Herança

BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

Departamento de Computação - UFOP
Baseado nos slides do Prof. Marco Antônio Carvalho

Introdução I

- A herança é uma forma de **reuso de software**
  - O programador cria uma classe que **absorve os dados e o comportamento** de uma classe existente;
  - Ainda é possível **aprimorá-la com novas capacidades**.

Introdução II

- Além de **reduzir o tempo de desenvolvimento**, o reuso de software **aumenta** a probabilidade de **eficiência** de um software
- Componentes já debugados e de qualidade provada contribuem para isto.

Introdução III

- Uma **classe existente** e que será absorvida é chamada de **classe base** (ou **superclasse**);
- A **nova classe**, que absorve, é chamada de **classe derivada** (ou **subclasse**)
  - Considerada uma **versão especializada** da classe base.

Introdução IV

- É importante **identificar a diferença** entre **composição** e **herança**:
  - Na **herança**, um objeto da subclasse **"é um"** objeto da superclasse. Por exemplo, o carro é um veículo;
  - Enquanto que na **composição** um objeto **"tem um"** outro objeto. Por exemplo, o carro tem uma direção.

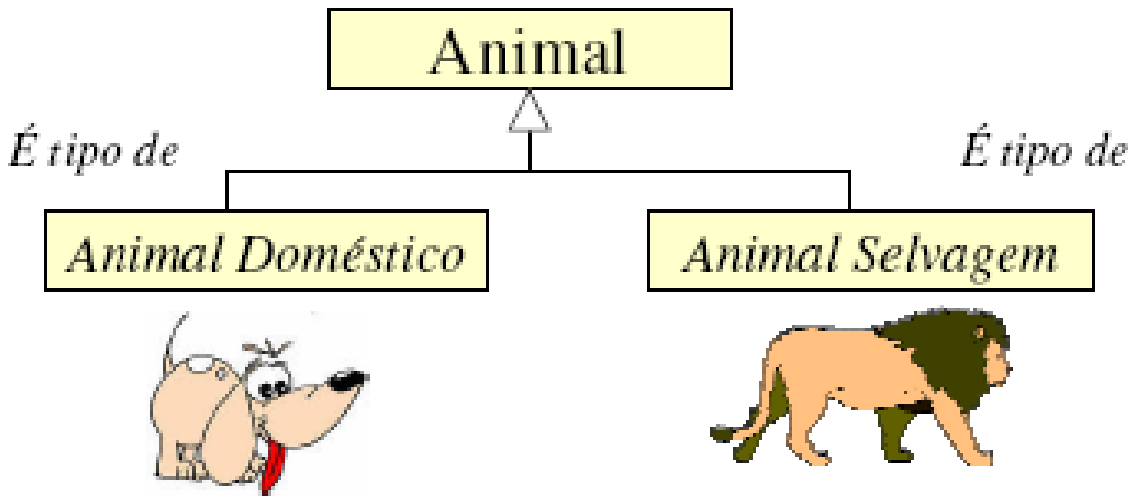
Herança I

- **Herança** é o **mecanismo** pelo qual **elementos mais específicos incorporam** a **estrutura e comportamento** de elementos **mais gerais**.
- Uma **classe derivada herda** a estrutura de **atributos e métodos** de sua **classe "base"**, mas pode seletivamente:
  - **adicionar** novos **métodos**
  - **estender** a estrutura de **dados**
  - **redefinir a implementação de métodos** já existentes

