www.devmedia.com.br
[versão para impressão]
Link original: https://www.devmedia.com.br/programacao-orientada-a-objetos-versus-programacao-estruturada/32813

Programação Orientada a Objetos versus Programação Estruturada

Vamos abordar neste artigo qual tipo de programação deve ser utilizada no decorrer do desenvolvimento: orientada a objetos ou programação estruturada.

Vamos abordar neste artigo os dois tipos de programação bem conhecidos e que ainda geram algumas dúvidas em suas utilizações e definições, que é a **programação orientada a objetos** e a **programação estruturada**.

Existem dois tipos de programação bem usuais: Orientada a Objetos (OO) e Estruturada. Em cada uma delas iremos conhecer os seus principais conceitos e suas respectivas utilizações e, por fim, mostraremos um breve comparativo entre esses dois tipos e qual o melhor a ser utilizado quando um projeto de desenvolvimento de software for iniciado. Utilizaremos como linguagem de programação base para os nossos exemplos as linguagens Java e C#.

Quer aprender mais sobre **Orientação a objetos em Java**? Confira o curso de Java básico orientado a objetos da DevMedia.

Programação Orientada a Objetos

A *programação orientada a objetos* é um modelo de programação onde diversas classes possuem características que definem um objeto na vida real. Cada classe determina o comportamento do objeto definido por métodos e seus estados possíveis definidos por atributos. São exemplos de linguagens de programação orientadas a objetos: *C++, Java, C#, Object Pascal, entre outras*. Este modelo foi criado com o intuito de aproximar o mundo real do mundo virtual. Para dar suporte à definição de Objeto, foi criada uma estrutura chamada Classe, que reúne objetos com características em comum, descreve todos os serviços disponíveis por seus objetos e quais informações podem ser armazenadas.

Vamos começar exemplificando uma classe carro utilizando a *linguagem C#*, como mostra a **Listagem 1**.

Listagem 1. Classe Carro

```
class Carro {
     string marca; // Define a marca do carro
     string modelo; // Define o modelo do carro
     int ano; // Define o ano de fabricação do carro
     string categoria; // Define a categoria, por exemplo carro de passeio, utilitário...
     double potencia; // Define a potência do carro
     double autonomia; // Define a autonomia em km
     boolean ligado; // Define se o carro está ligado

     public Carro() {

     }

     public Carro(string marca, string modelo, int ano, string categoria, double potencia, double autonomia, boolean ligado) {
         this.marca = marca;
         this.modelo = modelo;
         this.ano = ano;
         this.categoria = categoria;
         this.potencia = potencia;
         this.autonomia = autonomia;
         this.ligado = ligado;
     }
 }
```

A classe “Carro” contém um conjunto de características em comum que definem um objeto do tipo carro: Marca, modelo, ano, categoria, potência e autonomia são atributos que todos os carros precisam ter. Temos também dois construtores para a classe, ou seja, os responsáveis por criar o objeto em memória, onde um é inicializa o objeto carro e o outro, além da inicialização do objeto, define que o preenchimento dos parâmetros seja obrigatório nos atributos da classe.

Agora vamos criar um objeto para que o mesmo possa ser utilizado, como mostra a **Listagem 2**.

Listagem 2. Invocando o construtor e criando um objeto

```
class Program {
    public static void main(String[] args) {
        // Instanciando a classe carro a partir do construtor sem a passagem de parâmetros
        Carro meuCarro = new Carro();

        // Atribuindo valores aos atributos que compõem a classe carro
        meuCarro.marca = “Fiat”;
        meuCarro.modelo = “Palio”;
        meuCarro.ano = 2013;
        meuCarro.categoria = “Passeio”;
        meuCarro.potencia = 86.7;
        meuCarro.autonomia = 320.6;
        meuCarro.ligado = false;
    }
}
```

Instanciamos nossa classe Carro e atribuímos a uma variável “meuCarro” todos os atributos da nossa classe. Poderíamos também invocar o construtor que já recebe os parâmetros com os respectivos atributos da classe, como mostra a **Listagem 3**.

Listagem 3. Invocando o construtor com parâmetros e criando o objeto

```
class Program {
      public static void main(String[] args) {
          // Instanciando a classe carro a partir do construtor que contêm os devidos parâmetros
          Carro meuCarro = new Carro("Fiat", "Palio", 2013, "Passeio", 86.7, 420.5, false);
      }
}
```

Na **Listagem 4** criaremos alguns métodos, que é são comportamentos que a classe Carro possui.

