Análise e Projeto Orientados por Objetos

Aula 09 – Padrões GoF (Adapter e Composite)

Edirlei Soares de Lima
<edirlei@iprj.uerj.br>

Padrões GoF

- Criação:
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Estruturais:
  - **Adapter**
  - Bridge
  - **Composite**
  - Decorator
  - Façade
  - Flyweight
  - Proxy
- Comportamentais:
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

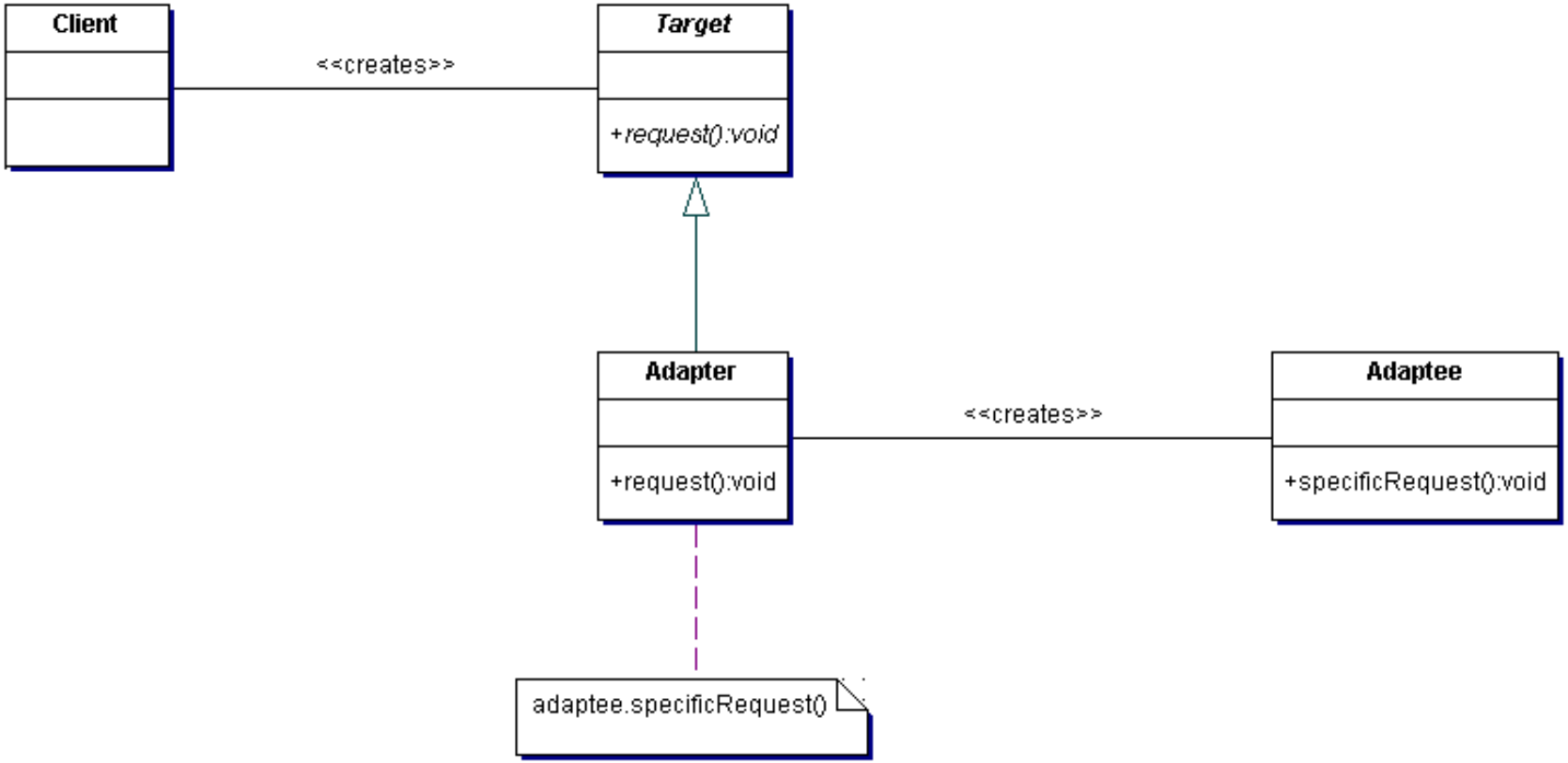
Adapter

- **Intenção:** adaptar um objeto preexistente para uma interface específica com a qual um outro objeto espera se comunicar.