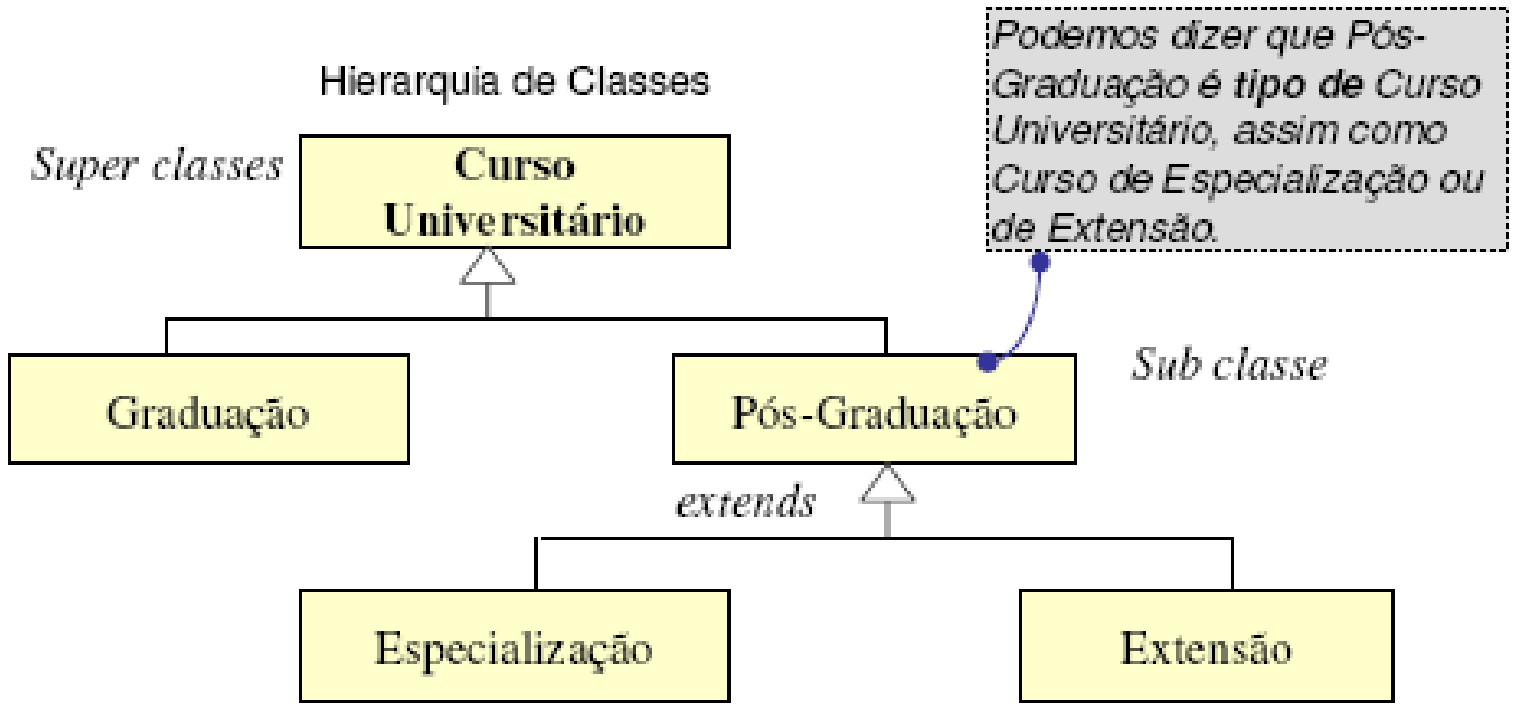
Herança II

- Uma classe "pai" ou super classe proporciona a **funcionalidade que é comum a todas** as suas classes derivadas, filhas ou sub classe,

- Uma classe derivada proporciona a **funcionalidade adicional que especializa seu comportamento**.



Herança III


Hierarquia de Classes
Super classes
Curso Universitário
Podemos dizer que Pós-Graduação é tipo de Curso Universitário, assim como Curso de Especialização ou de Extensão.
Graduação
Pós-Graduação
Sub classe
extends
Especialização
Extensão

Herança IV

- Um **possível problema** com herança é ter que **herdar atributos ou métodos desnecessários** ou inapropriados
- É **responsabilidade do projetista** determinar se as **características da classe base** são **apropriadas** para herança direta e também para futuras classes derivadas.

Herança V

- Ainda, **é possível** que **métodos necessários não se comportem** de maneira especificamente necessária
- Nestes casos, é possível que a **classe derivada redefina** o método para que este tenha uma implementação específica.
- Um **objeto** de uma **classe derivada** pode ser **tratado** como um objeto da **classe base**.

Herança VI

- Os métodos de uma classe derivada podem necessitar acesso aos métodos e atributos da classe base
  - **Somente** os **membros não privados** estão **disponíveis**;
  - **Membros** que **não** devem ser **acessíveis** através de herança devem ser **privados**;
  - Atributos privados poderão ser acessíveis por **getters** e **setters** públicos

Exemplo 1 I

- Hierarquia de classes para descrever a comunidade acadêmica: classe *CommunityMember*;
- Primeiras formas de especialização serão
  - Empregados (*Employee*);
  - Estudantes (*Students*);
  - Ex-Alunos (*Alumnus*);

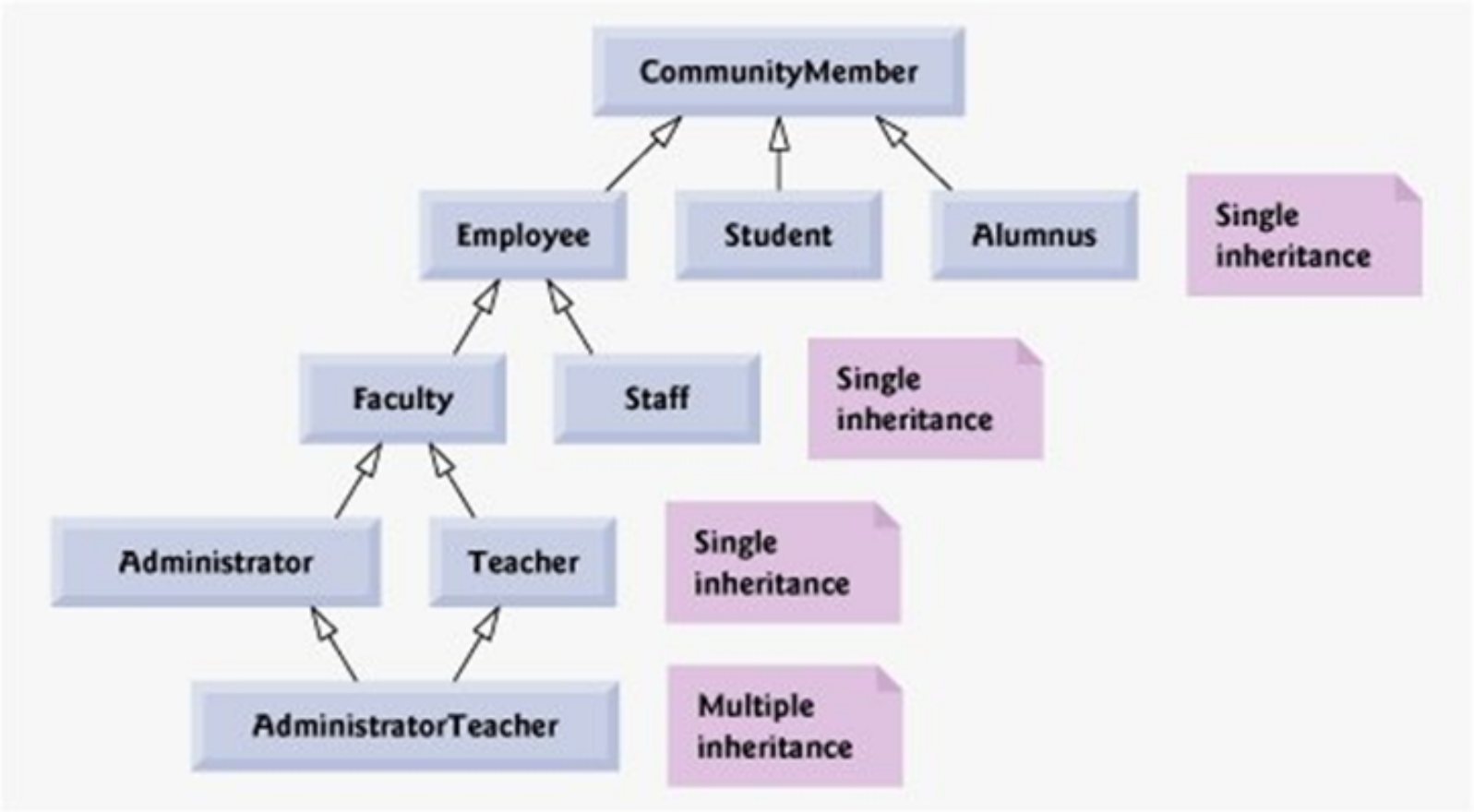
Exemplo 1 II

- Entre os empregados, temos:
  - Corpo docente (*Faculty*);
  - Funcionários de apoio (*Staff*).
- Entre o corpo docente (*Faculty*) temos:
  - Professores (*Teacher*)
  - Administradores (*Administrator*)

Exemplo 1 III

- Cada seta na hierarquia apresentada a seguir representa um relacionamento "**é um**";
- *CommunityMember* é a base direta ou indireta de todas as classes da hierarquia.

