

# Programação I

## Aula 6: Agregação e particionamento

### Apresentação

---

Nesta aula, estudaremos o conceito de particionamento, que visa a decomposição de classes extensas em classes menores, permitindo criar objetos menores e mais simples. Exploraremos também o conceito de relacionamentos entre objetos, desenvolvendo novas aplicações utilizando os conceitos de agregação e particionamento.

### Objetivos

---

- Examinar o conceito de agregação e particionamento;
- Aplicar os conceitos de agregação e particionamento em Java.

### Primeiras palavras

---

A programação orientada a objetos nos ajuda a resolver de forma mais simples problemas com alta complexidade. A agregação de classes é um conceito voltado a facilitar a solução de problemas muito complexos. Podemos dividir uma classe em classes menores, particionando esta classe em diversas outras classes mais simples, para posteriormente as reunirmos em conjunto, formando uma classe maior e mais complexa.

**Atenção!** Aqui existe uma videoaula, acesso pelo conteúdo online



Fonte: Shutterstock.

## Particionamento

É a decomposição de classes extensas em classes menores, que podem ser mais bem reaproveitadas em outras classes, além de permitir melhor controle e manutenção.

O **particionamento de classes** nos permite criar objetos menores e mais simples, que poderão ser reunidos em conjunto, capazes de criar novas classes, maiores e mais complexas.

Vamos imaginar um computador do tipo Desktop, que é um objeto bem complexo, com diferentes partes. Muitas dessas partes são usadas por outros objetos também, tal como Notebooks e Servidores.

Um HD (HardDisk), por exemplo, pode ser usado por cada um deles, assim como a placa de vídeo, a placa-mãe, o vídeo, a memória, além de outros dispositivos. Se formos criar uma classe para representar um Desktop, teremos uma classe com muitos atributos, o que a tornaria grande e complexa, com muitas linhas de código e de difícil manutenção. Isso porque, se fosse necessário realizar alguma mudança, teríamos que trabalhar em uma classe altamente complexa. Outro ponto importante seria a criação das classes Notebook e Servidor, que seriam igualmente complexas, sem contar que, para realizar uma alteração em um único componente que fosse, teríamos que fazê-la em todas as três classes (Desktop, **Notebook e Servidor**).

Inicialmente, nossa classe desktop ficaria com os seguintes atributos:

## Classe

## Atributos

Desktop	MarcaPlacaMae : texto ModeloPlacaMae : texto PrecoPlacaMae: real TipoProcessador : texto MarcaHD : texto ModeloHD : texto PrecoHD : real TipoHD : texto CapacidadeHD : inteiro MarcaPlacaVideo : texto ModeloPlacaVideo : texto PrecoPlacaVideo : real Padrao : texto MarcaMemoria : texto ModeloMemoria : texto PrecoMemoria: real TipoMemoria : texto CapacidadeMemoria : inteiro
---------	--

Podemos analisar como ficará a classe Desktop:



Desktop

MarcaPlacaMae : texto  
ModeloPlacaMae : texto  
PrecoPlacaMae: real  
TipoProcessador : texto  
MarcaHD : texto  
ModeloHD : texto  
PrecoHD : real  
TipoHD : texto  
CapacidadeHD : inteiro  
MarcaPlacaVideo : texto  
ModeloPlacaVideo : texto  
PrecoPlacaVideo : real  
Padrao : texto  
MarcaMemoria : texto  
ModeloMemoria : texto  
PrecoMemoria: real  
TipoMemoria : texto  
CapacidadeMemoria : inteiro

### Declaração dos atributos da classe Desktop:

```
public class Desktop {  
    public String marcaPlacaMae, modeloPlacaMae;  
    public double precoPlacaMae;  
    public String tipoProcessador, marcaHD, modeloHD;  
    public double precoHD;  
    public String tipoHD;  
    public int capacidadeHD;  
    public String marcaPlacaVideo, modeloPlacaVideo;  
    public double precoPlacaVideo;  
    public String padrao, marcaMemoria, modeloMemoria;  
    public double precoMemoria;  
    public String tipoMemoria;  
    public int capacidadeMemoria;  
}
```

A classe ficou extensa, complexa, sem contar que precisaremos ainda incluir os métodos de acesso (Setters e Getters), construtores, entradaDados, imprimir, cadastrar, além de outros métodos que possam se fazer necessários, sem contar a grande quantidade de atributos.

