INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
PARAÍBA
Campus Cajazeiras

Engenharia de Controle e Automação
Programação Orientada a Objetos – 2021.1

Professor: Leandro Luttiane da Silva Linhares
E-mail: leandro.luttiane@ifpb.edu.br

Documento Texto

Classes Abstratas e Polimorfismo

Na semana anterior da disciplina de **Programação Orientada a Objetos (POO)** do **Curso de Engenharia de Controle e Automação**, iniciamos o estudo do conceito de polimorfismo em nossas hierarquias de classe. A partir da definição de funções virtuais, vimos que é possível sobrescrever nas classes derivadas, funções-membro de mesmo nome que foram inicialmente definidas nas chamadas classes básicas.

O uso do conceito de polimorfismo permitiu, por exemplo, que a partir de um ponteiro de classe básica, fosse possível apontar para um objeto instanciado de classe derivada e efetuar a chamada de uma função-membro `virtual` que foi sobrescrita por esta classe. Em outras palavras, a chamada passa a ser definida por meio de vinculação dinâmica. Assim, dependendo do objeto que está sendo apontado por um ponteiro de classe básica é que se define, em tempo de execução, a função-membro que será executada: a que foi definida na classe básica, caso o objeto apontado seja da classe básica, ou a que foi sobrescrita na classe derivada, caso o objeto apontado seja de classe derivada. É o objeto apontado e não o *handle* que irá definir o método a ser chamado.

Nesta semana iremos finalizar o nosso estudo sobre polimorfismo. Inicialmente serão apresentadas as classes abstratas e as funções virtuais puras e, em seguida, será analisado um estudo de caso que simula um sistema de folha de pagamento. Antes de iniciarmos, é importante comentar que o conteúdo apresentado neste documento tem como base o ***Capítulo 13 do livro C++: Como Programar*, de Deitel e Deitel (2006)**. Em conjunto com este texto, também estão sendo disponibilizados exemplos de código extraídos do livro mencionado. Sugere-se fortemente que estes arquivos sejam abertos quando as suas linhas de código forem ser analisadas por este material didático.

1. Classes abstratas e funções virtuais puras

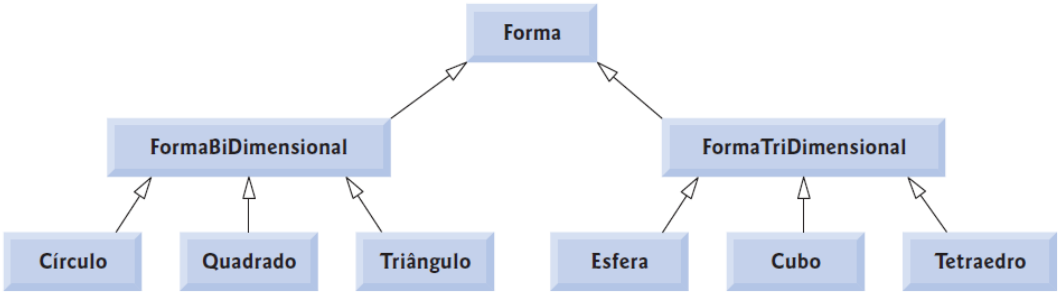
Se pensarmos em uma classe como sendo um tipo definido pelo usuário, é natural imaginar que os programas irão fazer uso desta classe, criando/instanciando objetos desse tipo. Porém, por mais estranho isso possa parecer, há casos em que será útil definir classes das quais o programador nunca irá instanciar qualquer objeto. Classes deste tipo são chamadas de classes abstratas ou classes básicas abstratas, pois normalmente são utilizadas como classes básicas em hierarquias de herança. Logo mais veremos que as classes básicas são incompletas, por esta razão essas classes

não podem ser utilizadas para a criação de objetos. Serão as classes derivadas que irão definir as partes ausentes das classes abstratas básicas.