Exemplo 1 IV


CommunityMember
Employee
Student
Alumnus
Single inheritance
Faculty
Staff
Single inheritance
Administrator
Teacher
Single inheritance
AdministratorTeacher
Multiple inheritance

Exemplo 1 V

Sintaxe em C++

```
class classe-derivada : acesso classe-base
{
    corpo da nova clase
}
```

O operador acesso é opcional, mas claro se presente tem de ser *public*, *private* ou *protected*.

Exemplo 1 VI

Se o acesso for *public*

- membros *public* da classe base: é como fazer copias dos métodos e colocássemos como *public* na classe derivada
- membros *private*: não são passados, acessível apenas pelo métodos da classe base e funções amigas
- membros *protected*: são "copiados" como *protected* na classe derivada

Funções amigas (*friend*) não são herdadas.

Exemplo Funcionários I

- São considerados dois tipos de funcionários:
  - *Funcionários Comissionados* são pagos com comissões sobre vendas;
  - *Funcionários Comissionados com salário base* são pagos com um salário fixo e também recebem comissões sobre vendas.

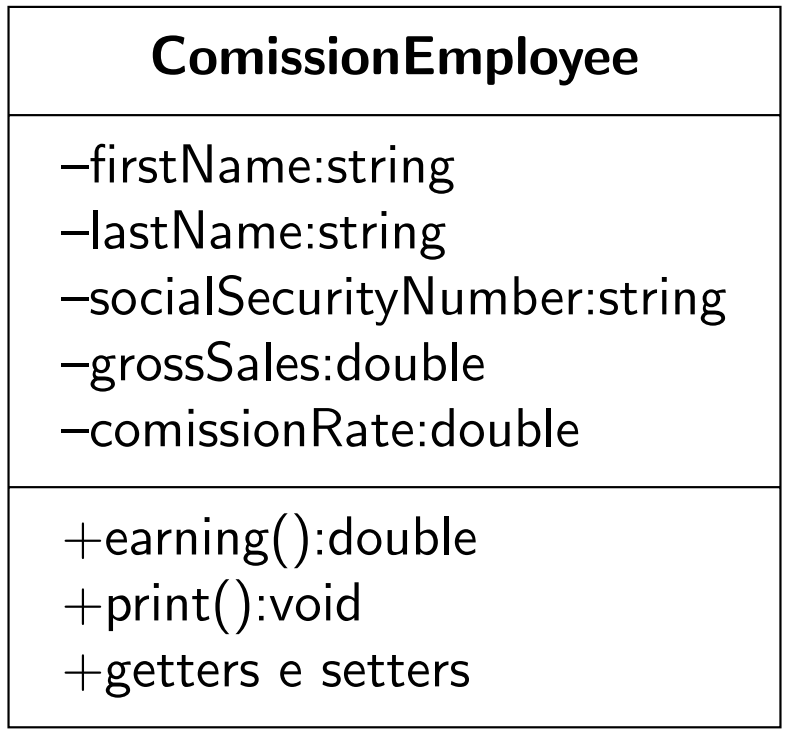
Exemplo Funcionários II

- A partir deste contexto, serão apresentados 4 exemplos:
  1. Classe *ComissionEmployee*, que representa funcionários comissionados
     - Membros Privados
  2. Classe *BasePlusCommissionEmployee*, que representa funcionários comissionados com salário base.
     - Sem herança.

Exemplo Funcionários III

3. Nova classe *BasePlusCommissionEmployee*
   - Usando herança;
   - Classe *ComissionEmployee* com membros *protected*.
4. Nova herança, porém, classe *ComissionEmployee* com membros *private*.

ComissionEmployee I



ComissionEmployee.h I

```
#ifndef COMISSION_H
#define COMISSION_H
#include <string>
using std::string;
class ComissionEmployee
{
    string firstName;
    string lastName;
    string socialSecurityNumber;
    double grossSales;
    double comissionRate;
public:
    ComissionEmployee(const string&, const string&,
            const string&, double=0.0, double=0.0);
    ~ComissionEmployee();
    void setFirstName(const string&);
    string getFirstName() const;
    ...
```

ComissionEmployee.h II

```
    . . .
    void setLastName(const string&);
    string getLastName() const;
    void setSocialSecurityNumber(const string&);
    string getSocialSecurityNumber() const;
    void setGrossSales(double);
    double getGrossSales() const;
    void setComissionRate(double);
    double getComissionRate() const;
    double earnings() const;
    void print() const;

};
#endif
```

ComissionEmployee.h III

- Os atributos são privados
  - Não acessíveis por classes derivadas.
- Os *getters* e *setters* desta classe são públicos
  - Acessíveis por classes derivadas;
  - Podem ser utilizados como meio para acessar os atributos
- Além disto, *getters* são declarados como *const*
  - Não podem alterar o valor de nenhum argumento.