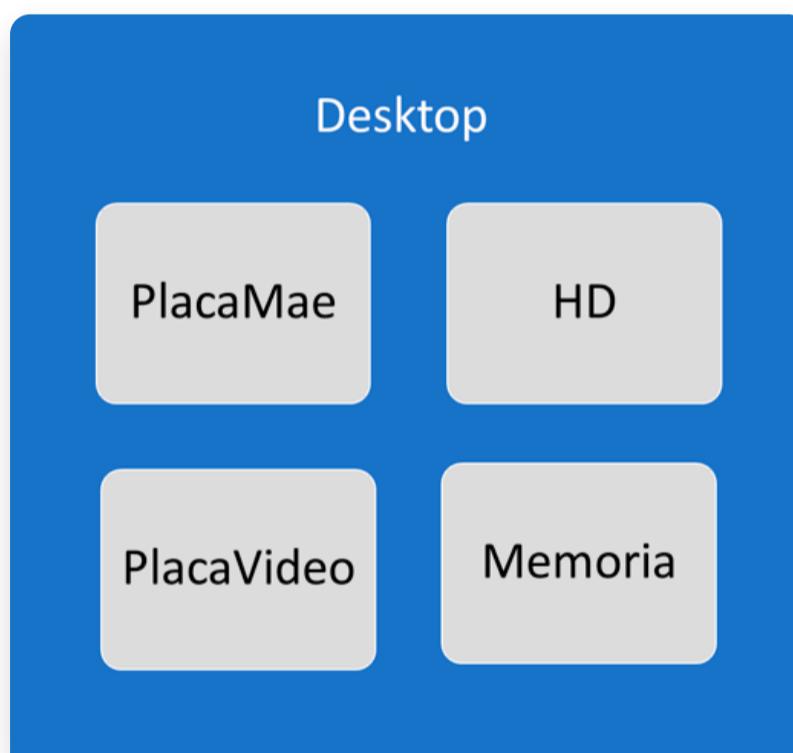
Observe que muitos dos atributos são comuns a todas as classes e seria necessário diferenciar os atributos que possuem o mesmo nome, tal como: modeloPlacaMae; modeloHD, modeloPlacaVideo, modeloMemoria etc.

O problema ainda poderia ser maior, se tivéssemos mais slots de memória, com modeloMemoria0, modeloMemoria1, modeloMemoria2, modeloMemoria3, para quatro slots, por exemplo. Essas dificuldades aumentam bastante o tamanho da classe e a sua complexidade; além disso, neste exemplo, começariamos com 18 atributos, sem levar em consideração a questão dos diferentes slots de memória.

Entretanto, se pensarmos segundo a ótica do particionamento, podemos dividir a classe Desktop, grande e complexa, em classes menores e mais simples. Como sugestão, poderíamos criar as classes que foram apresentadas como exemplo no parágrafo anterior em:

- Placa-mãe (PlacaMae);
- Disco Rígido (HD);
- Placa de vídeo (PlacaVideo);
- Memória (Memoria).