Uma classe abstrata fornece uma classe básica apropriada para que outras classes possam herdar as suas características. As classes que podem ser utilizadas para instanciar objetos são chamadas de classes concretas, fornecendo implementação para cada um dos métodos por elas definidas. Como exemplo, poderíamos ter uma classe básica abstrata `FormaTridimensional` e derivar as classes concretas `Cubo`, `Esfera` e `Cilindro`. Podemos afirmar que as classes básicas abstratas são bastante genéricas, o que as torna inapropriadas para definir objetos reais. É necessário ser mais específico para se instanciar objetos.

Não é obrigatório que uma hierarquia de herança contenha uma classe abstrata, mas inúmeros sistemas orientados a objetos bem projetados são compostos por hierarquias de classe que possuem como base classes básicas abstratas. As classes abstratas podem constituir alguns níveis superiores da hierarquia de herança.

A figura a seguir apresenta uma hierarquia de formas, que começa com a classe básica abstrata `Forma`. No próximo nível encontram-se mais duas classes abstratas, chamadas de `FormaBidimensional` e `FormaTridimensional`. Somente no terceiro nível da hierarquia é que são definidas classes concretas para formas bidimensionais (`Circulo`, `Quadrado` e `Triangulo`) e para formas tridimensionais (`Esfera`, `Cubo` e `Tetraedro`).



Para que uma classe seja considerada abstrata, basta que uma ou mais de suas funções virtuais sejam declaradas como **puras**. Uma função virtual pura é identificada por conter o '`= 0`' ao final de sua declaração, como no trecho de código abaixo:

```
virtual void desenhar() const = 0;
```

O '`= 0`' é conhecido como um especificador puro. As funções virtuais puras não possuem implementação. Toda classe derivada de uma classe abstrata básica deve sobrescrever as funções virtuais puras da classe básica, fornecendo implementações concretas para essas funções. Vocês podem estar se perguntando neste momento: "Qual a diferença entre uma função virtual e uma função virtual pura?"

Uma função virtual possui uma implementação e dá a opção à classe derivada de sobrescrever ou não essa função. Em contrapartida, uma função virtual pura não fornece uma implementação

e exige que a classe derivada sobrescreva esta função para que a classe derivada seja concreta. Caso a função virtual pura não seja sobrescrita em uma classe derivada, então essa classe será também abstrata (`FormaBidimensional` e `FormaTridimensional`).

Algumas observações devem ser efetuadas em relação às classes abstratas. Uma classe abstrata define uma interface pública que é comum para as diversas classes presentes em uma hierarquia de classes. Uma classe abstrata deve conter uma ou mais funções virtuais puras que as classes derivadas concretas devem sobrescrever. Não é possível instanciar objetos de uma classe abstrata. A tentativa de efetuar tal operação ocasiona um erro de compilação. Uma classe abstrata também pode conter atributos e métodos concretos, incluindo construtores e destrutores, que estarão sujeitos às regras normais de herança.

Apesar de não ser possível criar objetos a partir de uma classe básica abstrata, ela pode ser utilizada para declarar ponteiros e referências que podem referenciar objetos de qualquer uma de suas classes derivadas. Desta maneira, os programas podem utilizar esses ponteiros e referências para trabalhar com objetos de classe derivada de forma polimórfica.

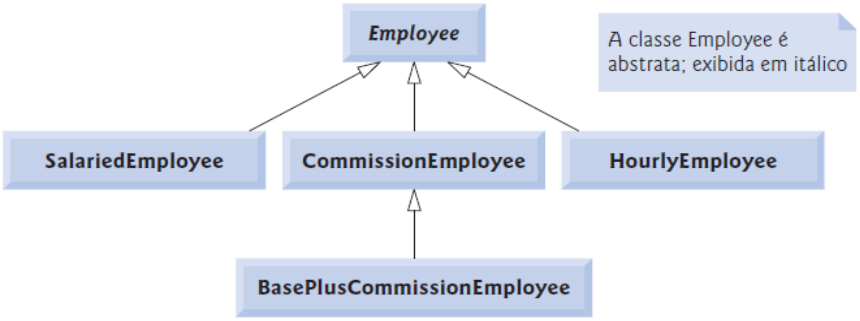
2. Estudo de caso: sistema de folha de pagamento

