UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

Programação e Desenvolvimento de Software 2

Programação Orientada a Objetos (Wrap-up/Hands-on)

Prof. Douglas G. Macharet
douglas.macharet@dcc.ufmg.br

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Introdução

- Abstração
  - Representação de detalhes relevantes do domínio do problema na linguagem de solução
- Encapsulamento
  - Um sistema orientado a objetos baseia-se no contrato e não na implementação interna
- Herança
  - Permite a hierarquização das classes

Herança e Composição

- Herança
  - Relação do tipo "é um" (is-a)
  - Estudante **é uma** Pessoa
- Composição
  - Relação do tipo "tem um" (has-a)
  - Estudante **tem um** Curso

Herança e Composição

- Como escolher/saber qual utilizar?
- O TipoB vai manter toda o contrato do TipoA, e poderá ser usado onde o TipoA é esperado?
  - Herança
- O TipoB deseja apenas utilizar parte do comportamento exposto pelo TipoA?
  - Composição

Herança e Composição
Boas práticas e recomendações

- Composição
  - Seja crítico em relação número de membros
  - "7 ± 2" [Miller, 1956]
    - Número de itens que uma pessoa consegue lembrar enquanto executando outras tarefas
  - Decompor a classe em outras menores

Herança e Composição
Boas práticas e recomendações

- Herança
  - Mova interfaces, dados e comportamentos comuns o mais alto possível na hierarquia
  - Suspeite de classes base com apenas uma classe derivada (futuro/presente)
  - Evite hierarquias muito profundas
  - Atenção ao encapsulamento (protected)

Classes abstratas

- Uma Classe que não pode ser instanciada
- Define um conjunto de métodos
  - Totalmente implementados
  - Parcialmente implementados (contrato)
- Usadas na implementação de outra classe
- Pelo menos uma função virtual pura
  - Quando é usado um especificador-puro (= 0)

https://en.cppreference.com/book/intro/abstract_classes

Classes abstratas

```
class Ponto {
  public:
    int _x, _y, _z;

    Ponto(int x, int y, int z) : _x(x), _y(y), _z(z) {}

    virtual void imprimirInfo() = 0;

    void testeMetodoNaoVirtual() {
      cout << "Metodo implementado!" << endl;
    }
};

class Ponto2D : public Ponto {
  public:
    Ponto2D(int x, int y) : Ponto(x, y, 0) {}

    void imprimirInfo() {
      cout << "Ponto 2D [" << _x << "," << _y << "]" << endl;
    }
};
```

Ponto → Ponto2D

Classes abstratas

```
int main() {
  Ponto p1;

  Ponto2D p2(10, 20);
  p2.imprimirInfo();
  p2.testeMetodoNaoVirtual();

  return 0;
}
```

Erro! Não pode-se instanciar essa classe!

https://wandbox.org/permlink/oT9GY3xietMvuzIr

Interfaces

- Possui unicamente o papel de um contrato
  - Imposição de que uma determinada Classe irá implementar um certo grupo de métodos
- C++ não implementa explicitamente
  - Outras linguagem sim, por exemplo, Java
  - Semelhante a uma Classe Abstrata, porém possui só métodos que são puramente virtuais

Interfaces

```
class Interface {

  public:
    //Contrato
    virtual void method() = 0;

};

class ClasseConcreta : public Interface {

  public:
    void method() {
      cout << "Comportamento definido." << endl;
    }

};
```


Exemplo

```
class Player {
  public:
    virtual void play() = 0;
    virtual void stop() = 0;
    virtual void pause() = 0;
    virtual void reverse() = 0;
};
```

```
class DVDPlayer : public Player {
  public:
    void play() { cout << "DVD->play()" << endl; };
    void stop() { cout << "DVD->stop()" << endl; };
    void pause() { cout << "DVD->pause()" << endl; };
    void reverse() { cout << "DVD->reverse()" << endl; };
};
```

```
class CDPlayer : public Player {
  public:
    void play() { cout << "CD->play()" << endl; };
    void stop() { cout << "CD->stop()" << endl; };
    void pause() { cout << "CD->pause()" << endl; };
    void reverse() { cout << "CD->reverse()" << endl; };
};
```

Exemplo

```
class Recorder: public Player {
  public:
    virtual void record() = 0;
};
```

```
class TapePlayer : public Recorder {
  public:
    void play() { cout << "Tape->play()" << endl; };
    void stop() { cout << "Tape->stop()" << endl; };
    void pause() { cout << "Tape->pause()" << endl; };
    void reverse() { cout << "Tape->reverse()" << endl; };
    void record() { cout << "Tape->record()" << endl; };
};
```


Exemplo

```
int main() {

  CDPlayer cd;
  cd.play();

  TapePlayer* tape = new TapePlayer();
  tape->record();









  return 0;
}
```

https://wandbox.org/permlink/g2ori6VjqTxcSg3o

```
class Studio {
  public:
    Player* player;
};
```

Dependa de abstrações, não implementações!