Listagem 4. Métodos da classe Carro

```
void Ligar() {
      // Atribui true a propriedade ligado
      this.ligado = true;
      System.out.println("Carro ligado!");
}

boolean Anda(double autonomia) {
    // Condição que verifica se o carro está ligado e se autonomia é maior que 0,
    // retornando verdadeiro caso a condição seja satisfeita, ou falso caso contrário
    if (this.ligado && autonomia > 0) {
        return true;
    } else {
        return false;
    }
}

void Desligar() {
    // Atribui false a propriedade ligado
    this.ligado = false;
    System.out.println("Carro desligado!");
}
```

No método “Liga” atribuímos ao atributo “ligado” o valor verdadeiro e escrevemos no console de saída uma informação que o carro foi ligado.

No método “Anda” temos uma condição onde se o carro estiver ligado e a autonomia do carro for maior que zero, o carro ainda pode andar. Caso contrário, se a autonomia for menor ou igual a zero ou então o carro esteja desligado, o mesmo fica impossibilitado de andar.

E por último temos o método “Desliga”, que atribui o valor falso ao atributo “ligado” e escrevemos no console de saída a mensagem que o carro foi desligado.

Veja que para utilizar esses métodos presentes na classe Carro precisamos utilizar dois conceitos cruciais: Construtores e Destrutores. Construtores são métodos invocados no momento em que o objeto é criado: por convenção é utilizado o mesmo nome da classe como, por exemplo, para a classe Carro temos o construtor Carro(). Já os Destrutores realizam a função inversa aos Construtores, onde são liberados da memória todos os objetos que foram criados no momento da execução do programa, na maioria das linguagens de programação esse método é executado automaticamente quando o objeto é eliminado e, por convenção, é utilizado também o mesmo nome da classe, porém com um “~” antecedendo o nome como, por exemplo, para a classe Carro temos o destrutor ~Carro().

Agora que já conhecemos um pouco sobre orientação a objetos, vamos conhecer seus princípios básicos

Abstração

Este princípio é uma forma de abstrair o quão complexo é um método ou rotina de um sistema, ou seja, o usuário não necessita saber todos os detalhes de como sua implementação foi realizada, apenas para que serve determinada rotina e qual o resultado esperado da mesma. Sendo assim, podemos também dividir internamente problemas complexos em problemas menores, onde resolvemos cada um deles até encontrarmos a solução do problema inteiro. Um exemplo da vida real para ilustrar esse conceito seria o conceito de carro a abstração de um veículo, que é utilizado como meio de transporte por várias pessoas para mover-se de um ponto a outro. Não é necessário que a pessoa informe que irá se locomover com a ajuda de um veículo movido a combustível, contendo rodas e motor. Basta a pessoa informar que utilizará um carro para tal, pois esse objeto é conhecido por todos e abstrai toda essa informação por trás disso.

Veja na **Listagem 5** um exemplo de abstração utilizando a linguagem Java.

Listagem 5. Exemplo de abstração

```
public class Conta {
      int cod_banco;
      int num_conta;
      double saldo;
      double limite;

      void ConsultarSaldo() {
          System.out.println(“Conta: ” + this.num_conta);
          System.out.println(“Saldo: ” + this.saldo);
      }

      void Depositar(double valor) {
          this.saldo = this.saldo + valor;
      }

      void Sacar(double valor) {
          this.saldo = this.saldo - valor;
      }
  }
```

Para este caso, um cliente só precisa entender que uma conta é um local, em um determinado banco, onde é possível ser depositado e sacado valores.Para exemplificar este caso, criamos uma classe Conta com os atributos: código do banco, número da conta, saldo e limite. Criamos também um método “ConsultarSaldo”, onde ele retorna qual o saldo da conta naquele momento. Criamos também outro método chamado “Depositar” onde passamos um valor como parâmetro e ele soma esse ao saldo atual da conta. Outro método chamado “Sacar” foi criado com um valor passado por parâmetro, onde o mesmo subtrai esse valor do saldo atual da conta.

Encapsulamento

O princípio do encapsulamento é a forma pela qual o programa é divido a ponto de se tornar o mais isolado possível, ou seja, cada método pode ser executado isoladamente e retornar um resultado satisfatório ou não para cada situação. Sendo assim, o objeto não necessita conhecer qual forma cada método foi implementado.

Vejamos na **Listagem 6** um exemplo prático de encapsulamento, onde é possível obter e atribuir valores a propriedades da classe Pessoa de forma independente, sem alterar o funcionamento do restante da classe.