ComissionEmployee.cpp I

```
#include "ComissionEmployee.h"

ComissionEmployee::ComissionEmployee(const string& first,
    const string& last, const string& ssn,
    double sales, double rate){
    firstName = first;
    lastName = last;
    socialSecurityNumber = ssn;
    setGrossSales(sales);
    setComissionRate(rate);
}
ComissionEmployee::~ComissionEmployee(){}

void ComissionEmployee::setFirstName(const string& first){
    firstName = first;
}

string ComissionEmployee::getFirstName() const{
    return firstName;
}
```

ComissionEmployee.cpp II

- Note que o construtor da classe base acessa e realiza atribuições aos atributos diretamente
- Os atributos *sales* e *rate* são acessados através de métodos porque estes realizam testes de consistência nos valores passados por parâmetros;
- Os demais atributos também poderiam ser checados quanto a sua consistência

ComissionEmployee.cpp III

```
void ComissionEmployee::setLastName(const string& last){
    lastName = last;
}

string ComissionEmployee::getLastName() const{
    return lastName;
}

void ComissionEmployee::setSocialSecurityNumber(
    const string& ssn){
    socialSecurityNumber = ssn;
}

string ComissionEmployee::getSocialSecurityNumber() const{
    return socialSecurityNumber;
}
```

ComissionEmployee.cpp IV

```
void ComissionEmployee::setGrossSales(double sales){
    grossSales = (sales < 0) ? 0 : sales;
}

double ComissionEmployee::getGrossSales() const{
    return grossSales;
}

void ComissionEmployee::setComissionRate(double rate){
    comissionRate = (rate >= 0.0 && rate <= 1.0) ?
        rate : 0 ;
}

double ComissionEmployee::getComissionRate() const{
    return comissionRate;
}

double ComissionEmployee::earnings() const{
    return comissionRate * grossSales;
}
```

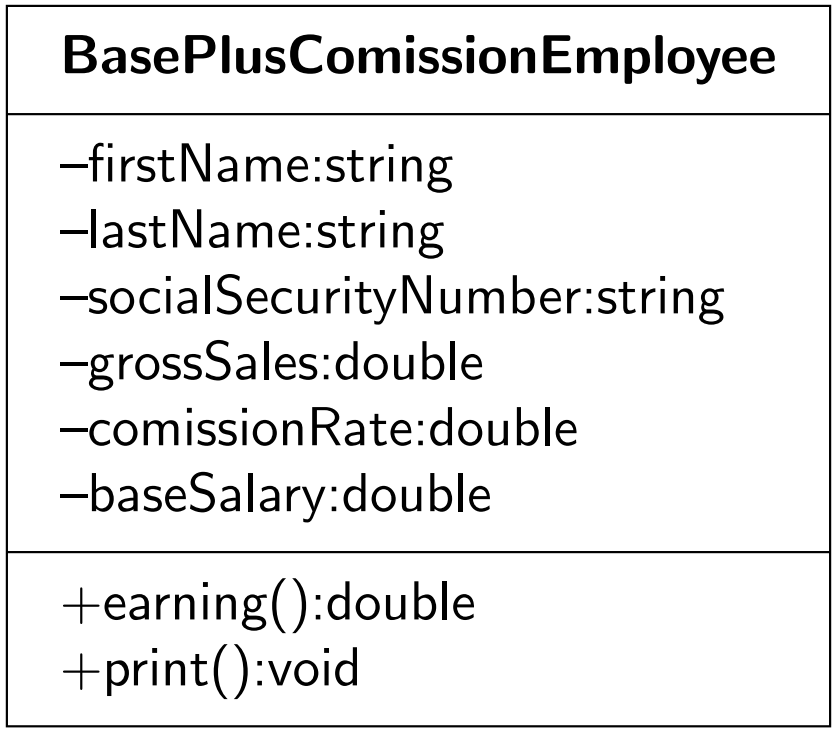
ComissionEmployee.cpp V

```
void ComissionEmployee::print() const
{
    cout<< "\n comission employee: " << firstName
        << " " << lastName
        << "\n social security number: "
        << socialSecurityNumber
        << "\n gross sales: " << grossSales
        << "\n comission rate: " << comissionRate;
}
```

Main.cpp I

```
#include <iostream>
#include <string>
#include <iomanip>
#include "ComissionEmployee.h"
using namespace std;

int main()
{
    ComissionEmployee employee("Sue", "Jones",
    "222-22-2222", 10000.0, 0.1);
    cout << fixed << setprecision(2);
    employee.print();
    cout << "\n Employee earning: "
         << employee.earnings();

    return 0;
}
```

BasePlusComissionEmployee I



BasePlusComissionEmployee II

- Vamos agora criar a classe *BasePlusCommissionEmployee*, que representa funcionários comissionados com salario base
- Embora seja uma especialização da classe anterior, vamos definir completamente a classe.

BasePlusComissionEmployee.h I

```
#ifndef BASEPLUS_H
#define BASEPLUS_H
#include <string>
using std::string;
class BasePlusComissionEmployee
{
    string firstName;
    string lastName;
    string socialSecurityNumber;
    double grossSales;
    double comissionRate;
    double baseSalary;
public:
    BasePlusComissionEmployee(const string&,
        const string&, const string&, double=0.0,
        double=0.0, double=0.0);
    ~BasePlusComissionEmployee();
    void setFirstName(const string&);
    string getFirstName() const;
    . . .
```

BasePlusComissionEmployee.h II

```
    . . .
    void setLastName(const string&);
    string getLastName() const;
    void setSocialSecurityNumber(const string&);
    string getSocialSecurityNumber() const;
    void setGrossSales(double);
    double getGrossSales() const;
    void setComissionRate(double);
    double getComissionRate() const;
    void setBaseSalary(double);
    double getBaseSalary() const;
    double earnings() const;
    void print() const;
};
#endif
```

BasePlusComissionEmployee.h III

- Foram adicionados um atributo com respectivo *getter* e *setter*, além de um parâmetro adicional no construtor;
- O restante do código é basicamente redundante em relação à classe *ComissionEmployee*.