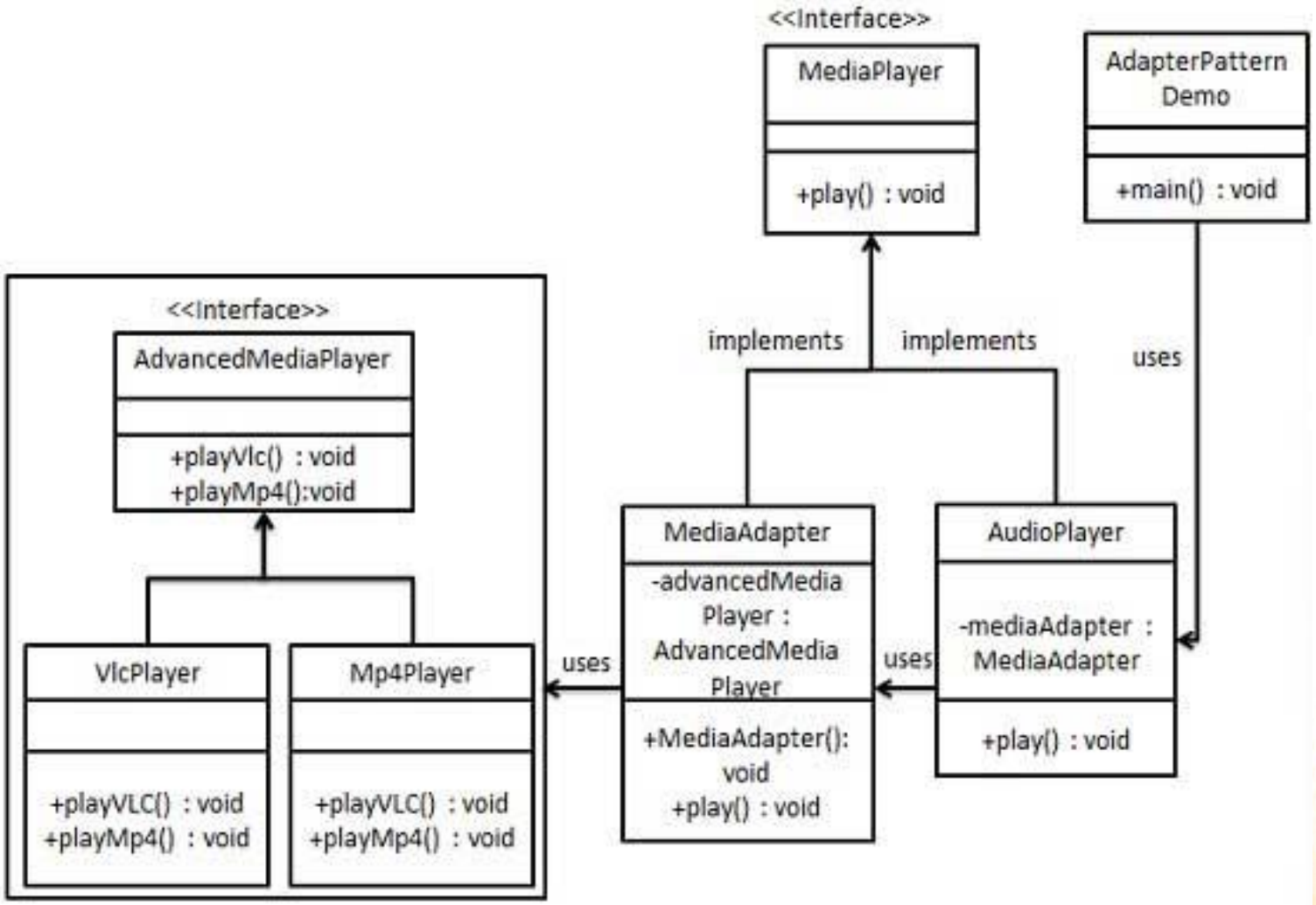
- **Solução:** definir uma classe que serve como um adaptador e que age como um intermediário entre o objeto e seus clientes (utilizar herança ou composição). O adaptador traduz comandos do cliente para o fornecedor e os resultados do fornecedor para o cliente.