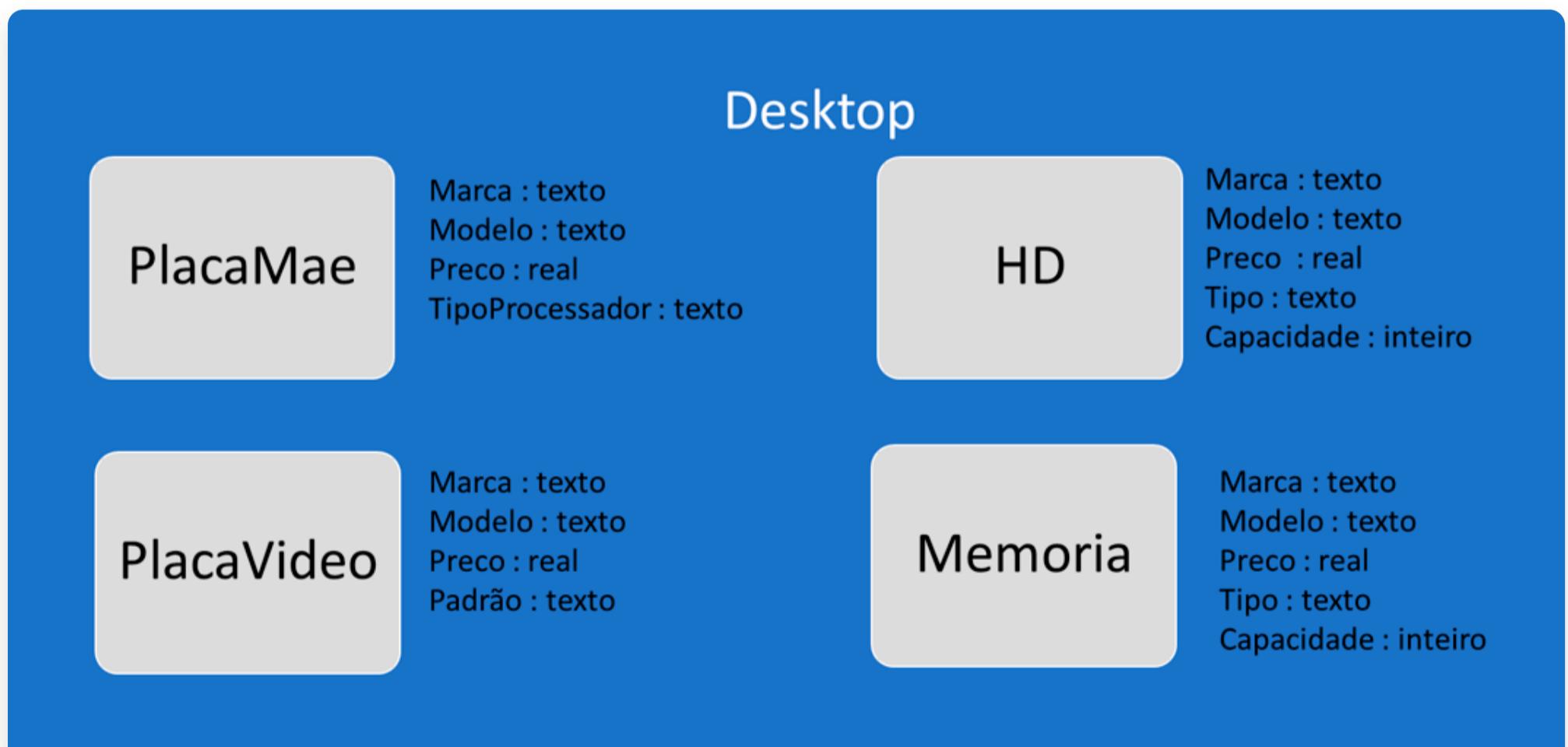
A classe Desktop seria então decomposta (particionada) da seguinte forma:



Vamos determinar poucos atributos para cada uma delas, apenas para entendermos melhor o conceito.

Classe	Atributos
PlacaMae	Marca : texto Modelo : texto Preco : real TipoProcessador : texto
HD	Marca : texto Modelo : texto Preco : real Tipo : texto Capacidade : inteiro
PlacaVideo	Marca : texto Modelo : texto Preco : real Padrão : texto
Memoria	Marca : texto Modelo : texto Preco : real Tipo : texto Capacidade : inteiro

A classe Desktop decomposta ficará da seguinte forma, com seus atributos:



