Introdução à Programação Orientada a Objetos: Conceitos, aplicações, linguagens

Carlos Olarte

31 de Julho de 2019

Índice

1. Motivação
2. Conceitos fundamentais
3. Linguagems de Programação OO
4. Conclusões

Sistemas complexos

Considere um sistema de telefonia:

- Vários Usuários: pessoal técnico, suporte ao usuário, cobrança, venda de serviços, etc.
- Muitas entidades: contratos, chamadas, clientes, cupons fiscais, reclamações/queixas, telefones, etc.
- Um número muito grande de relações: um usuário possui vários contratos, um contrato gera muitos cupons fiscais, uma linha está associada a um só contrato, etc.

Desafio

Como abordar a construção desse sistema sem morrer no processo?

Sistemas complexos

Possíveis Abordagens

- Programação não estruturada: um grave erro!
- Programação estruturada: não tem mecanismos explícitos para ocultar informação.

Programação Orientada a Objetos

- Entidades reais (do mundo) correspondem a classes.
- Peças de código podem ser substituídas facilmente.
- As relações entre entidades são explícitas.
- Mecanismos para construir e organizar abstrações/refinamentos (herança).

Programação Orientada a Objetos

- Modelo geral (não atrelado a nenhuma linguagem particular).
- Surge como a evolução da programação estruturada para controlar o acesso as estruturas de dados (encapsulamento).
- Modelo baseado em estados (*stateful*).
- Boa escalabilidade: desde simples aplicações até sistemas complexos.

O *design* baseado em OO é similar ao jeito em que as pessoas pensam!

Um exemplo

"Considere um sistema que permite às pessoas enviar flores. De cada pessoa se conhece seu nome, endereço e cidade. Cada pacote de flores tem um preço. Um florista pode pedir a um outro florista entregar as flores. Os pedidos de envio tem 3 estados: solicitado, em processo, entregue. Todo comerciante deve poder emitir um cupom fiscal. Os floristas são comerciantes."

Classe: Abstração para agrupar objetos comuns que têm o mesmo comportamento.

Classe

Membros de classe

Atributos

Características particulares de cada objeto:

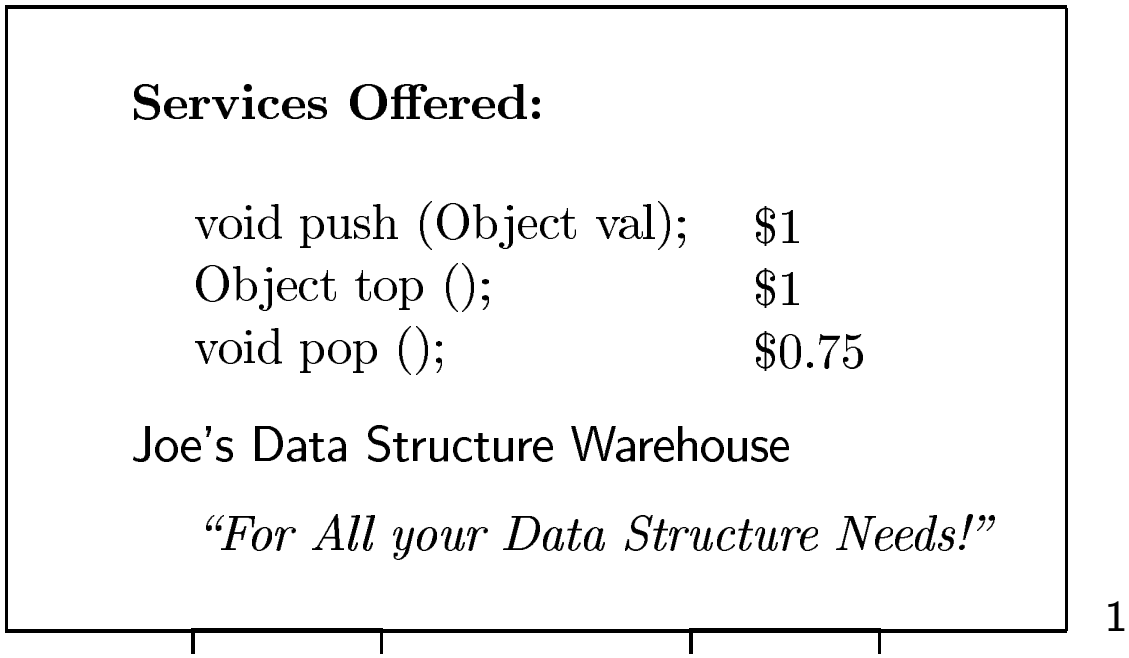
- Uma pessoa possui: nome, telefone, endereço, etc.
- Um cupom fiscal possui: número, valor, data, etc.