- **Exemplo:** Leitor de cartões de memória;

Adapter


Client
<<creates>>
Target
+request():void
Adapter
+request():void
<<creates>>
Adaptee
+specificRequest():void
adaptee.specificRequest()

Adapter – Exemplo



Adapter – Implementação

```
public interface MediaPlayer
{
   public void play(String audioType, String fileName);
}
```

```
public interface AdvancedMediaPlayer
{
   public void playVlc(String fileName);
   public void playMp4(String fileName);
}
```

Adapter – Implementação

```
public class VlcPlayer implements AdvancedMediaPlayer
{

   @Override
   public void playVlc(String fileName)
   {
      System.out.println("Playing vlc file. Name: "+ fileName);
   }

   @Override
   public void playMp4(String fileName)
   {
      //do nothing
   }
}
```

Adapter – Implementação

```
public class Mp4Player implements AdvancedMediaPlayer
{

   @Override
   public void playVlc(String fileName)
   {
      //do nothing
   }

   @Override
   public void playMp4(String fileName)
   {
      System.out.println("Playing mp4 file. Name: "+ fileName);

   }
}
```

Adapter – Implementação

```
public class MediaAdapter implements MediaPlayer {
   AdvancedMediaPlayer advancedMusicPlayer;

   public MediaAdapter(String audioType){
      if(audioType.equalsIgnoreCase("vlc") ){
         advancedMusicPlayer = new VlcPlayer();
      }else if (audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer = new Mp4Player();
      }
   }
   @Override
   public void play(String audioType, String fileName) {
      if(audioType.equalsIgnoreCase("vlc")){
         advancedMusicPlayer.playVlc(fileName);
      }
      else if(audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer.playMp4(fileName);
      }
   }
}
```

Adapter – Implementação

```
public class AudioPlayer implements MediaPlayer {
   MediaAdapter mediaAdapter;

   @Override
   public void play(String audioType, String fileName) {

      if(audioType.equalsIgnoreCase("mp3")){
         System.out.println("Playing mp3 file. Name: " + fileName);
      }else if(audioType.equalsIgnoreCase("vlc") ||
               audioType.equalsIgnoreCase("mp4")){
         mediaAdapter = new MediaAdapter(audioType);
         mediaAdapter.play(audioType, fileName);
      }else{
         System.out.println("Invalid media. " + audioType + " format
                              not supported");
      }
   }
}
```

Adapter – Implementação

```
public static void main(String[] args)
{
    AudioPlayer audioPlayer = new AudioPlayer();

    audioPlayer.play("mp3", "beyond the horizon.mp3");
    audioPlayer.play("mp4", "alone.mp4");
    audioPlayer.play("vlc", "far far away.vlc");
    audioPlayer.play("avi", "mind me.avi");
}
```

Adapter – Aplicabilidade

- Quando se quer usar uma classe já existente e sua **interface não combina** com a esperada pelo cliente;

- Quando se quer criar uma classe reutilizável que **coopera com classes não relacionadas ou não previstas**, isto é, classes que não necessariamente tenham interfaces compatíveis;

- Quando se necessita **usar várias classes existentes**, mas é impraticável adaptar através da transformação de suas interfaces para transformá-las em subclasses de uma mesma classe.