Assim, vamos criar as classes separadamente, dividindo a classe Desktop conforme proposto:

```
public class PlacaMae {  
    public String marca, modelo, tipoProcessador;  
    public double preco;  
}
```

```
public class HD {  
    public String marca, modelo, tipo;  
    public double preco;  
    public int capacidade;  
}
```

```
public class PlacaVideo {  
    public String marca, modelo;  
    public double preco;  
    public int capacidade;  
}
```

```
public class Memoria {  
    public String marca, modelo, tipo;  
    public double preco;  
    public int capacidade;  
}
```

```
public class PlacaVideo {  
    public String marca, modelo;  
    public double preco;  
    public int capacidade;  
}
```

```
public class Memoria {  
    public String marca, modelo, tipo;  
    public double preco;  
    public int capacidade;  
}
```

## Atenção

1. A decomposição da classe Desktop foi feita em quatro classes mais simples;
2. Os nomes dos atributos puderam ser mantidos na forma original, sem que um interfira no outro;
3. Cada classe mais simples se torna mais fácil de codificar;
4. As classes criadas serão mais fáceis para se realizar qualquer tipo de manutenção.

Outro ponto muito importante é que todas essas classes criadas através do particionamento poderão ser reaproveitadas para as classes Notebook e Servidor. Dessa forma, a nossa biblioteca de classes poderá criar novos objetos quando reunida em conjunto.

**Atenção!** Aqui existe uma videoaula, acesso pelo conteúdo online



Fonte: Shutterstock.

## Agregação

A reunião de uma ou mais classes para formar novas classes é chamada de agregação. Uma nova classe pode ser formada por um conjunto de diferentes objetos. Seguindo nosso exemplo, poderíamos reaproveitar as classes PlacaMae, HD, PlacaVideo, e Memória para criar novas classes, como Desktop, Notebook e Servidor, como nos exemplos a seguir:

```
public class PlacaVideo {  
    public String tipoCooler;  
  
    public PlacaMae pm = new PlacaMae();  
    public PlacaVideo pv = new PlacaVideo();  
    public HD hd = new HD();  
    public Memoria me = new Memoria();  
}
```

```
public class Notebook {  
    public double peso;  
  
    public PlacaMae pm = new PlacaMae();  
    public PlacaVideo pv = new PlacaVideo();  
    public HD hd = new HD();  
    public Memoria me = new Memoria();  
}
```

```
public class Servidor {  
    public int numeroPlacasRede;  
  
    public PlacaMae pm = new PlacaMae();  
    public PlacaVideo pv = new PlacaVideo();  
    public HD hd = new HD();  
    public Memoria me = new Memoria();  
}
```

## Atenção

1. Foram incluídos novos atributos para entendermos que a agregação pode incluir não apenas classes, mas também novos atributos;
2. Como toda classe é também um tipo, podemos declarar atributos como do tipo Classe;
3. Cada atributo criado a partir de uma classe (não sendo de tipos básicos) é uma agregação à classe principal, sendo assim, temos as quatro agregações (PlacaMae, HD, PlacaVideo e Memoria) para cada classe principal;
4. As classes Desktop, Notebook e Servidor foram criadas a partir de fragmentos menores, mas todos os atributos originais da classe Desktop estão presentes.

Uma oportunidade se apresenta com o uso da agregação, que não é possível resolver facilmente com a herança.

## Atenção

Não confunda herança com agregação, pois são conceitos diferentes.

Imagine agora a situação dos *slots* de memória: na herança, só poderíamos herdar uma memória, mas, com a agregação, podemos criar quantas memórias quisermos, como no exemplo a seguir. Para a classe Memoria foram criados apenas os métodos de Acesso (Setters e Getters) para facilitar o entendimento:

```

public class Memoria {
    public String marca, modelo, tipo;
    public double preco;
    public int capacidade;
    public String getMarca() {
        return marca;
    }
    public void setMarca( String ma ) {
        if(!ma.isEmpty()) {
            marca = ma;
        }
    }
    public String getModelo() {
        return modelo;
    }
    public void setModelo( String mo ) {
        if(!mo.isEmpty()) {
            modelo = mo;
        }
    }
    public String getTipo() {
        return tipo;
    }
    public void setTipo( String ti ) {
        if(!ti.isEmpty()) {
            tipo = ti;
        }
    }
    public double getPreco() {
        return preco;
    }
    public void setPreco( double pr ) {
        if(pr>0) {
            preco = pr;
        }
    }
    public int getCapacidade() {
        return capacidade;
    }
    public void setCapacidade(int ca) {
        if(ca>0) {
            capacidade = ca;
        }
    }
}