Métodos (e construtores)

Definem o comportamento da classe:

- Um florista pode enviar flores
- Um comerciante pode emitir um cupom fiscal
- Uma pessoa pode mudar de endereço

Assinatura e Contratos



Services Offered:

void push (Object val);	$1
Object top ();	$1
void pop ();	$0.75

Joe's Data Structure Warehouse

"For All your Data Structure Needs!"

1

Os métodos definem os serviços que uma classe oferece a seus usuários.

[1] Tomado de [Bud01]

Objetos

Objetos são instâncias de uma classe:

- Encapsulam um estado
- Respondem às mensagens com a execução de um método.

Os objetos de uma mesma classe compartem o comportamento definido pela sua classe.

Até agora

- Um programa OO é uma comunidade de agentes chamados objetos.
- Os objetos oferecem serviços que são definidos em uma classe.
- Os objetos enviam mensagens a outros objetos, que respondem executando um método.
- Um objeto encapsula o estado e a linguagem protege o acesso aos dados.

Dados estruturados vs POO

Defina o tipo estruturado `Complexo` para representar um número complexo com a parte real (`float`) e a parte imaginária (`float`). Utilizando o tipo `Complexo`,

- Faça uma função que imprima na tela um número complexo (e.g., `3.4 + 2.1i`).
- Faça uma função que retorna a soma de dois números complexos.
- Faça una função que retorna o módulo de um número complexo:

$$z = \sqrt{real^2 + img^2}$$

- Faça um bloco main para testar as funções acima.

Dados estruturados vs POO

```
// Estrutura de dados
struct Complexo{
  float real, img;
};
// Imprimir (e.g, 3 + 2i)
void print(Complexo x){
  cout<<x.real<<" + " <<x.img<<"i"<<endl;
}
// Módulo
float modulo(Complexo x){
  return sqrt(pow(x.real,2.0) + pow(x.img,2.0));
}
// Soma de complexos
Complexo soma(Complexo x, Complexo y){
  Complexo z;
  z.real = x.real + y.real;
  z.img = x.img + y.img;
  return z;
}
```

Dados estruturados vs POO

```
int main(void){
  Complexo c, d;
  c.real = 3.1;
  c.img = 5.4;
  d.real = 2.2;
  d.img = 7.0;

  cout<<modulo(d)<<endl;
  print(soma(c,d));
  return 0;
}
```

Dados estruturados vs POO

Versão Java

```
class Complexo {
  private float img, real; // Atributos privados
  // construtor (inicializar os atributos )
  public Complexo(){
                this.real = 0;
                this.img=0;
  }
  // Acesso controlado aos dados
  public float getReal(){ return this.real; }
  public void setReal(float x){ this.real = x; }

  public float Modulo(){
  //this: referência ao (estado do) próprio objeto
   return Math.sqrt(
      Math.pow(this.real,2) +
      Math.pow(this.img,2));
  }
}
```

Dados estruturados vs POO

```
public static void main (String arg[]){
   Complexo c = new Complexo();

   // c.real = 5; ERRO!!

   c.setReal(5); // Acesso controlado
}
```

Dados estruturados vs POO

Versão Python

```
import math
class Complexo:
    '''Representação dos números complexos'''
    def __init__(self, real, img):
        '''Inicializar a parte real e a parte
           imaginária'''
        self.real = real
        self.img = img

    def module(self):
        '''módulo de um número complexo'''
        return math.sqrt(math.pow(self.real,2) +
                         math.pow(self.img,2));

    def __str__(self):
        '''string representando o número'''
        return f'{self.real} + {self.img}i'
```

Dados estruturados vs POO

Versão Python

```
C = Complexo(3,2)
print(C)
print(C.module())
```

Herança

Considere o exemplo a seguir:

Defina os tipos estruturados `Quadrado` e `Círculo` e faça as seguintes funções:

- `area1`: para retornar a área de um quadrado.
- `area2`: para retornar a área de um círculo.
- `area3`: para retornar o somatório das áreas de um vetor de quadrados.
- `area4`: para retornar o somatório das áreas de um vetor de círculos.
- `area5`: dados um vetor de quadrados e um vetor de círculos, retornar o somatório de todas as áreas.