BasePlusComissionEmployee.cpp I

```
#include "BasePlusComissionEmployee.h"

BasePlusComissionEmployee::BasePlusComissionEmployee(
    const string& first, const string& last,
    const string& ssn, double sales,
    double rate, double salary){
    firstName = first;
    lastName = last;
    socialSecurityNumber = ssn;
    setGrossSales(sales);
    setComissionRate(rate);
    setBaseSalary(salary);
}

BasePlusComissionEmployee::~BasePlusComissionEmployee() {}
```

BasePlusComissionEmployee.cpp II

```
void BasePlusComissionEmployee::setBaseSalary(
    double salary){
    baseSalary = salary < 0 ? 0 : salary;
}

double BasePlusComissionEmployee::getBaseSalary() const{
    return baseSalary;
}

double BasePlusComissionEmployee::earnings() const{
    return baseSalary + (grossSales * comissionRate)
```

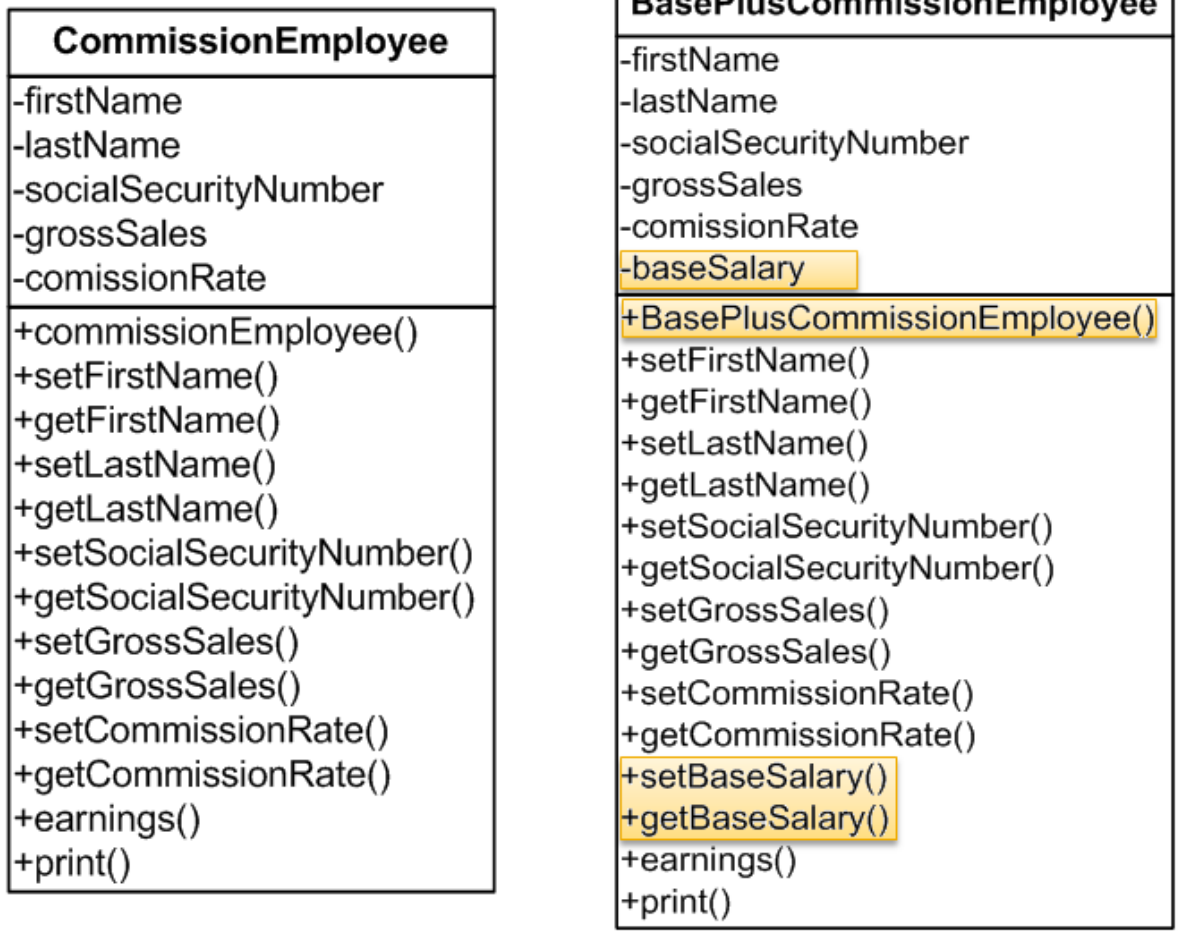
BasePlusComissionEmployee.cpp III

```
void BasePlusComissionEmployee::print() const{
    cout << "\n base-salaried comission employee: "
        << firstName << " " << lastName
        << "\n social security number: "
        << socialSecurityNumber
        << "\n gross sales: " << grossSales
        << "\n comission rate: " << comissionRate;
        << "\n base salary: " << baseSalary;
}
```

Main.cpp I

```
#include <iostream>
#include <string>
#include <iomanip>
#include "BasePlusComissionEmployee.h"
using namespace std;

int main()
{
        BasePlusComissionEmployee employee("Bob",
        "Lewis", "333-33-3333", 5000.0,
        0.04, 300.0);
        cout << fixed << setprecision(2);
        employee.print();
        cout << "\n Employee earning:
             << employee.earnings();
    return 0;
}
```

Comparação I



Herança I

- Ambas as classes compartilham a maior parte dos atributos (privados);
- *getters* e *setters* também são compartilhados
- Foi acrescentado apenas um getter e um setter devido ao novo atributo.
- Literalmente o código original foi copiado e adaptado.
- Quando a **redundância** entre **classes** acontece, **caracteriza-se** a necessidade de **herança**.

Herança II

- A replicação de código pode resultar em replicação de erros:
- A **manutenção é dificultada**, pois cada cópia tem que ser corrigida;
- Define-se uma classe que absorverá os atributos e métodos redundantes
  - A classe base;
  - A manutenção dada na classe base se reflete nas classes derivadas automaticamente.