```

```

public class Desktop {
    // atributos da classe
    public String tipoCooler;
    // atributos agregados
    public PlacaMae pm = new PlacaMae();
    public PlacaVideo pv = new PlacaVideo();
    public HD hd = new HD();
    public Memoria slot0 = new Memoria();
    public Memoria slot1 = new Memoria();
    public Memoria slot2 = new Memoria();
    public Memoria slot3 = new Memoria();
}

```

```

public class AppAgregacao {
    public static void main(String[] args) {
        Desktop desk = new Desktop();
        //para usar algum método do objeto criado a partir da classe agregada,
        //devemos usar o identificador do objeto:
        // slot0
        desk.slot0.setMarca("Samsung");
        desk.slot0.setCapacidade(16);
        // slot1
        desk.slot1.setMarca("Kingston");
        desk.slot1.setCapacidade(8);
        // slot2
        desk.slot2.setMarca("Sandisk");
        desk.slot2.setCapacidade(4);
        // slot3
        desk.slot3.setMarca("Crucial");
        desk.slot3.setCapacidade(2);
        // total de memória:
        System.out.println("Memória total: " + (desk.slot0.getCapacidade() + desk.slot1.getCapacidade() +
        desk.slot2.getCapacidade() + desk.slot3.getCapacidade() ) );
    }
}

```

## Atenção

1. O Desktop possui quatro *slots* de memória;
2. Como cada *slot* é um objeto diferente, eles possuem propriedades diferentes, sem que nenhum tenha relação direta com os demais;
3. Não é comum montar um computador dessa forma, mas foram definidas marcas e capacidades diferentes para indicar que os valores armazenados não sofrem influência dos demais objetos agregados.

## Leitura

Vamos analisar o uso dos métodos através de [atributos agregados](#).

Agora que vimos como aplicar os conceitos de agregação e particionamento, podemos aplicar também o conceito de herança e melhorar nossas classes. Vamos aplicar o conceito de herança às nossas classes particionadas para termos uma aplicação mais concisa e mais fácil para realizar manutenções.