Nesta seção iremos remodelar a hierarquia de classe de empregados que estamos analisando desde que começamos a estudar herança de classes. Neste momento, iremos utilizar uma classe abstrata em conjunto com o conceito de polimorfismo para efetuar cálculos de folha de pagamento, tendo como base diferentes tipos de empregados. Suponha que seja necessário resolver o seguinte problema:

> *Os empregados de uma empresa são pagos semanalmente. Há na empresa quatro tipos de empregados: assalariados, que recebem salários fixos independentemente do número de horas trabalhadas, os que recebem por hora trabalhada, e recebem horas extras por todas as horas trabalhadas além das 40 horas, comissionados, que recebem uma porcentagem sobre suas vendas e os empregados assalariados-comissionados que recebem um salário base adicionado a um percentual sobre suas vendas. Para o período de pagamento atual, a empresa decidiu recompensar os empregados comissionados assalariados, fornecendo um acréscimo de 10% ao salário base.*

Iremos utilizar uma classe abstrata denominada `Employee` para representar o conceito geral de um empregado. Três classes irão derivar diretamente de `Employee`, são elas: `SalariedEmployee`, `CommissionEmployee` e `HourlyEmployee`. O último tipo de empregado é representado pela classe `BasePlusCommissionEmployee`, que é derivada da classe `CommissionEmployee`. Para que vocês compreendam melhor como a hierarquia de classe do nosso exemplo didático é formada, o diagrama de classe UML* a seguir ilustra a estrutura de herança de nosso programa de pagamento de empregados.

.



A classe básica abstrata `Employee` será responsável por declarar uma 'interface' para a toda a hierarquia. Ou seja, ela irá definir atributos e um conjunto de métodos que poderão ser utilizados por todos os objetos derivados de `Employee`. Por exemplo, cada empregado possui um nome, um sobrenome e um número de seguro social, independentemente da forma como seus vencimentos são calculados. Desta maneira, os atributos privados `firstName`, `lastName` e `socialSecurityNumber` são declarados na classe básica abstrata `Employee`.

2.1 Classe básica abstrata `Employee`

A classe `Employee` é implementada pelos arquivos `Employee.h` e `Employee.cpp`. Ela fornece as funções-membro `earnings` e `print`, além de inúmeros métodos `get` e `set` para manipular os atributos da classe `Employee`. Um método responsável por calcular os vencimentos de empregados possui um aspecto geral, mas os cálculos a serem realizados dependem do tipo/classe do empregado. Por esta razão, a função-membro `earnings` é definida como sendo uma função virtual pura na classe básica `Employee`, pois uma implementação-padrão não faz sentido para essa função. Ao fazermos isso, obriga-se que toda classe concreta derivada diretamente de `Employee` sobrescreva `earnings` com uma implementação apropriada.

A função `print` na classe `Employee` exibe o nome, o sobrenome e o número de seguro social do empregado. Mais a frente veremos que as classes derivadas de `Employee` irão sobrescrever a função `print` para apresentar informações específicas para cada tipo de empregado.

Vamos avaliar o arquivo de definição `Employee.h`. Na seção `public`, encontramos o construtor da classe (linha 12), um conjunto de funções `set` e `get` que configuram e retornam, respectivamente, o nome, sobrenome e o número de seguro social (linhas 14 a 21). Além disso, também encontramos a definição da função virtual pura `earnings` (linha 24) e a função virtual `print` (linha 25).

Lembre-se de que `earnings` foi declarada como uma função virtual pura porque primeiro devemos conhecer o tipo de `Employee` (empregado) para determinar quais os cálculos devem ser realizados para obter os `earnings` (vencimentos) apropriados. Declarar essa função como virtual pura indica que cada classe derivada concreta deve fornecer uma implementação de `earnings` e que um programa pode utilizar os ponteiros de classe básica `Employee` para efetuar a chamada da função `earnings` polimorficamente para qualquer tipo de `Employee`.