Listagem 6. Exemplo de encapsulamento

```
public class Pessoa {
      private String nome; // Define o nome da pessoa
      private String cpf; // Define o cpf da pessoa
      private Date dat_nasc; // Define a data de nascimento da pessoa

      // Obtem o nome da pessoa
      public String getNome() {
          return nome;
      }

      // Atribui o nome a pessoa
      public void setNome(String nome) {
          this.nome = nome;
      }

      // Obtem o cpf da pessoa
      public String getCpf() {
          return cpf;
      }

      // Atribui o cpf a pessoa
      public void setCpf(String cpf) {
          this.cpf = cpf;
      }

      // Obtem a data de nascimento da pessoa
      public Date getDatNasc() {
          return dat_nasc;
      }

      // Atribui a data de nascimento a pessoa
      public void setDatNasc(Date dat_nasc) {
          this.dat_nasc = dat_nasc;
      }
  }
```

Herança

Esse princípio diz que, uma classe pode compartilhar métodos e atributos entre si como, por exemplo, em um sistema escolar, onde temos uma classe Pessoa que contém características que definem uma pessoa. Porém, dentro do sistema temos uma outra classe Funcionário, que contém os mesmos atributos de Pessoa, além de outros atributos que apenas funcionários podem ter. Outro exemplo seria uma classe Aluno, que também contém atributos de pessoas e outros atributos que apenas pertencem a aluno.

Vejamos na **Listagem 7** como podemos implementar esse princípio utilizando a linguagem de programação Java.

Listagem 7. Implementando o princípio da herança

```
public class Pessoa {
      public String nome; // Define o nome da pessoa
      public String cpf;  // Define o cpf da pessoa
      public Date data_nascimento; // Define a data de nascimento da pessoa

      // Construtor da classe Pessoa, passando todos os seus atributos
      public Pessoa(String _nome, String _cpf, Date _data) {
          this.nome = _nome;
          this.cpf = _cpf;
          this.data_nascimento = _data;
      }
  }

  // Classe Aluno que herda da classe Pessoa
  public class Aluno extends Pessoa {
```

```
      // Herda os atributos da classe super
      public Aluno(String _nome, String _cpf, Date _data) {
          super(_nome, _cpf, _data);
      }
      public String matricula; // Define a matricula do Aluno
  }

  public class Professor extends Pessoa {
      // Herda os atributos da classe super
      public Professor(String _nome, String _cpf, Date _data) {
          super(_nome, _cpf, _data);
      }
      public double salario; // Define o salário do professor
      public String disciplina; // Define a disciplina do professor
  }

  public class Funcionario extends Pessoa {
      // Herda os atributos da classe super
      public Funcionario(String _nome, String _cpf, Date _data) {
          super(_nome, _cpf, _data);
      }
      public double salario; // Define o salário do funcionário
      public Date data_admissao; // Define a data de admissão do funcionário
      public String cargo; // Define o cargo do funcionário
  }
```

Note que temos uma classe Pessoa que contém propriedades em comum com as classes Aluno, Professor e Funcionário. Essas outras classes que herdam de Pessoa recebem a palavra reservada “extends”, que indica que as mesmas contêm as propriedades nome, cpf e data, presentes na classe Pessoa.

Polimorfismo

Polimorfismo é uma característica na qual os mesmos atributos ou métodos podem ser utilizados por objetos distintos e com implementações de lógica diferentes. Por exemplo, podemos dizer que um carro e uma bicicleta são veículos utilizados para locomoção e ambos contêm um método “Andar” em comum, porém, a implementação de cada um deles é feita de forma diferente.

Vamos exemplificar a implementação de uma classe Forma, onde a mesma contém um método Desenhar, como mostra a **Listagem 8**.

Listagem 8. Classe Forma

```
public abstract class Forma
  {
      public Forma()
      {
      }

      public virtual void Desenhar(System.Drawing.Graphics g)
      {
      }
  }
```

Note que a classe Forma é uma classe abstrata e que o método desenhar não tem implementação, pois cada figura tem uma forma diferente de ser desenhada. Porém, é nas classes derivadas da classe Forma que esse método será implementado e, por isso, essa nova classe será uma classe virtual, como mostra a **Listagem 9**.

Listagem 9. Classe Circulo

```
using System.Drawing;
  public class Circulo : Forma
  {
      // Implementa o método desenhar baseado na forma Circulo
      public override void Desenhar(System.Drawing.Graphics g)
      {
```

```
            base.Desenhar (g);
            g.DrawEllipse(Pens.Red, 5, 5, 100, 100);
        }
    }
```

Veja agora que a classe Circulo herda da classe Forma, portanto o método Desenhar deve ser implementado. Partindo desse princípio podemos ter diversas classe diferentes que herdem da classe Forma e implementem de outra forma o método Desenhar, como mostra a **Listagem 10**.

Listagem 10. Classe Quadrado

```
public class Quadrado : Forma
  {
      // Implementa o método desenhar baseado na forma Quadrado
      public override void Desenhar(System.Drawing.Graphics g)
      {
          base.Desenhar (g);
          g.DrawRectangle(Pens.Green, 5, 5, 100, 100);
      }
  }
```

Classes abstratas são classes que jamais serão instanciadas diretamente, pois não possuem implementação suficiente para oferecer funcionalidades concretas a serem utilizadas.