Herança

```
struct Circulo{
  double raio;
};

struct Quadrado{
  double lado;
};

double area(Circulo c){
   return M_PI * pow(c.raio,2);
}

double area(Quadrado q){
   return pow(q.lado,2);
}
```

Herança

```
double sum_areas(Quadrado vq[], int n){
  double soma = 0;
  for(int i=0;i<n;i++){
    soma += area(vq[i]);
  }
  return soma;
}

double sum_areas(Circulo vc[], int n){
  double soma = 0;
  for(int i=0;i<n;i++){
    soma += area(vc[i]);
  }
  return soma;
}
```

Herança

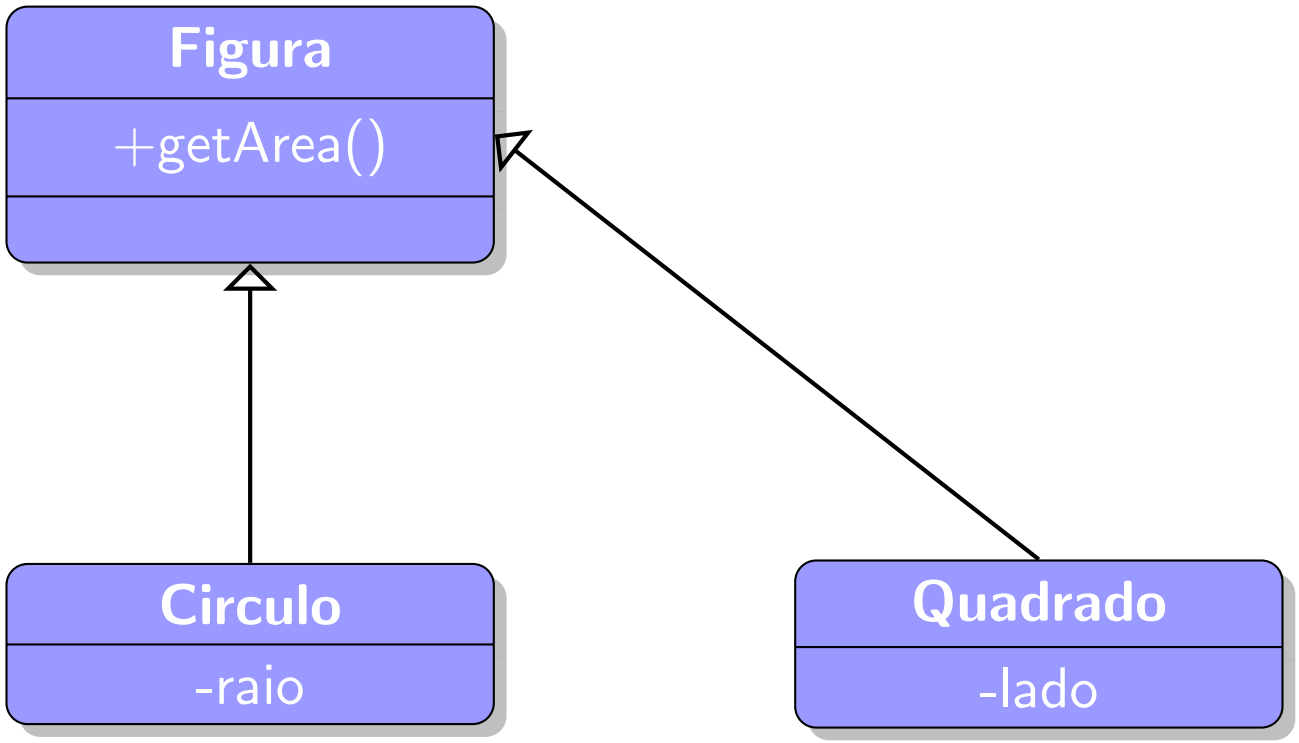
```
double sum_areas(Circulo vc[], int n, Quadrado vq
    [], int m){
  double soma = 0;
  soma += sum_areas(vc,n);
  soma += sum_areas(vq,m);
  return soma;
}

int main(void){
  Quadrado vq[] = {{3.2},{5.3}};
  Circulo vc[] = {{5.4},{1.2}};
  cout<<"Circulos:"<<sum_areas(vc,2)<<endl;
  cout<<"Quadrados:"<<sum_areas(vq,2)<<endl;
  cout<<"Total:"<<sum_areas(vc,2,vq,2)<<endl;
  return 0;
}
```

Herança

Figuras Geométricas

- Quadrados e Círculos são figuras.
- Toda figura deveria oferecer um método para calcular a área.