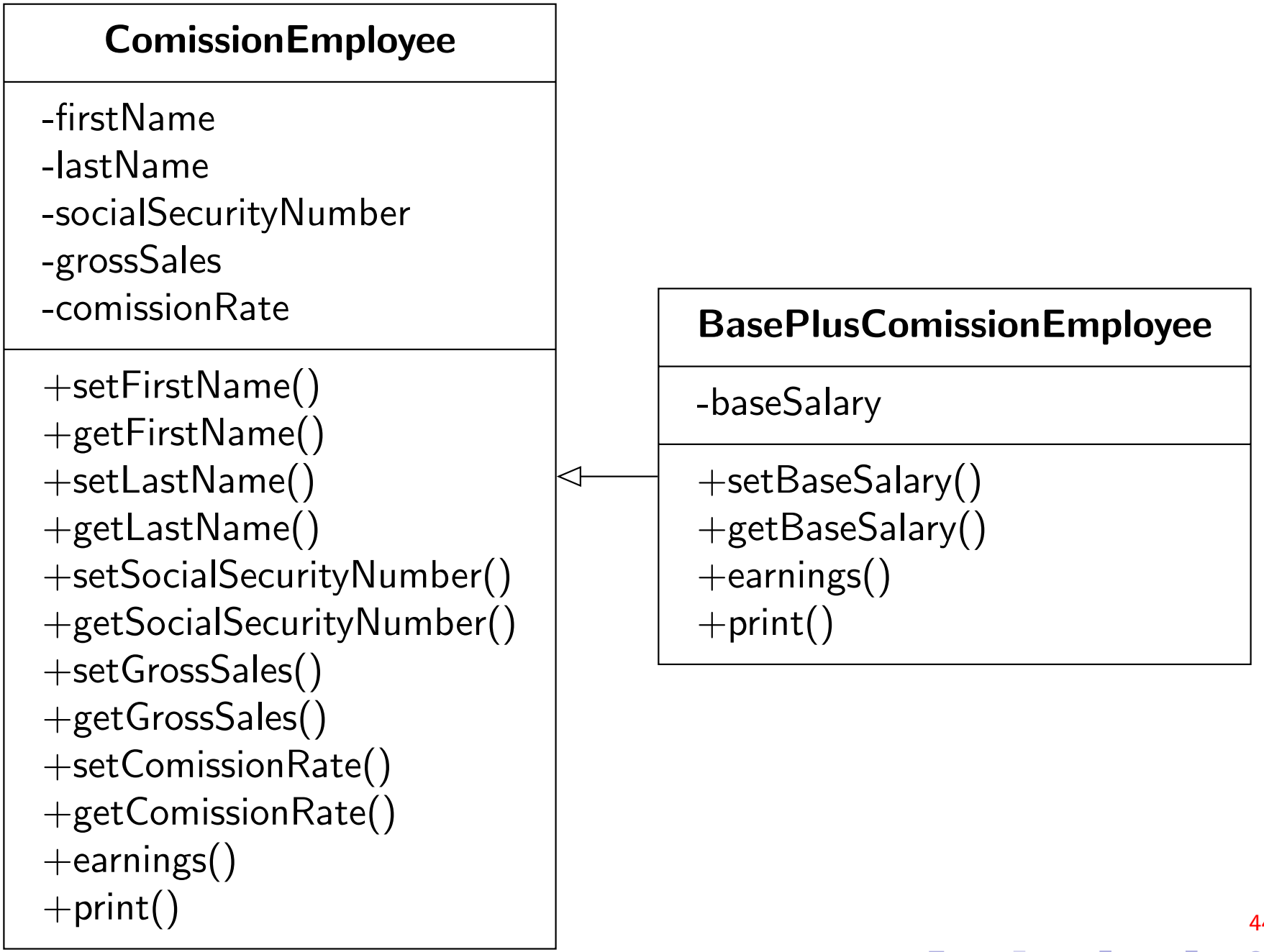
Herança III

- O próximo exemplo fixa a classe *ComissionEmployee* como classe base
- Será utilizada a mesma definição desta classe do exemplo anterior;
- Os atributos continuam privados.
- A classe *BasePlusCommissionEmployee* será a classe derivada

Herança IV

- Acrescenta-se o atributo *baseSalary*;
- Acrescenta-se também *getter* e *setter* e redefinirá dois métodos.
- Qualquer tentativa de acesso aos membros privados da classe base gerará erro de compilação.

Herança V


ComissionEmployee
-firstName
-lastName
-socialSecurityNumber
-grossSales
-comissionRate
+setFirstName()
+getFirstName()
+setLastName()
+getLastName()
+setSocialSecurityNumber()
+getSocialSecurityNumber()
+setGrossSales()
+getGrossSales()
+setComissionRate()
+getComissionRate()
+earnings()
+print()
BasePlusComissionEmployee
-baseSalary
+setBaseSalary()
+getBaseSalary()
+earnings()
+print()

BasePlusComissionEmployee.h I

```
#ifndef BASEPLUS_H
#define BASEPLUS_H
#include <string>
#include "ComissionEmployee.h"
using std::string;
class BasePlusComissionEmployee :
    public ComissionEmployee{
    double baseSalary;
public:
    BasePlusComissionEmployee(const string&,
        const string&, const string&, double=0.0,
        double=0.0, double=0.0);
    ~BasePlusComissionEmployee();
    void setBaseSalary(double);
    double getBaseSalary() const;
    double earnings() const;
    void print() const;
};
#endif
```

BasePlusComissionEmployee.h II

- O operador **:** define a herança;
- A herança é pública
- Todos os métodos públicos da classe base são também métodos públicos da classe derivada
  - Embora não os vejamos na definição da classe derivada, eles fazem parte dela.
- Foi definido um construtor específico.
- É necessário incluir o *header* da classe a ser herdada

BasePlusComissionEmployee.cpp I

```
#include "BasePlusComissionEmployee.h"

BasePlusComissionEmployee::BasePlusComissionEmployee(
    const string& first, const string& last,
    const string& ssn, double sales,
    double rate, double salary) :
    ComissionEmployee(first, last,
        ssn, sales, rate){
    setBaseSalary(salary);
}

BasePlusComissionEmployee::~BasePlusComissionEmployee() {}
```

BasePlusComissionEmployee.cpp II

```
void BasePlusComissionEmployee::setBaseSalary(
    double salary){
    baseSalary = salary < 0 ? 0 : salary;
}

double BasePlusComissionEmployee::getBaseSalary() const{
    return baseSalary;
}
//classe derivada não pode acessar dados
//private da classe base
double BasePlusComissionEmployee::earnings() const{
    return baseSalary + (grossSales * comissionRate)
```

BasePlusComissionEmployee.cpp III

```
//classe derivada não pode acessar dados
//private da classe base
void BasePlusComissionEmployee::print() const{
    cout << "\n base-salaried comission employee: "
        << firstName << " " << lastName
        << "\n social security number: "
        << socialSecurityNumber
        << "\n gross sales: " << grossSales
        << "\n comission rate: " << comissionRate;
        << "\n base salary: " << baseSalary;
}
```

BasePlusComissionEmployee.cpp IV

- O construtor da classe derivada chama explicitamente o construtor da classe base
- É necessário que a classe derivada tenha um construtor para que o construtor da classe base seja chamado;
- Se o construtor da classe base não for chamado explicitamente, o compilador chamará implicitamente o construtor *default* (sem argumentos) da classe base
- Se este não existir, ocorrerá um erro de compilação.