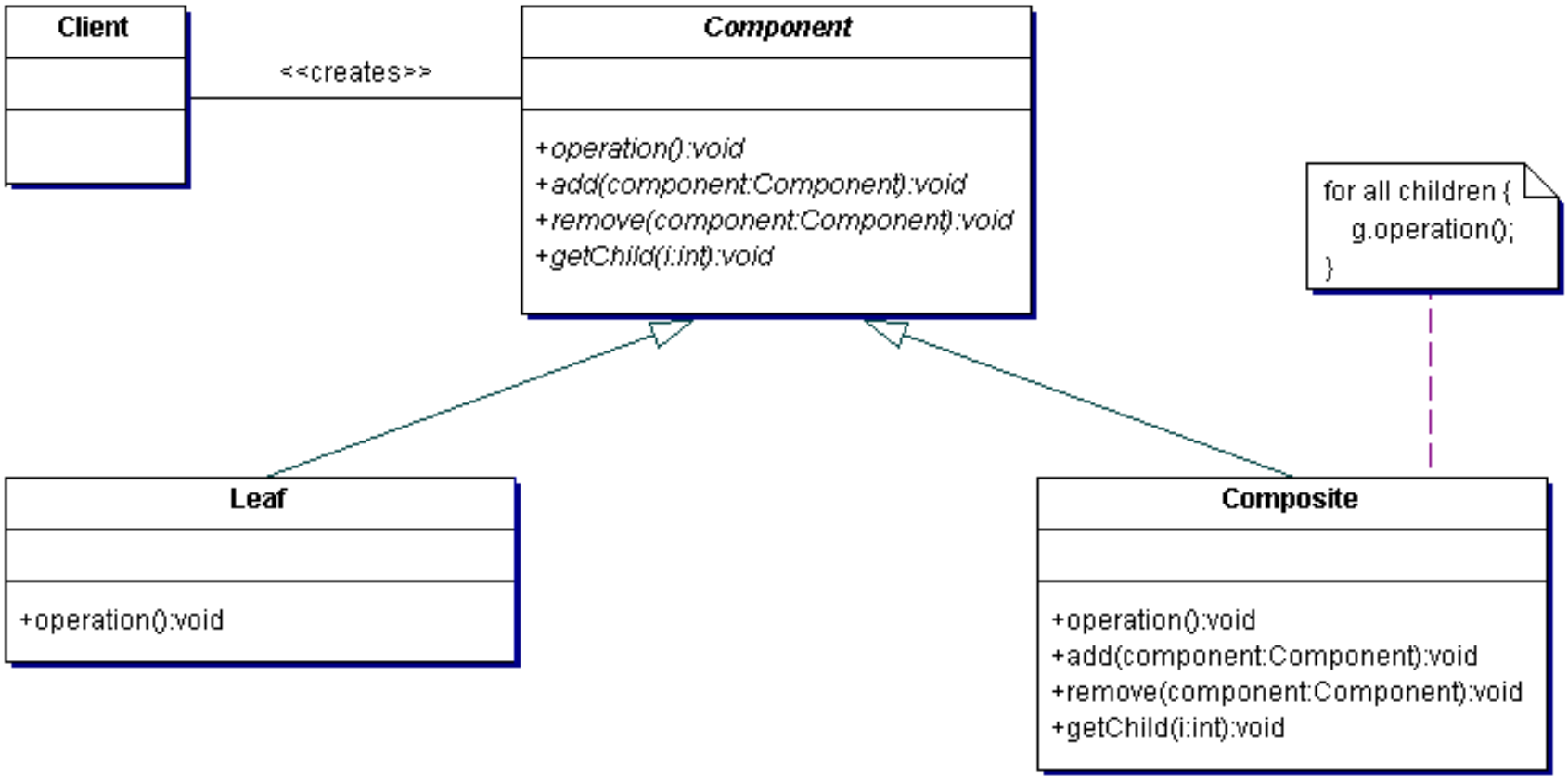
Composite

- São comuns as situações onde temos que lidar com uma coleção de elementos estruturada hierarquicamente (em vez coleções "lineares").