O arquivo `Employee.cpp` contém as implementações dos métodos da classe `Employee`. Entretanto, observem que nenhuma implementação é oferecida para a função virtual pura `earnings`. A função virtual `print` (linhas 54–58) possui uma implementação que será sobrescrita por cada uma das classes derivadas apresentadas em nosso estudo de caso. Entretanto, cada uma dessas funções irá fazer uso da versão `print` da classe abstrata para imprimir informações comuns a todas as classes na hierarquia `Employee`. Lembrem-se que uma função virtual `print` não precisa ser obrigatoriamente sobrescrita em classes derivadas. Caso isso não seja realizado, a classe derivada irá herdar a implementação de sua classe básica.

2.2 Classe derivada concreta `SalariedEmployee`

A classe `SalariedEmployee` é composta por seu arquivo de definição `SalariedEmployee.h` e pelo arquivo de implementação `SalariedEmployee.cpp`. Esta classe herda as características da classe `Employee` (linha 8). A seção `public` é composta pelo construtor (linhas 11–12), função `set` para atribuir um novo valor não negativo ao membro de dados `weeklySalary` (linha 14) e uma função `get` para retornar o valor de `weeklySalary` (linha 15). A classe ainda possui a função virtual `earnings` que calcula os rendimentos de um `SalariedEmployee` (linha 18) e a função virtual `print` que apresenta informações relacionadas com o tipo de empregado representado pela classe `SalariedEmployee` (linha 19).

O arquivo `SalariedEmployee.cpp` contém as implementações dos métodos da classe. O construtor da classe passa o nome, o sobrenome e o número de seguro social para o construtor da classe `Employee` (linha 11) a fim de inicializar os membros de dados `private` que são herdados da classe básica. Lembrem-se que os atributos privados não são acessíveis de forma direta na classe derivada.

A função `earnings` (linha 30–33) sobrescreve a função virtual pura `earnings` definida na classe básica `Employee`, fornecendo uma implementação concreta que retorna o salário semanal de um empregado do tipo `SalariedEmployee`. Se o método `earnings` não fosse implementado, a classe `SalariedEmployee` seria também uma classe abstrata e qualquer tentativa de instanciar um objeto desta classe resultaria em um erro de compilação.

A função `print` da classe `SalariedEmployee` (linhas 36–41) sobrescreve a função `print` da classe abstrata básica `Employee`. Se isso não fosse realizado, a classe `SalariedEmployee` herdaria a função `print` da classe `Employee`. Nesse caso, a função `print` de `SalariedEmployee` apresentaria apenas o nome completo e o número de seguro social do empregado. Para imprimir informações completas de um empregado do tipo `SalariedEmployee`, a função `print` da classe derivada exibe em tela as características gerais de todos os empregados, por meio da chamada da função-membro `print` da classe básica (linha 39). Além disso, também é apresentado o salário semanal do empregado, armazenado no atributo `weeklySalary`, que é obtido por meio da chamada do método `getWeeklySalary` da classe `SalariedEmployee`.

2.3 Classe derivada concreta `HourlyEmployee`

A classe `HourlyEmployee` é implementada pelos arquivos `HourlyEmployee.h` e `HourlyEmployee.cpp`. Em seu arquivo de definição, pode-se notar que essa classe herda características da classe básica `Employee` (linha 8). A sua seção `public` é composta pelo construtor da classe (linhas 11–12), as funções `set` e `get` utilizadas para atribuir e retornar os valores dos atributos `wage` e `hours`, respectivamente (linhas 14 e 17). Além disso, também podemos encontrar a função virtual `earnings` que calcula os rendimentos de um empregado `HourlyEmployee` (linha 21) e a função virtual `print` que exibe informações específicas do empregado (linha 22).

No arquivo `HourlyEmployee.cpp`, encontram-se as implementações dos métodos da classe `HourlyEmployee`. As linhas 18–21 e 30–34 definem as funções `set` que atribuem valores aos atributos `wage` e `hours`, respectivamente. A função `setWage` (linhas 18–21) assegura que `wage` é não negativo, e a função `setHours` (linhas 30–34) assegura que o membro de dados `hours` está entre 0 e 168, que é o número total de horas em uma semana. As funções `get` da classe `HourlyEmployee` são implementadas nas linhas 24–27 e 37–40.