Conversão de tipo
Type Casting

- Conversão de um tipo para outro
  - Implítica (feita pelo compilador)
  - Explícita (feita pelo programador)
- Converte a referência de um objeto
  - O Objeto em si não é alterado (memória)
  - O novo tipo deve ser compatível

Conversão de tipo
Type Casting – Exemplo

```
class ClasseBase {

  public:
    virtual void metodoA() { cout << "ClasseBase->MetodoA." << endl; }

};

class ClasseDerivada : public ClasseBase {

  public:
    int atributo;

    ClasseDerivada(int valor) : atributo(valor) { }

    void metodoA() override { cout << "ClasseDerivada->MetodoA" << endl; }

    void metodoB() { cout << atributo << endl; }

};
```


Conversão de tipo
Type Casting – Exemplo

```
int main() {

  ClasseBase *b = new ClasseDerivada(123);
  b->metodoA();                          ——→ "ClasseDerivada->MetodoA"

  b->metodoB();                          ——→ ERRO!

  ClasseDerivada *d = (ClasseDerivada*) b;
  d->metodoA();                          ——→ "ClasseDerivada->MetodoA"

  d->metodoB();                          ——→ 123

  delete d;
  return 0;
}
```

https://wandbox.org/permlink/DDDjeNrmNa2Pp6Gv ⚠

Conversão de tipo
Type Casting

- Classe / Objeto
  - Tipo Estático → Contrato
  - Tipo Dinâmico → Comportamento
- C++
  - Possui operadores auxiliaries para isso
- Não é uma boa prática de implementação
  - Sempre que possível utilizar polimorfismo!

Exercício

- Modelar um sistema de gestão acadêmica
  - Quais entidades deveriam existir?
  - Quais atributos devem existir em cada uma?
  - Quais métodos devem existir em cada uma?

Exercício

```
class Pessoa {

  public:
    string nome;
    int cpf;

    Pessoa(string nome, int cpf) : nome(nome), cpf(cpf) { }

};
```


Exercício

```
class Pessoa {

  private:
    string _nome;
    int _cpf;

  public:
    Pessoa(string nome, int cpf) : _nome(nome), _cpf(cpf) { }

    string getNome() { return this->_nome; };
    int getCPF() { return this->_cpf; };

};
```


Exercício

```
class Estudante : public Pessoa {

  private:
    int _matricula;
    Curso* _curso;

  public:
    Estudante(string nome, int cpf, int matricula, Curso curso) :
      _matricula(matricula), _curso(curso) { }

};
```

Exercício

```
class Estudante : public Pessoa {

  private:
    int _matricula;
    Curso* _curso;

  public:
    Estudante(string nome, int cpf, int matricula, Curso curso) :
      Pessoa(nome, cpf), _matricula(matricula), _curso(curso) { }

};
```


Exercício

```
class Curso {

  private:
    string _nome;
    int _codigo;

  public:
    Curso(string nome, int codigo) : _nome(nome), _codigo(codigo) { }

    string getNome() { return this->_nome; };
    int getCodigo() { return this->_codigo; };

};
```

Exercício

```
class Pessoa {

  private:
    string _nome;
    int _cpf;

  public:
    Pessoa(string nome, int cpf) : _nome(nome), _cpf(cpf) { }

    string getNome() { return this->_nome; };
    int getCPF() { return this->_cpf; };

    virtual void meuNome() {
      cout << "PESSOA: " << getNome() << endl;
    }

};
```


Exercício

```
class Estudante : public Pessoa {

  private:
    int _matricula;
    Curso* _curso;

  public:
    Estudante(string nome, int cpf, int matricula, Curso* curso) :
      Pessoa(nome, cpf), _matricula(matricula), _curso(curso) { }

    void meuNome() override {
      cout << "ESTUDANTE: " << getNome() << endl;
    }

    void meuCurso() {
      cout << "ESTUDANTE->Curso: " << _curso->getNome() << endl;
    }

};
```


Exercício

```
class Professor : public Pessoa {

  private:
    Departamento* _departamento;

  public:
    Professor(string nome, int cpf, Departamento* departamento) :
      Pessoa(nome, cpf), _departamento(departamento) { }

    void meuNome() override {
      cout << "PROFESSOR: " << getNome() << endl;
    }

    void meuDepartamento() {
      cout << "PROFESSOR->Departamento: " << _departamento->getNome() << endl;
    }

};
```


Exercício

```
class Departamento {

  private:
    string _nome;
    int _codigo;

  public:
    Departamento(string nome, int codigo) : _nome(nome), _codigo(codigo) { }

    string getNome() { return this->_nome; };
    int getCodigo() { return this->_codigo; };

};
```


Exercício

```
int main() {

    Pessoa p("Pi", 123);
    p.meuNome();

    Curso curso("Computacao", 777);

    Estudante estudante("Joao", 000, 123, &curso);
    estudante.meuNome();
    estudante.meuCurso();

    Professor* prof = new Professor("Douglas", 999, new Departamento("DCC", 1));
    prof->meuNome();
    prof->meuDepartamento();

    return 0;

}
```


Exercício

```
class Pessoa {

  private:
    string _nome;
    int _cpf;

  public:
    Pessoa(string nome, int cpf) : _nome(nome), _cpf(cpf) { }

    string getNome() { return this->_nome; };
    int getCPF() { return this->_cpf; };

    virtual void meuNome() = 0;

};
```

Exercício

```
int main() {

    {...}

    list<Pessoa*> pessoas;
    pessoas.push_back(prof);
    pessoas.push_back(&estudante);

    for (Pessoa* p : pessoas) {
      p->meuNome();
    }

    return 0;

}
```

https://wandbox.org/permlink/uZVNSyCNM3BRYjr5

Exercício

- Tarefas
  - Modularizar o código (.hpp, .cpp)
  - Depurar
    - Existem vazamentos de memória?
    - Resolvê-los!
  - Dividir a classe Estudante
    - EstudanteGraduacao, EstudantePos