Polimorfismo

```
import math

class Figura:
    '''Abstração de uma figura'''
    def area(self):
        return 0

class Circulo(Figura):
    '''Representação de um círculo'''
    def __init__(self, raio):
        self.raio = raio

    def area(self):
        return math.pi * math.pow(self.raio,2)
```

Polimorfismo

```
class Quadrado(Figura):
    '''Um quadrado'''
    def __init__(self,lado):
        self.lado = lado
    def area(self):
        return math.pow(self.lado,2)
```

O vetor `figuras` pode conter círculos e quadrados. A chamada ao método `area()` é polimórfica !!

```
def sum_areas(figuras):
    '''Somatório das áreas das figuras'''
    soma=0
    for f in figuras:
        soma += f.area() #Chamada polimórfica
    return soma

L = [Circulo(2), Quadrado(5), Circulo(4)]
print(sum_areas(L))
```

Polimorfismo

Polimorfismo

Habilidade para reagir de maneira diferente à mesma messagem.

fala()

AUAUA!

fala()

MIAU!

OO vs Prog. Estruturada

Por que preciso de OO?

As linguagens OO permitem encapsular o estado dos objetos e ocultar informação que não é pública.

- A herança permite construir abstrações de forma incremental.
- O mecanismo de herança permite reutilizar código em um projeto.
- O polimorfismo é uma habilidade muito poderosa que simplifica o código.

Resumindo

Desenvolvendo aplicações OO

Software reutilizable

- Os componentes podem ser reutilizáveis.
- Cada componente tem uma interface bem definida!
- Catálogos de componentes: estruturas de dados básicas, conexão a bancos de dados, componentes gráficos, etc.

Linguagem de POO

Um pouco de história

- Origem: 60's (Simula). Herança
- Ganhou popularidade nos 80's (C++)
- Desenvolvimentos teóricos (80s): (Eiffel, Smaltalk)
- Novas linguagem (90's, 2000's): Java, C#, Objective C
- Metodologias de desenvolvimento (90s): Booch, Jacobson, Rumbaugh, UML
- (2000): Concorrência, JML

OO has become THE dominant programming paradigm. [a]

[a] Timothy Budd

Linguagem de POO

Algumas diferenças:

- Puras: Toda construção na linguagem é um objeto (e.g. Eiffel)
- Não Puras: Objetos co-existem com procedimentos e funções (C++, Object Pascal, Java, Python, etc)
- Herança simples (Java) ou múltipla (Python, C++).
- Interpretados (Python) / Compilados (C++, Java)
- Mecanismos de reflexão e introspeção.
- Controle da memória (garbage collection)

Tendências

Verificação do contratos (JML)

```
//@ requires count > 0;
//@ ensures count  == \old(count) - 1;
void decrement(){
   count --;
}
```

Conclusões

OO provê muitas vantagens para desenvolver software:

- As abstrações do mundo se representam como classes.
- As classes podem se organizar em hierarquias de herança.
- O modelo OO permite a reutilização de código (contratos).
- Os objetos encapsulam todo o comportamento e assim, a depuração é mais fácil.
- Muitos padrões de design suportam o desenvolvimento de software.

Conclusões

Mas cuidado....

- Uma linguagem OO não força a utilizar *o modelo* OO corretamente.
- Nesta disciplina o mais importante é o modelo OO... não uma linguagem particular!


Martin Abadi and Luca Cardelli.
A theory of Objects.
Springer, 1996.


Grady Booch, James Rumbaugh, and Ivar Jacobson.
The Unified Modeling Language User Guide.
Addison-Wesley, 2005.


Timothy A. Budd.
An Introduction to Object Oriented Programming.
Addison-Wesley, 2001.


Paul Deitel and Harvey Deitel.
Java How to Program.
Prentice Hall, 2011.


Paul Deitel and Harvey Deitel.
C++ How to Program.
Prentice Hall, 2013.


Java API specification.

http://docs.oracle.com/javase/6/docs/api/.


Barbara Liskov and John Guttag.
Program Development in Java.
Pearson, 2005.


Bertrand Meyer.
Object-Oriented Software Construction.
Prentice Hall, 1997.


The plural tool.
http://code.google.com/p/pluralism/.


Peter Van Roy and Seif Haridi.
Concepts, Techniques, and Models of Computer Programming.
MIT Press, 2004.


Sven Stork, Paulo Marques, and Jonathan Aldrich.
Concurrency by default: using permissions to express dataflow in stateful programs.
In Shail Arora and Gary T. Leavens, editors, *OOPSLA Companion*, pages 933–940. ACM, 2009.