Observem que o construtor da classe `HourlyEmployee`, faz uso do construtor da classe abstrata básica `Employee` (linha 11) para inicializar os atributos privados herdados pela classe `HourlyEmployee`. A implementação da função `print` efetua a chamada do método `print` da classe básica (linha 56) para imprimir em tela informações comuns a todos os tipos de empregados (nome, sobrenome e número de seguro social).

Nas linhas 44–50, a função pura virtual `earnings` é sobrescrita pela classe `HourlyEmployee`. Este método calcula os vencimentos deste tipo de empregado, fazendo com que a classe se torne concreta. O empregado que ganha por hora trabalhada terá a sua quantidade de horas (`hours`) de serviço multiplicada pelo valor de sua hora de trabalho (`wage`). Caso a quantidade de horas trabalhadas ultrapasse as 40 horas, o empregado receberá um adicional por cada hora extra trabalhada.

2.4 Classe derivada concreta `CommissionEmployee`

Vocês já devem estar bastante familiarizados com a classe `CommissionEmployee`, pois ela tem servido de exemplo para outros momentos da disciplina. Neste exemplo em específico, a classe `CommissionEmployee` herda características da classe abstrata básica `Employee` (linha 8 do arquivo `CommissionEmployee.h`).

No arquivo `CommissionEmployee.cpp`, encontram-se as implementações dos métodos da classe. O construtor (linhas 9–15) faz uso do construtor da classe básica `Employee` para atribuir valores aos atributos presentes em todos os empregados da hierarquia de classe e dos métodos `setGrossSales` e `setCommissionRate` para configurar as vendas brutas (`grossSales`) e a taxa de comissão (`commissionRate`). Os métodos `set` (linhas 18–21 e 30–33), para atribuir

novos valores aos atributos `commissionRate` e `grossSales`, respectivamente. As funções `get` (linhas 24–27 e 36–39) retornam os valores desses mesmos atributos.

O método `earnings` (linhas 43–46) sobrescreve a função virtual pura definida na classe básica `Employee`. Ela é utilizada para calcular os rendimentos de um empregado do tipo `CommissionEmployee`. A função virtual `print` da classe básica `Employee` é sobrescrita nas linhas 49–55. Observem que em sua implementação é efetuada a chamada do método `print` de classe `Employee` para exibir informações básicas de todos os empregados (nome, sobrenome e o número de seguro social). Além disso, são apresentadas informações específicas, como o valor dos atributos `grossSales` e `commissionRate`.

2.5 Classe derivada concreta indireta `BasePlusCommissionEmployee`

Diferente das outras classes, a classe `BasePlusCommissionEmployee` herda diretamente da classe `CommissionEmployee` (linha 8 do arquivo `BasePlusCommissionEmployee.h`). Isso implica que, de forma, indireta, esta `BasePlusCommissionEmployee` é uma classe derivada da classe `Employee`.

O arquivo `BasePlusCommissionEmployee` inclui a implementação do construtor da classe (linhas 10–16). Este construtor realiza a chamada do construtor de sua classe básica direta `CommissionEmployee` (linha 13) para inicializar os atributos herdados e faz uso do método `setBaseSalary` para configurar o salário base do empregado. A classe também contém `BasePlusCommissionEmployee` também contém uma função `set` (linhas 19–22) para atribuir um novo valor ao membro de dados `baseSalary` e uma função `get` (linhas 25–28) para retornar o valor `baseSalary`.

A função virtual `earnings` é sobrescrita nas linhas de código 32–35 para calcular os rendimentos dos empregados do tipo `BasePlusCommissionEmployee`. Na linha 34, podemos notar que a função sobrescrita efetua a chamada da função `earnings` da classe básica `CommissionEmployee` para calcular a parte baseada na comissão dos rendimentos do empregado. É importante comentar que a classe `CommissionEmployee` é uma classe básica para a classe `BasePlusCommissionEmployee` e, ao mesmo tempo, é uma classe derivada da classe `Employee`.