BasePlusComissionEmployee.cpp V

- Os métodos *earnings* e *print* deste exemplo gerarão erros de compilação
- Ambos tentam acessar diretamente membros privados da classe base, o que não é permitido
- Utilizar os *getters* associados a tais atributos para evitar os erros de compilação
  - *getters* são públicos
- Os métodos podem ser redefinidos sem causar erros de compilação, como veremos em breve

Herença I

- Vamos modificar o primeiro exemplo (classe *ComissionEmployee*) para que seus atributos sejam **protected**
- O modificador de acesso *protected* permite que um membro seja acessível por membros e funções amigas da classe base e derivada.
- Desta forma o segundo exemplo (classe *BasePlusComissionEmployee*) compilará sem erros

ComissionEmployee.h I

```
#ifndef COMISSION_H
#define COMISSION_H
#include <string>
using std::string;
class ComissionEmployee
{
protected:
    string firstName;
    string lastName;
    string socialSecurityNumber;
    double grossSales;
    double comissionRate;
public:
    ComissionEmployee(const string&, const string&,
                      const string&, double=0.0, double=0.0);
    ~ComissionEmployee();
    . . .
};
#endif
```

ComissionEmployee.h II

- A implementação da classe *ComissionEmployee* não muda em rigorosamente nada
- Internamente à classe base, nada muda se um membro é privado ou protegido.
- A implementação da classe *BasePlusComissionEmployee* também não muda
  - Na verdade, apenas herdando da nova classe base é que ela funciona!

ComissionEmployee.h III

- Utilizar atributos protegidos nos dá uma leve melhoria de desempenho
- Não é necessário ficar chamando funções (*getters* e *setters*).
- Por outro lado, também nos **gera dois problemas** essenciais:
  - Uma vez que não se usa getter e setter, pode haver inconsistência nos valores do atributo
  - Nada garante que o programador que herdar fará uma validação.
- Uma classe derivada deve depender do serviço prestado, e não da implementação da classe base.

ComissionEmployee.h IV

- Quando devemos usar *protected* então?
  - Quando a classe base precisa fornecer um serviço (método) apenas para classes derivadas e funções amigas, ninguém mais.
- **Declarar membros** como **privados** permite que a **implementação** da **classe base** seja **alterada sem implicar em alteração** da implementação da **classe derivada**;
- O **padrão mais utilizado** é **atributos privados** e *getters* e *setters* **públicos**.

ComissionEmployee.h V

- Vamos adequar nossas classes dos exemplos anteriores ao padrão proposto
  - De acordo com as boas práticas de engenharia de software.
- A classe base volta a ter atributos privados
- A implementação também passa a utilizar *getters* e *setters* internamente.
- A classe derivada não acessa os atributos diretamente

ComissionEmployee.h I

```
#ifndef COMISSION_H
#define COMISSION_H
#include <string>
using std::string;
class ComissionEmployee
{
private:
    string firstName;
    string lastName;
    string socialSecurityNumber;
    double grossSales;
    double comissionRate;
public:
    ComissionEmployee(const string&, const string&,
            const string&, double=0.0, double=0.0);
    ~ComissionEmployee();
    void setFirstName(const string&);
    string getFirstName() const;
    ...
```

ComissionEmployee.cpp I

```
#include "ComissionEmployee.h"

ComissionEmployee::ComissionEmployee(const string& first,
    const string& last, const string& ssn,
    double sales, double rate) :
    firstName(first), lastName(last),
    socialSecurityNumber(ssn){
    setGrossSales(sales);
    setComissionRate(rate);
}

double ComissionEmployee::earnings() const{
    return getcomissionRate() * getGrossSales();
}
```

ComissionEmployee.cpp II

```
void ComissionEmployee::print() const
{
    cout<< "\n comission employee: " << getFirstName()
        << " " << getLastName()
        << "\n social security number: "
        << getSocialSecurityNumber()
        << "\n gross sales: " << getGrossSales()
        << "\n comission rate: " << getComissionRate();
}
```

ComissionEmployee.cpp III

- O construtor inicializa os atributos usando uma lista de inicialização
- Executado antes do próprio construtor;
- Em alguns casos, é obrigatório: constantes e referências.