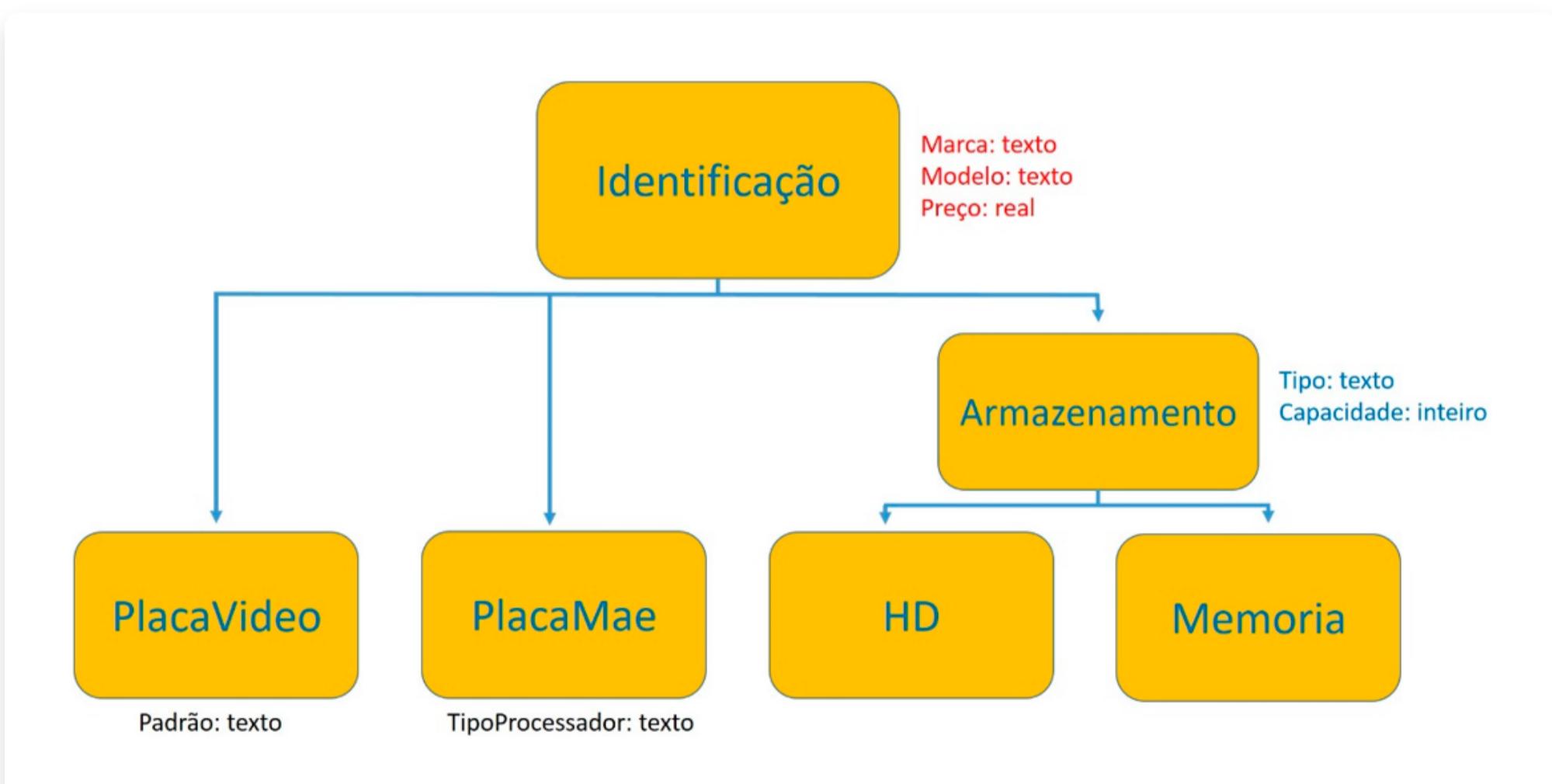
# Aplicação dos conceitos de herança, particionamento e agregação

## Classes Particionadas



Devemos identificar os atributos comuns às classes, sendo que os atributos Marca, Modelo e Preço são comuns a todas as classes. Já os atributos Tipo e Capacidade são comuns apenas às classes HD e Memoria, sendo necessária uma classe intermediária. O atributo TipoProcessador pertence apenas à classe PlacaMae e o atributo Padrão pertence apenas à classe PlacaVideo. As classes HD e Memoria não terão atributos específicos, ficarão apenas nas superclasses.

## Classes Particionadas redefinidas após a aplicação da Herança



## Classe Particionada Identificação:

```
import java.util.Scanner;

public class Identificacao {
    public String marca, modelo;
    public double preco;
    public Identificacao ( ) { }
    public Identificacao ( String ma ) {
        setMarca( ma );
    }
    public Identificacao ( double pr ) {
        setPreco( pr );
    }
    public Identificacao ( String ma, String mo ) {
        setMarca( ma );
        setModelo( mo );
    }
    public Identificacao ( String ma, double pr ) {
        setMarca( ma );
        setPreco( pr );
    }
    public Identificacao ( String ma, String mo, double pr ) {
        setMarca( ma );
        setModelo( mo );
        setPreco( pr );
    }
    public String getMarca() {
        return marca;
    }
    public void setMarca( String ma ) {
        if(!ma.isEmpty()) {
            marca = ma;
        }
    }
    public String getModelo() {
        return modelo;
    }
    public void setModelo( String mo ) {
        if(!mo.isEmpty()) {
            modelo = mo;
        }
    }
    public double getPreco() {
        return preco;
    }
    public void setPreco(double pr) {
        if(pr>0) {
            preco = pr;
        }
    }
    public void cadastrar ( String ma, String mo, double pr ) {
        setMarca( ma );
        setModelo( mo );
        setPreco( pr );
    }
    public void imprimir( ) {
        System.out.println( "Marca      :" + getMarca( ) );
        System.out.println( "Modelo     :" + getModelo( ) );
    }
}
```

```
System.out.println( "Preço      :" + getPreco( ) );
}

public void entradaDados( ) {
    Scanner entrada = new Scanner( System.in );
    System.out.println("Marca      :");
    setMarca( entrada.nextLine( ) );
    System.out.println("Modelo     :");
    setModelo( entrada.nextLine( ) );
    System.out.println("Preço      :");
    setPreco(Double.parseDouble( entrada.nextLine( ) )));
}
}
```