A função `print` de `BasePlusCommissionEmployee` (linhas 38–43) desencadeia uma sequência de chamadas de funções que se distribuem pelos três níveis da hierarquia de classes. Esta função-membro realiza a chamada do método `print` de sua classe básica `CommissionEmployee` (linha 41), que, por sua vez, efetua a chamada da função virtual da classe abstrata básica `Employee` (linha 52 do arquivo `Employee.cpp`). Estas duas funções são chamadas para exibir os valores dos atributos herdados das classes básicas direta e indireta. Além dessas informações, o método `print` da classe `BasePlusCommissionEmployee` exibe em tela o valor do salário base do empregado, característica específica os objetos desta classe.

3. Arquivo de exemplo de utilização

O arquivo `mainPolimorfismo.cpp` apresenta um programa de exemplo que faz uso de nossa hierarquia de classe. Inicialmente, nas linhas 31–38, são criados objetos para cada uma das classes concretas que representam os tipos de empregados do problema: `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` e `BasePlusCommissionEmployee`.

As linhas 43–51 exibem em tela informações e os rendimentos de cada empregado. Cada chamada de método é um exemplo de vinculação estática, pois em tempo de compilação, o compilador é capaz de identificar o tipo de cada objeto para definir quais métodos `print` e `earnings` serão chamados. Isso é possível a partir da utilização de `handles` de nome (os nomes dos próprios objetos instanciados).

A linha 54 cria um `vector` de quatro ponteiros da classe abstrata `Employee`. Cada uma das posições do vetor aponta para um dos objetos anteriormente criados das classes concretas (linhas 57–60). Essas atribuições somente são possíveis porque os objetos das classes derivadas também são da classe básica `Employee`, mesmo esta última classe sendo uma classe abstrata. Portanto, é possível atribuir os endereços dos objetos das classes derivadas aos ponteiros da classe básica `Employee`.

A estrutura de repetição `for` é utilizada nas linhas 68–69 para percorrer o `vector employees`, chamando a função `virtualViaPointer` (linhas 83–87) para cada elemento em armazenado em `employees`. Esta função recebe no parâmetro `baseClassPtr` o endereço armazenado em um elemento de `employees`. Na implementação desta função, são feitas chamadas para as funções virtuais `print` (linha 85) e `earnings` (linha 86).

Notem que a função `virtualViaPointer` não contém informações de qual tipo é o objeto que está sendo apontado. A função conhece apenas o tipo de classe básica `Employee`. Desta forma, em tempo de compilação, o compilador não tem como saber que funções de classe concreta devem ser chamadas a partir do parâmetro `baseClassPtr`. Entretanto, isto pode ser definido em tempo de execução, cada chamada de função virtual irá ocasionar a execução da função adequada para o objeto para o qual o ponteiro `baseClassPtr` aponta no momento. Assim, todas as chamadas de função virtual para `print` e `earnings` são convertidas em tempo de execução com vinculação dinâmica. Por exemplo, se o ponteiro `baseClassPtr` apontar para um objeto da classe `SalariedEmployee`, serão os métodos `print` e `earnings` sobrescritos por essa classe que serão chamados na função `virtualViaPointer`.

Nas linhas 75–76 outra estrutura de repetição `for` efetua a chamada da função `virtualViaReference` (linhas 91–95) para cada elemento do `vector employees`. A função `virtualViaReference` recebe em seu parâmetro `baseClassRef` uma referência formada desreferenciando o ponteiro armazenado em cada elemento de `employees` (linha 76). Cada chamada para `virtualViaReference` chama as funções virtual `print` (linha 93) e `earnings` (linha 94) por meio da referência `baseClassRef` para demonstrar que o comportamento polimórfico também ocorre com referências de classe básica, em mais um exemplo de vinculação dinâmica.

REFERÊNCIAS

DEITEL, H. M.; DEITEL, P. J. **C++: como programar**. 5 ed. São Paulo: Pearson Prentice Hall, 2006.