BasePlusComissionEmployee.cpp I

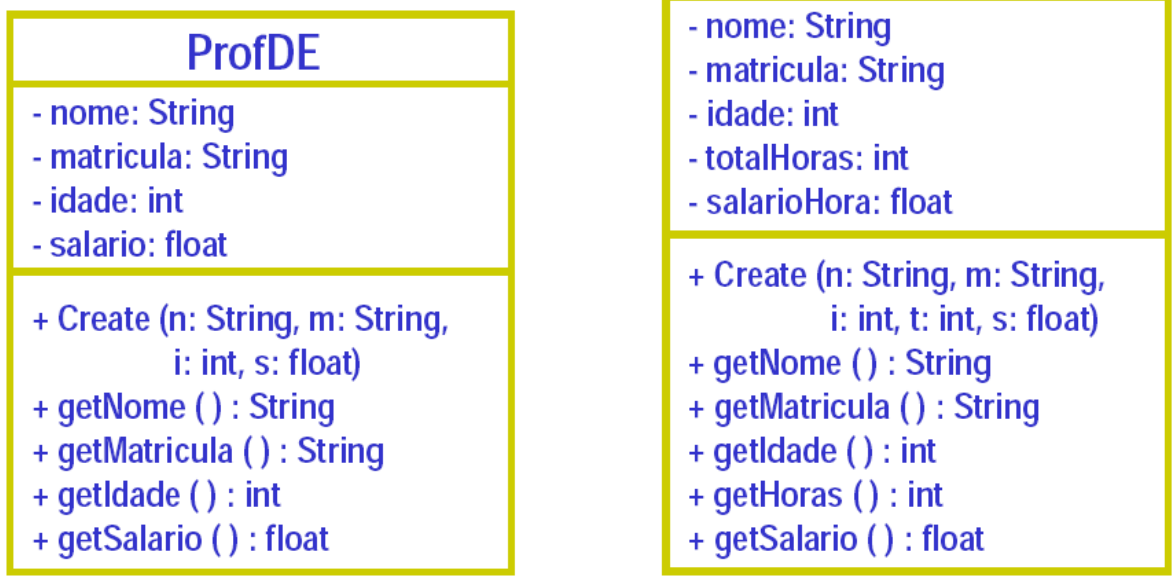
```
double BasePlusComissionEmployee::earnings() const{
    return getBaseSalary() +
        ComissionEmployee::earnings();

void BasePlusComissionEmployee::print() const{
    cout << "\n base-salaried ";

    ComissionEmployee::print();

    cout << "\n base salary: " << baseSalary;
}
```

Exercício I

- Os professores de uma universidade dividem-se em 2 categorias
  - professores em dedicação exclusiva (DE)
  - professores horistas
- Professores em dedicação exclusiva possuem um salário fixo para 40 horas de atividade semanais.
- Professores horistas recebem um valor estipulado por hora.

Exercício II

- Problema pode ser resolvido através da seguinte modelagem de classes:

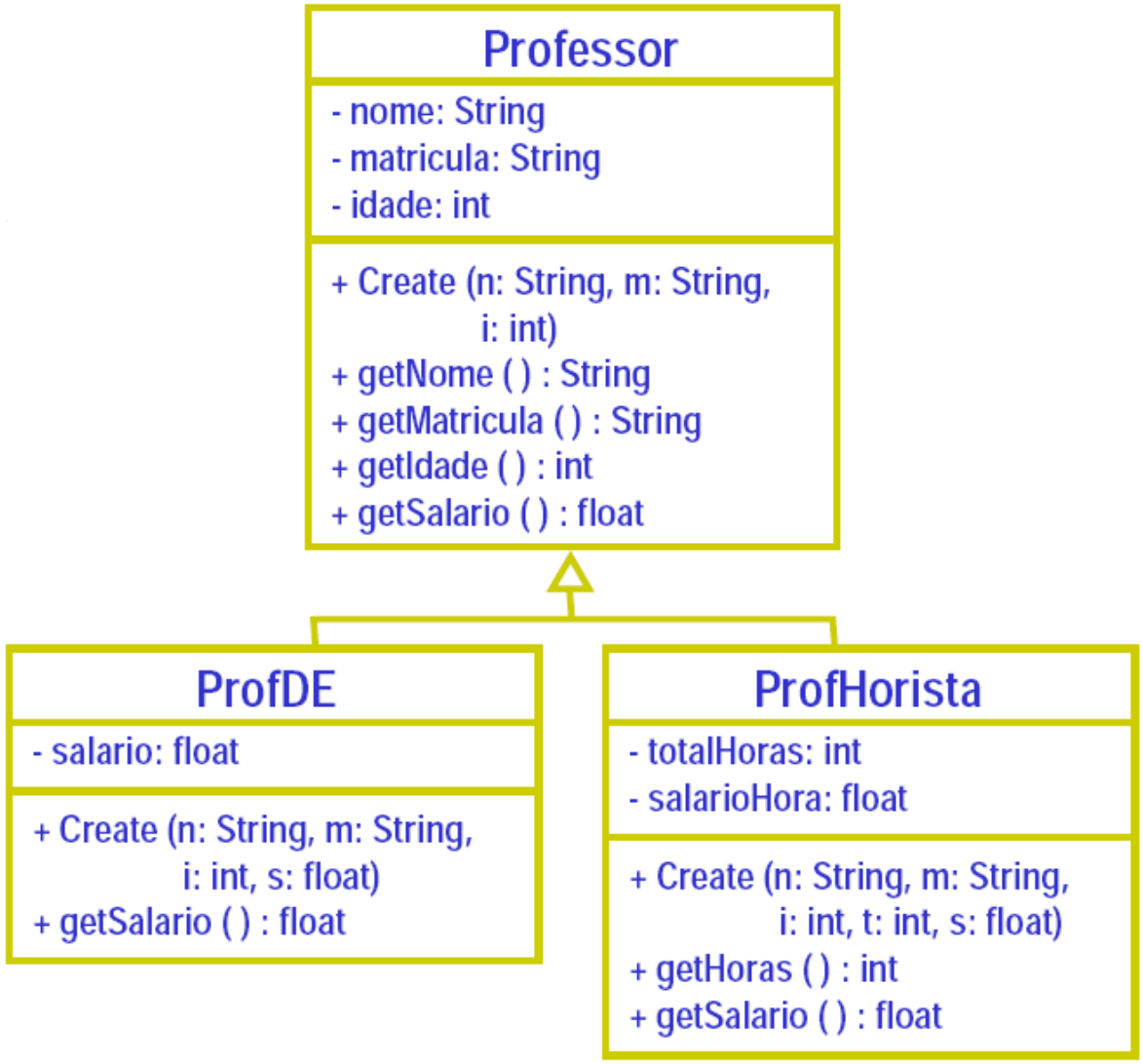


Exercício III

- Analisando a solução:
  - Alguns atributos e métodos são iguais
  - Como resolver? **Herança!**
- Sendo assim:
  - Cria-se uma classe *Professor*, que contém os atributos e métodos comuns aos dois tipos de professor:
  - A partir dela, cria-se duas novas classes, que representarão os professores horistas e DE.
  - Para isso, essas classes deverão "**herdar**" os atributos e métodos declarados pela classe "pai", *Professor*.

Exercício IV

- Solução



FIM