Programação Estruturada

O princípio básico da **programação estruturada** é que um programa pode ser divido em três partes que se interligam: *sequência, seleção* e *iteração*.

Sequência

Na sequência são implementados os passos de processamento necessários para descrever determinada funcionalidade. Um exemplo básico seria um fluxograma, onde primeiro é executado a Etapa 1 e após a sua conclusão a Etapa 2 é executada, e assim por diante.

Seleção

Na seleção o fluxo a ser percorrido depende de uma escolha. Existem duas formas básicas para essa escolha.

- A primeira é através do condicional “Se”, onde se uma determinada condição for satisfatória o fluxo a ser corrido é um e, caso contrário, o fluxo passa a ser outro. Ou seja, se o fluxo só percorre apenas um caminho, apenas uma ação é processada.
- A outra forma de escolha é onde o número de condições se estende a serem avaliadas. Por exemplo, se a Condição 1 for verdade faça Processamento 1, caso contrário, se a Condição 2 for verdade faça Processamento 2, caso contrário, se a Condição 3 for verdade faça Processamento 3, e assim por diante.

Iteração

Na iteração é permito a execução de instruções de forma repetida, onde ao fim de cada execução a condição é reavaliada e enquanto seja verdadeira a execução de parte do programa continua.

Modularização

A medida que o sistema vai tomando proporções maiores, é mais viável que o mesmo comece a ser divido em partes menores, onde é possível simplificar uma parte do código deixando a compreensão mais clara e simplificada. Essa técnica ficou conhecida como Subprogramação ou Modularização. No desenvolvimento utilizamos essa técnica através de procedimentos, funções, métodos, rotinas e uma série de outras estruturas. Com essa divisão do programa em partes podemos extrair algumas vantagens, como:

- Cada divisão possui um código mais simplificado;
- Facilita o entendimento, pois as divisões passam a ser independentes;
- Códigos menores são mais fáceis de serem modificados;
- Desenvolvimento do sistema através de uma equipe de programadores;

- Reutilização de trechos de códigos.

Hoje as linguagens estruturadas, como Algol, Pascal e C, ainda são encontradas em muito sistemas, apesar das linguagens Orientadas a objetos serem mais usuais.

Para exemplificar esse tipo de programação, vamos ver um exemplo em C na **Listagem 11**.

Listagem 11. Exemplo de Programação estruturada

```
# include <stdio.h>
  int main()
  {
   int soma, n=1;
   soma = 0;  // inicialização da variável soma
  for (n=1; n<=100; n++)
      soma= soma + n; // atualização da variável soma

  printf("O valor da soma = %d\n",soma);
  return 0;
  }
```

Portanto, como vimos no decorrer do artigo, a programação orientada a objetos define uma programação voltada aos conceitos de classes e herança e, em contrapartida, a programação estruturada define uma programação voltada a procedimentos e funções definidas pelo usuário. Vejamos na **Tabela 1** um breve comparativo entre programação orientada a objetos e programação estruturada.

Programação orientada a objetos	**Programação estruturada**
Métodos	Procedimentos e funções
Instâncias de variáveis	Variáveis
Mensagens	Chamadas a procedimentos e funções
Classes	Tipos de dados definidos pelo usuário
Herança	Não disponível
Polimorfismo	Não disponível

Tabela 1. Comparativo entre programação orientada a objetos e programação estruturada.

Na **Programação estruturada** observamos algumas vantagens como um controle mais eficaz quanto ao fluxo de execução do programa e a facilidade de compreender o código quando o mesmo é analisado. Como principal desvantagem temos a facilidade em desenvolver um código confuso caso o desenvolvedor faça o tratamento dos dados juntamente com o restante da execução do programa, além do que o reuso de código se torna um pouco complicado devido a não definição da tarefa.

Já na programação orientada a objetos temos como vantagens a reutilização de código e a melhor organização do código do programa. Porém, alguns detalhes tornam essa programação um pouco prejudicada quando comparada a programação estruturada como, por exemplo, o desempenho do código e a confusão na hora de aplicar os conceitos de orientação a objetos.

Portanto, não podemos assumir que um tipo de programação é mais vantajoso que o outro, pois em ambos existem características bem peculiares nas suas definições, deixando cada uma mais viável de ser implementada, dependendo do negócio escolhido, para facilitar o bom entendimento e a manutenibilidade futura do código feito.

Manutenibilidade é uma característica inerente a um projeto de sistema ou produto, e se refere à facilidade, precisão, segurança e economia na execução de ações de manutenção nesse sistema ou produto.

Espero que tenham gostado do artigo.

Um forte abraço e até a próxima.