## Classe Particionada Armazenamento:

```
import java.util.Scanner;

public class Armazenamento extends Identificacao{
    public String tipo;
    public int capacidade;
    public Armazenamento( ) {
        super();
    }
    public Armazenamento( String ti, int ca ) {
        setTipo( ti );
        setCapacidade( ca );
    }
    public Armazenamento( String ma, String mo ) {
        super( ma, mo );
    }
    public Armazenamento( String ma, double pr ) {
        super( ma, pr );
    }
    public Armazenamento( String ma, String mo, double pr ) {
        super( ma, mo, pr );
    }
    public Armazenamento(String ma, String mo, double pr, String ti, int ca){
        super( ma, mo, pr );
        setTipo( ti );
        setCapacidade( ca );
    }
    public String getTipo() {
        return tipo;
    }
    public void setTipo( String ti ) {
        if(!ti.isEmpty()) {
            tipo = ti;
        }
    }
    public int getCapacidade() {
        return capacidade;
    }
    public void setCapacidade(int ca) {
        if(ca>0) {
            capacidade = ca;
        }
    }
    public void cadastrar(String ma, String mo, String ti, double pr, int ca){
        super.cadastrar( ma, mo, pr );
        setTipo( ti );
        setCapacidade( ca );
    }
    public void imprimir( ) {
        super.imprimir();
        System.out.println( "Processador :" + getTipo( ) );
        System.out.println( "Capacidade :" + getCapacidade( ) );
    }
    public void entradaDados( ) {
        Scanner entrada = new Scanner( System.in );
        super.entradaDados();
        System.out.println("Tipo      :");
        setTipo( entrada.nextLine( ) );
    }
}
```

```
System.out.println("Capacidade :");
setCapacidade(Integer.parseInt( entrada.nextLine( )));

}
}
```

### Classe Particionada PlacaMae:

```
import java.util.Scanner;

public class PlacaMae extends Identificacao {
    public String tipoProcessador;
    public PlacaMae() { }
    public PlacaMae( String ma, String mo ) {
        super( ma, mo );
    }
    public PlacaMae( String ma, String mo, String tp ) {
        super( ma, mo );
        setTipoProcessador( tp );
    }
    public PlacaMae( String ma, String mo, String tp, double pr ) {
        super( ma, mo, pr );
        setTipoProcessador( tp );
    }
    public String getTipoProcessador() {
        return tipoProcessador;
    }
    public void setTipoProcessador( String tp ) {
        if(!tp.isEmpty()) {
            tipoProcessador = tp;
        }
    }
    public void cadastrar ( String ma, String mo, String tp, double pr ) {
        super.cadastrar( ma, mo, pr );
        setTipoProcessador( tp );
    }
    public void imprimir( ) {
        super.imprimir();
        System.out.println( "Processador :" + getTipoProcessador() );
    }
    public void entradaDados( ) {
        Scanner entrada = new Scanner( System.in );
        super.entradaDados();
        System.out.println("Processador :");
        setTipoProcessador( entrada.nextLine( ) );
    }
}
```