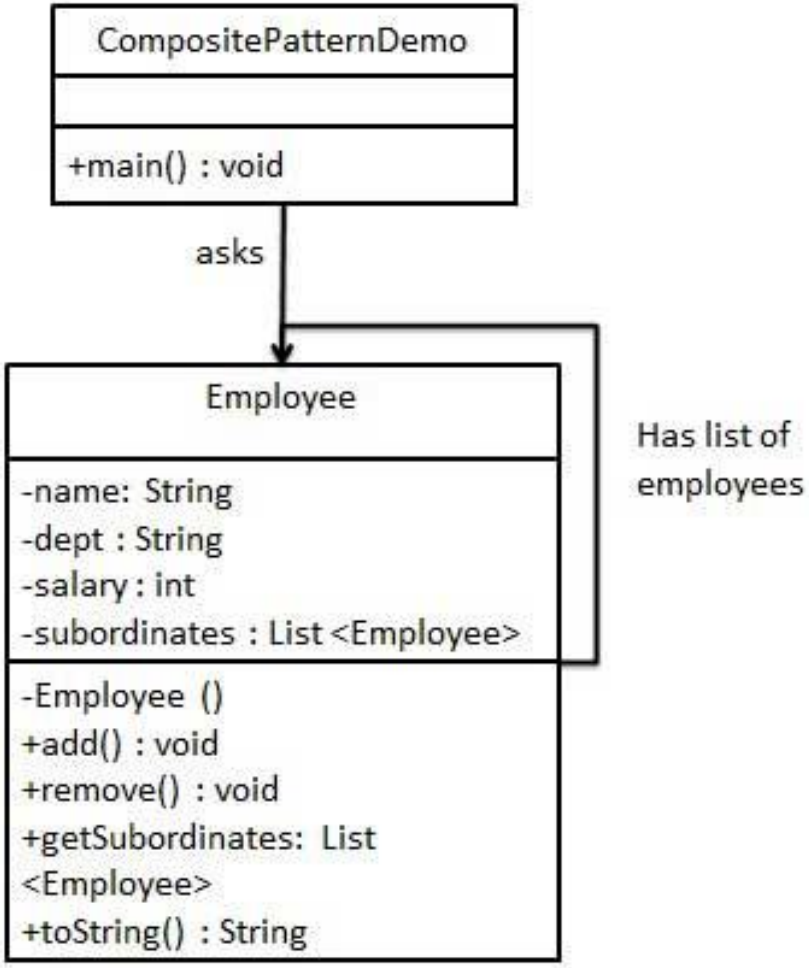
- **Problema:** como criar objetos utilizando partes de tal forma que tanto o objeto todo quanto os objetos parte forneçam a mesma interface para os seus clientes?

- Composições podem cumprir com este requisito e ainda permitir:
  - o tratamento da composição como um todo;
  - ignorar as diferenças entre composições e elementos individuais;
  - a adição transparente de novos tipos a hierarquia;

Composite


Client
<<creates>>
Component
+operation():void
+add(component:Component):void
+remove(component:Component):void
+getChild(i:int):void
for all children {
g.operation();
}
Leaf
+operation():void
Composite
+operation():void
+add(component:Component):void
+remove(component:Component):void
+getChild(i:int):void

Composite – Exemplo



Composite – Implementação

```
public class Employee {
   private String name;
   private int salary;
   private String dept;
   private List<Employee> subordinates;
   public Employee(String name, String dept, int sal) {
      this.name = name; this.dept = dept; this.salary = sal;
      subordinates = new ArrayList<Employee>();
   }
   public void add(Employee e) {
      subordinates.add(e);
   }
   public void remove(Employee e) {
      subordinates.remove(e);
   }
   public List<Employee> getSubordinates(){
     return subordinates;
   }
   public String toString(){
      return ("Employee :[ Name : " + name + ", salary :" + salary+" ]");
   }
}
```

Composite – Implementação

```
public static void main(String[] args) {
  Employee CEO = new Employee("John","CEO", 30000);
  Employee headSales = new Employee("Robert","Head Sales", 20000);
  Employee headMarketing = new Employee("John","Head Marketing", 20000);
  Employee clerk1 = new Employee("Laura","Marketing", 10000);
  Employee clerk2 = new Employee("Bob","Marketing", 10000);
  Employee salesExecutive1 = new Employee("Richard","Sales", 10000);
  Employee salesExecutive2 = new Employee("Rob","Sales", 10000);
  CEO.add(headSales);
  CEO.add(headMarketing);
  headSales.add(salesExecutive1);
  headSales.add(salesExecutive2);
  headMarketing.add(clerk1);
  headMarketing.add(clerk2);
  System.out.println(CEO);
  for (Employee headEmployee : CEO.getSubordinates()) {
    System.out.println(headEmployee);
    for (Employee employee : headEmployee.getSubordinates())
       System.out.println(employee);
  }
}
```