasses arbitrárias são freqüentemente mais lentos do que código especializado. Ainda assim, a preocupação com eficiência é geralmente desnecessária. Primeiro, a diferença é freqüentemente pequena. Segundo, a redução em velocidade de execução pode ser balanceada por um aumento da velocidade de desenvolvimento de software. Finalmente, a maioria dos programadores não possui a mínima idéia de como o tempo de execução está sendo usado no programa. É muito melhor desenvolver um sistema que funcione, monitorá-lo para descobrir onde o tempo de execução está sendo usado e melhorar aquelas seções, do que gastar uma quantidade desordenada de tempo se preocupando com eficiência no início do projeto.

- **Tamanho do programa**. O uso de qualquer biblioteca de software em geral acarreta o aumento de tamanho do programa, o que não acontece com sistemas construídos através de um projeto específico. Embora este gasto possa ser substancial, quando o custo de memória for reduzido a níveis insignificantes. Conter custos de desenvolvimento e produzir código de alta qualidade e livre de erro são preocupações que se tornaram muito mais importantes do que limitar o tamanho dos programas.

- **Overhead de envio de mensagens**. Muito do que tem sido feito do fato que envio de mensagens é por natureza uma operação mais custosa do que chamada de procedimento. Assim como velocidade de execução, entretanto, a preocupação neste caso também é supervalorizada, pois a diferença nos custos de envio de mensagens e chamada de procedimento é marginal. Talvez, duas ou três instruções adicionais de linguagem de montagem e um gasto adicional de 10 porcento do tempo total. Este custo, como muitos outros, pode ser balanceado pelos benefícios das técnicas de orientação a objetos.

- **Complexidade do programa**. Embora a programação orientada a objetos seja tida como uma solução para a complexidade do software, o uso demasiado de herança pode simplesmente substituir uma forma de complexidade por outra. Entender o fluxo de controle de um programa que utiliza herança pode requerer várias varreduras no grafo de herança. Isto é conhecido como *o problema do iô-iô*.