### Classe Particionada HD:

```
public class HD extends Armazenamento {  
    public HD( ) {  
        super();  
    }  
    public HD( String ti, int ca ) {  
        setTipo( ti );  
        setCapacidade( ca );  
    }  
    public HD( String ma, String mo ) {  
        super( ma, mo );  
    }  
    public HD( String ma, double pr ) {  
        super( ma, pr );  
    }  
    public HD( String ma, String mo, double pr ) {  
        super( ma, mo, pr );  
    }  
    public HD( String ma, String mo, double pr, String ti, int ca ) {  
        super( ma, mo, pr );  
        setTipo( ti );  
        setCapacidade( ca );  
    }  
}
```

### Classe Particionada PlacaVideo:

```
import java.util.Scanner;

public class PlacaVideo extends Identificacao {
    public String padrao;
    public PlacaVideo( ) { }
    public PlacaVideo( String ma, String mo ) {
        super( ma, mo );
    }
    public PlacaVideo( String ma, String mo, double pr ) {
        super( ma, mo, pr );
    }
    public PlacaVideo( String ma, String mo, double pr, String pa ) {
        super( ma, mo, pr );
        setPadrao( pa );
    }
    public String getPadrao() {
        return padrao;
    }
    public void setPadrao( String pa ) {
        if( !pa.isEmpty() ) {
            padrao = pa;
        }
    }
    public void cadastrar( String ma, String mo, double pr, String pa ) {
        super.cadastrar(ma, mo, pr);
        setPadrao( pa );
    }
    public void imprimir( ) {
        super.imprimir();
        System.out.println( "Padrão      :" + getPadrao( ) );
    }
    public void entradaDados( ) {
        Scanner entrada = new Scanner( System.in );
        super.entradaDados();
        System.out.println("Padrão      :");
        setPadrao( entrada.nextLine( ) );
    }
}
```

### Classe Particionada Memoria:

```
public class Memoria extends Armazenamento {  
    public Memoria( ) {  
        super();  
    }  
    public Memoria( String ti, int ca ) {  
        setTipo( ti );  
        setCapacidade( ca );  
    }  
    public Memoria( String ma, String mo ) {  
        super( ma, mo );  
    }  
    public Memoria( String ma, double pr ) {  
        super( ma, pr );  
    }  
    public Memoria( String ma, String mo, double pr ) {  
        super( ma, mo, pr );  
    }  
    public Memoria( String ma, String mo, double pr, String ti, int ca ) {  
        super( ma, mo, pr );  
        setTipo( ti );  
        setCapacidade( ca );  
    }  
}
```

Com a evolução das classes particionadas com a aplicação do conceito de herança, foram mantidas as compatibilidades e as classes Desktop, Notebook e Servidor não precisam ser alteradas, assim como a aplicação, que funcionará da mesma forma que no exemplo anterior. As mudanças com a evolução das classes particionadas não afetaram as classes agregadoras, nem a aplicação.

Caso seja necessária alguma alteração específica, basta alterar a classe afetada. Entretanto, se for alguma alteração sobre as classes Memoria e HD, basta alterar a classe Armazenamento; caso seja uma alteração que afete ao mesmo tempo as quatro classes particionadas, basta alterar a superclasse Identificação.

Temos então um melhor controle sobre a aplicação e maior facilidade de manutenção de todo o sistema.

**Atenção!** Aqui existe uma videoaula, acesso pelo conteúdo online

## Atividade

1) Na orientação a objetos, para reunirmos uma ou mais classes, de forma a criar novas classes, aplicamos o conceito de:

- a) Agregação
- b) Polimorfismo
- c) Herança
- d) Encapsulamento
- e) Classes abstratas

## Referências

---

DEITEL, Paul. **Java**: como programar (Biblioteca Virtual). 10. ed. São Paulo: Pearson, 2017.

## Próxima aula

---

- Encapsulamento.

## Explore mais

---

Leia o texto:

Pesquise na internet sites, vídeos e artigos relacionados ao conteúdo visto. Em caso de dúvidas, converse com seu professor online por meio dos recursos disponíveis no ambiente de aprendizagem.